Logique de Séparation en Coq : théorie et pratique

Arthur Charguéraud

Inria & Université de Strasbourg, CNRS, ICube

16 janvier 2020

De la Logique de Hoare à la Logique de Séparation

$$\{H\}\ t\ \{Q\}$$

Triplet de Hoare :

H et Q décrivent l'intégralité de l'état mémoire

Triplet de Logique de Séparation :

H et Q décrivent une partie de l'état mémoire suffisante pour exécuter t

De la Logique de Hoare à la Logique de Séparation

$$\{H\}\ t\ \{Q\}$$

Triplet de Hoare :

H et Q décrivent l'intégralité de l'état mémoire

Triplet de Logique de Séparation :

H et Q décrivent une partie de l'état mémoire suffisante pour exécuter t

Règle d'encadrement, dite règle « frame »

$$\frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \star H'\}}$$

La Logique de Séparation en pratique

1. **Preuves automatiques**

2. **Preuves semi-automatiques**

3. **Preuves interactives**

La Logique de Séparation en pratique

1. Preuves automatiques

- L'utilisateur fournit son code.
- L'outil localise des bugs potentiels.
- Exemple : *Infer* (Facebook).

2. Preuves semi-automatiques

3. Preuves interactives

La Logique de Séparation en pratique

1. **Preuves automatiques**
   - L'utilisateur fournit son code.
   - L'outil localise des bugs potentiels.
   - Exemple : *Infer* (Facebook).
2. **Preuves semi-automatiques**
   - L'utilisateur fournit spécifications et annotations.
   - L'outil s'appuie sur des solveurs automatiques (SMT).
   - Exemples : *VeriFast* (KU Leuven) et *Viper* (ETH Zurich).
3. **Preuves interactives**

La Logique de Séparation en pratique

1. **Preuves automatiques**
   - L'utilisateur fournit son code.
   - L'outil localise des bugs potentiels.
   - Exemple : *Infer* (Facebook).

2. **Preuves semi-automatiques**
   - L'utilisateur fournit spécifications et annotations.
   - L'outil s'appuie sur des solveurs automatiques (SMT).
   - Exemples : *VeriFast* (KU Leuven) et *Viper* (ETH Zurich).

3. **Preuves interactives**
   - L'utilisateur fournit les spécifications et les preuves.
   - Les preuves sont réalisées dans un assistant de preuve (tel que Coq).

Logique de Séparation dans les assistants de preuve

Projet	**Outil**	**Langage**	**Application principale**
Ynot	Coq	ML	Structures de données
Sel4	Isabelle	C, assembleur	Micro-noyau pour OS
Flint	Coq	Assembleur	OS (système d'exploitation)
Bedrock	Coq	Assembleur	OS pour robots
VST	Coq	C	Protocoles concurrents
CakeML	HOL	SML	Fonctionalités d'un runtime ML
CFML	Coq	OCaml	Structures de données, algorithmes

(liste non exhaustive)

Avantages

- Expressivité presque sans limite de la logique d'ordre supérieur
- Cadre unifié pour prouver le code et les lemmes mathématiques
- Scripts de preuves relativement faciles à maintenir
- Haut degré de confiance dans la correction des outils

Construction

Tous ces outils sont construits selon le même schéma

1. Formalisation de la sémantique du langage source
2. Définition des prédicats de la Logique de Séparation
3. Énoncé des triplets et preuve des règles de raisonnement
4. Infrastructure permettant des preuves concises

Construction

Tous ces outils sont construits selon le même schéma

1. Formalisation de la sémantique du langage source
2. Définition des prédicats de la Logique de Séparation
3. Énoncé des triplets et preuve des règles de raisonnement
4. Infrastructure permettant des preuves concises

Objectif du séminaire : présenter cette construction pour

- un langage de programmation impératif minimaliste,
- la variante la plus simple possible de la Logique de Séparation.

Étape 1

Syntaxe et sémantique du langage source

Grammaire du langage source

```
Definition var : Type := string.

Definition loc : Type := nat.

Definition null : loc := 0.


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

with prim : Type :=
  | val_get : prim
  | val_set : prim
  | val_ref : prim
  | val_free : prim
  | val_eq : prim
  | val_add : prim
  ..
```

Saisie des programmes à vérifier

```
let rec mlength p = (* longueur d'une liste mutable en style C *)
  if p == null
    then 0
    else 1 + mlength p.tail
```

Saisie des programmes à vérifier

```
let rec mlength p = (* longueur d'une liste mutable en style C *)
  if p == null
    then 0
    else 1 + mlength p.tail
```

Définition Coq correspondante

```
Definition mlength : val :=
  val_fix "f" "p" (
     trm_if (trm_app (trm_app val_eq (trm_var "p")) (val_loc null))
            (val_int 0)
            (trm_app (trm_app val_add (val_int 1)) (trm_app (trm_var "f")
              (trm_app (val_get_field tail) (trm_var "p"))))).
```

Saisie des programmes à vérifier

```
let rec mlength p = (* longueur d'une liste mutable en style C *)
  if p == null
    then 0
    else 1 + mlength p.tail
```

Définition Coq correspondante

```
Definition mlength : val :=
  val_fix "f" "p" (
     trm_if (trm_app (trm_app val_eq (trm_var "p")) (val_loc null))
            (val_int 0)
            (trm_app (trm_app val_add (val_int 1)) (trm_app (trm_var "f")
              (trm_app (val_get_field tail) (trm_var "p"))))).
```

Définition Coq avec notations et coercions

```
Definition mlength : val :=
  VFix 'f 'p :=
    If_ 'p '= null
      Then 0
      Else 1 '+ 'f ('p'.tail).
```

Sémantique du langage source, à grands pas

```
Definition state : Type := fmap loc val.

Inductive eval : state → trm → state → val → Prop :=
  | eval_val : ∀s v,
      eval s (trm_val v) s v
  | eval_let : ∀s1 s2 s3 x t1 t2 v1 v,
      eval s1 t1 s2 v1 →
      eval s2 (subst x v1 t2) s3 v →
      eval s1 (trm_let x t1 t2) s3 v
  | eval_get : ∀s l v,
      Fmap.indom s l →
      eval s (val_get (val_loc l)) s (Fmap.read s l)
  ...
```

Étape 2

Prédicats et implications en Logique de Séparation

Prédicats de la Logique de Séparation

```
Definition hprop : Type := state → Prop.
```

Prédicats de la Logique de Séparation

```
Definition hprop : Type := state → Prop.

Definition hempty : hprop := (* noté [] *)
  fun h ⇒ h = Fmap.empty.
```

Prédicats de la Logique de Séparation

```
Definition hprop : Type := state → Prop.

Definition hempty : hprop := (* noté [] *)
  fun h ⇒ h = Fmap.empty.

Definition hpure (P:Prop) : hprop := (* noté [P] *)
  fun h ⇒ h = Fmap.empty ∧ P.
```

Prédicats de la Logique de Séparation

```
Definition hprop : Type := state → Prop.

Definition hempty : hprop := (* noté [] *)
  fun h ⇒ h = Fmap.empty.

Definition hpure (P:Prop) : hprop := (* noté [P] *)
  fun h ⇒ h = Fmap.empty ∧ P.

Definition hsingle (p:loc) (v:val) : hprop := (* noté (p↦ v) *)
  fun h ⇒ h = Fmap.single p v ∧ p ≠null.
```

Prédicats de la Logique de Séparation

```
Definition hprop : Type := state → Prop.

Definition hempty : hprop := (* noté [] *)
  fun h ⇒ h = Fmap.empty.

Definition hpure (P:Prop) : hprop := (* noté [P] *)
  fun h ⇒ h = Fmap.empty ∧ P.

Definition hsingle (p:loc) (v:val) : hprop := (* noté (p↦ v) *)
  fun h ⇒ h = Fmap.single p v ∧ p ≠null.

Definition hstar (H1 H2:hprop) : hprop := (* noté (H1 ⋆H2) *)
  fun h ⇒ ∃h1 h2, h = Fmap.union h1 h2
                  ∧ Fmap.disjoint h1 h2
                  ∧ H1 h1
                  ∧ H2 h2.
```

Prédicats de la Logique de Séparation

```
Definition hprop : Type := state → Prop.

Definition hempty : hprop := (* noté [] *)
  fun h ⇒ h = Fmap.empty.

Definition hpure (P:Prop) : hprop := (* noté [P] *)
  fun h ⇒ h = Fmap.empty ∧ P.

Definition hsingle (p:loc) (v:val) : hprop := (* noté (p↦ v) *)
  fun h ⇒ h = Fmap.single p v ∧ p ≠null.

Definition hstar (H1 H2:hprop) : hprop := (* noté (H1 ⋆H2) *)
  fun h ⇒ ∃h1 h2, h = Fmap.union h1 h2
                  ∧ Fmap.disjoint h1 h2
                  ∧ H1 h1
                  ∧ H2 h2.

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

Relation d'ordre sur les prédicats

Implication point-à-point

```
Definition himpl (H1 H2:hprop) : Prop := (* noté H1 ⊢ H2 *)
  ∀h, H1 h → H2 h.
```

Relation d'ordre sur les prédicats

Implication point-à-point

```
Definition himpl (H1 H2:hprop) : Prop := (* noté H1 ⊢ H2 *)
  ∀h, H1 h → H2 h.
```

Une relation d'ordre

```
Lemma himpl_reflexive :
  H ⊢ H.

Lemma himpl_transitive :
  (H1 ⊢ H2) →
  (H2 ⊢ H3) →
  (H1 ⊢ H3).
```

```
Lemma himpl_antisymmetric :
  (H1 ⊢ H2) →
  (H2 ⊢ H1) →
  H1 = H2.
```

Relation d'ordre sur les prédicats

Implication point-à-point

```
Definition himpl (H1 H2:hprop) : Prop := (* noté H1 ⊢ H2 *)
  ∀h, H1 h → H2 h.
```

Une relation d'ordre

```
Lemma himpl_reflexive :
  H ⊢ H.

Lemma himpl_transitive :
  (H1 ⊢ H2) →
  (H2 ⊢ H3) →
  (H1 ⊢ H3).
```

```
Lemma himpl_antisymmetric :
  (H1 ⊢ H2) →
  (H2 ⊢ H1) →
  H1 = H2.

Axiom predicate_extensionality :
  ∀(A:Type) (P1 P2:A→ Prop),
  (∀ x, P1 x ↔ P2 x) →
  P1 = P2.
```

Propriétés fondamentales de l'étoile

```
Lemma hstar_associative :
  (H1 ⋆ H2) ⋆ H3 = H1 ⋆ (H2 ⋆ H3).

Lemma hstar_commutative :
   H1 ⋆ H2 = H2 ⋆ H1.

Lemma hstar_hempty_neutral :
  [] ⋆ H = H.

Lemma hstar_hexists_distrib :
  (∃ x, J x) ⋆ H = ∃x, (J x ⋆ H).

Lemma hstar_monotone :
  H1 ⊢ H1' →
  (H1 ⋆ H2) ⊢ (H1' ⋆ H2).
```

Description des postconditions

Une précondition décrit l'état d'entrée

```
H : state → Prop (* = hprop *)
```

Une postcondition décrit l'état de sortie et la valeur de retour

```
Q : val → state → Prop (* = val→ hprop *)
```

Description des postconditions

Une précondition décrit l'état d'entrée

```
H : state → Prop (* = hprop *)
```

Une postcondition décrit l'état de sortie et la valeur de retour

```
Q : val → state → Prop (* = val→ hprop *)
```

Généralisation de l'étoile et de l'implication

```
                                                    (* noté Q ⋆H *)
Definition qstar (Q:val→ hprop) (H:hprop) : val→ hprop :=
  fun (v:val) ⇒ Q v ⋆ H.

Definition qimpl (Q1 Q2:val→ hprop) := (* noté Q1 ⊢ Q2 *)
  ∀(v:val), Q1 v ⊢ Q2 v.
```

Étape 3

Définition des triplets et preuve des règles de raisonnement

Définition des triplets, en correction forte

$$\{H\}\ t\ \{Q\}$$

Triplet de la Logique de Hoare

```
Definition hoare (t:trm) (H:hprop) (Q:val→hprop) : Prop :=
  ∀(s:state), H s →
  ∃(s':state) (v:val), eval s t s' v ∧ Q v s'.
```

Définition des triplets, en correction forte

$$\{H\}\ t\ \{Q\}$$

Triplet de la Logique de Hoare

```
Definition hoare (t:trm) (H:hprop) (Q:val→hprop) : Prop :=
  ∀(s:state), H s →
  ∃(s':state) (v:val), eval s t s' v ∧ Q v s'.
```

Triplet de la Logique de Séparation

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

Définition des triplets, en correction forte

$$\{H\}\ t\ \{Q\}$$

Triplet de la Logique de Hoare

```
Definition hoare (t:trm) (H:hprop) (Q:val→hprop) : Prop :=
  ∀(s:state), H s →
  ∃(s':state) (v:val), eval s t s' v ∧ Q v s'.
```

Triplet de la Logique de Séparation

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

Exemple

$$\{p \mapsto n\}\ (\texttt{incr}\ p)\ \{\lambda\_.\ p \mapsto (n+1)\}$$

```
Lemma triple_incr : ∀(p:loc) (n:int),
  triple (incr p) (p ↦ n) (fun _ ⇒ p ↦ (n+1)).
```

Règles structurelles de la Logique de Séparation

Règles principales

```
Lemma consequence_rule :
  triple t H1 Q1 →
  H2 ⊢ H1 →
  Q1 ⊢. Q2 →
  triple t H2 Q2.
```

```
Lemma frame_rule :
  triple t H Q →
  triple t (H ⋆ H') (Q ⋆. H').
```

Règles structurelles de la Logique de Séparation

Règles principales

```
Lemma consequence_rule :
  triple t H1 Q1 →
  H2 ⊢ H1 →
  Q1 ⊢. Q2 →
  triple t H2 Q2.
```

```
Lemma frame_rule :
  triple t H Q →
  triple t (H ⋆ H') (Q ⋆ H').
```

Règles d'extraction

```
Lemma extract_hpure :
  (P → triple t H Q) →
  triple t ([P] ⋆ H) Q.
```

```
Lemma extract_hexists  :
  (∀ x, triple t (J x) Q) →
  triple t (∃ x, J x) Q.
```

Règles de raisonnement, exemple de la séquence

Règle en Logique de Hoare

```
Lemma hoare_seq :
  hoare t1 H (fun v ⇒ H') →
  hoare t2 H' Q →
  hoare (trm_seq t1 t2) H Q.
```

Règle en Logique de Séparation

```
Lemma triple_seq :
  triple t1 H (fun v ⇒ H') →
  triple t2 H' Q →
  triple (trm_seq t1 t2) H Q.
```

Règles spécificiant les opérations primitives

```
Lemma triple_get :
  triple (val_get p)
    (p ↦ v)
    (fun r ⇒ [r = v] ⋆ (p ↦ v))
```

```
Lemma triple_set :
  triple (val_set p v')
    (p ↦ v)
    (fun _ ⇒ p ↦ v')
```

Règles spécificiant les opérations primitives

```
Lemma triple_get :
  triple (val_get p)
    (p ↦ v)
    (fun r ⇒ [r = v] ⋆ (p ↦ v))
```

```
Lemma triple_set :
  triple (val_set p v')
    (p ↦ v)
    (fun _ ⇒ p ↦ v')
```

```
Lemma triple_ref :
  triple (val_ref v)
    []
    (fun r ⇒ ∃p, [r = val_loc p] ⋆ p ↦ v)
```

```
Lemma triple_free :
  triple (val_free p)
    (p ↦ v)
    (fun _ ⇒ [])
```

Démo

Preuve d'un programme « à la main » en Logique de Séparation

Étape 4

Infrastructure pour rendre les preuves plus concises

Formules caractéristiques

Calcul de plus faible précondition

- Vise une logique de Hoare
- Vise du code annoté avec ses invariants

Calcul de formule caractéristique

- Vise la Logique de Séparation, avec la règle d'encadrement
- Vise du code sans aucune annotation

Formules caractéristiques

Calcul de plus faible précondition

- Vise une logique de Hoare
- Vise du code annoté avec ses invariants

Calcul de formule caractéristique

- Vise la Logique de Séparation, avec la règle d'encadrement
- Vise du code sans aucune annotation

Défis techniques pour le générateur

- définition structurellement récursive, qui s'évalue dans Coq
- produisant des formule lisibles par l'utilisateur

Plan d'attaque pour le générateur

1. **Pour gérer l'absence d'annotations**
   `wpgen` s'appuie sur la notion de `wp` sémantique.
2. **Pour supporter la règle d'encadrement**
   `wpgen` intègre un prédicat nommé `mkstruct`.
3. **Pour être structurellement récursif**
   `wpgen` effectue les substitutions de manière paresseuse.

Plan d'attaque pour le générateur

1. **Pour gérer l'absence d'annotations**
   `wpgen` s'appuie sur la notion de `wp` sémantique.
2. **Pour supporter la règle d'encadrement**
   `wpgen` intègre un prédicat nommé `mkstruct`.
3. **Pour être structurellement récursif**
   `wpgen` effectue les substitutions de manière paresseuse.
4. **Pour la lisibilité de sa sortie**
   `wpgen` utilise des définitions intermédiaires et des notations.

Plus faible précondition sémantique

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

Caractérisation 1

```
Parameter wp_pre :
  triple t (wp t Q) Q.
```

```
Parameter wp_weakest :
  triple t H Q →
  H ⊢ wp t Q.
```

Plus faible précondition sémantique

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

Caractérisation 1

```
Parameter wp_pre :
  triple t (wp t Q) Q.
```

```
Parameter wp_weakest :
  triple t H Q →
  H ⊢ wp t Q.
```

Caractérisation 2

```
Parameter wp_equiv :
  (H ⊢ wp t Q) ↔ (triple t H Q).
```

Plus faible précondition sémantique

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

Caractérisation 1

```
Parameter wp_pre :
  triple t (wp t Q) Q.
```

```
Parameter wp_weakest :
  triple t H Q →
  H ⊢ wp t Q.
```

Caractérisation 2

```
Parameter wp_equiv :
  (H ⊢ wp t Q) ↔ (triple t H Q).
```

Caractérisation 3

```
Definition wp (t:trm) (Q:val→hprop) : hprop :=
  ∃(H:hprop), H ⋆ [triple t H Q].
```

Logique de Séparation en style wp

Règles de raisonnement pour les termes

```
Lemma wp_seq :
  wp t1 (fun v ⇒ wp t2 Q) ⊢ wp (trm_seq t1 t2) Q.
```

Deux règles structurelles suffisent

```
Lemma wp_monotone :
  Q1 ⊢ Q2 →
  wp t Q1 ⊢ wp t Q2.
```

```
Lemma wp_frame :
  (wp t Q) ⋆ H ⊢ wp t (Q ⋆ H).
```

Définition du générateur (1/5)

Calcul de plus faible précondition pour termes non annotés

```
Fixpoint wpgen (t:trm) (Q:val→hprop) : hprop :=
  match t with
  | trm_val v ⇒ Q v
  | trm_var x ⇒ [False]
  | trm_app v1 v2 ⇒ wp t Q
  | trm_let x t1 t2 ⇒ wpgen t1 (fun v ⇒ wpgen (subst x v t2) Q)
  ...
  end.
```

Définition du générateur (2/5)

Reformulation avec contexte, pour rendre la terminaison évidente

```
Definition ctx := list (var * val).

Fixpoint wpgen (E:ctx) (t:trm) (Q:val→ hprop) : hprop :=
  match t with
  | trm_val v ⇒ Q v
  | trm_var x ⇒
      match lookup x E with
      | Some v ⇒ Q v
      | None ⇒ [False]
      end
  | trm_app v1 v2 ⇒ wp t Q
  | trm_let x t1 t2 ⇒ wpgen E t1 (fun v ⇒ wpgen ((x,v)::E) t2 Q)
  ...
  end.
```

Définition du générateur (3/5)

Inversion du `match` et du `fun Q ⇒ ..`

```
Fixpoint wpgen (E:ctx) (t:trm) : (val→hprop)→hprop :=
  match t with
  | trm_val v ⇒ fun Q ⇒ Q v
  | trm_var x ⇒ fun Q ⇒
      match lookup x E with
      | Some v ⇒ Q v
      | None ⇒ [False]
      end
  | trm_app v1 v2 ⇒ fun Q ⇒ wp t Q
  | trm_let x t1 t2 ⇒ fun Q ⇒
      wpgen E t1 (fun v ⇒ wpgen ((x,v)::E) t2 Q)
  ...
  end.
```

Définition du générateur (3/5)

Inversion du `match` et du `fun Q ⇒ ..`

```
Fixpoint wpgen (E:ctx) (t:trm) : (val→hprop)→hprop :=
  match t with
  | trm_val v ⇒ fun Q ⇒ Q v
  | trm_var x ⇒ fun Q ⇒
      match lookup x E with
      | Some v ⇒ Q v
      | None ⇒ [False]
      end
  | trm_app v1 v2 ⇒ fun Q ⇒ wp t Q
  | trm_let x t1 t2 ⇒ fun Q ⇒
      wpgen E t1 (fun v ⇒ wpgen ((x,v)::E) t2 Q)
  ...
  end.

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

Définition du générateur (4/5)

Insertion des définitions auxiliaires

```
Fixpoint wpgen (E:ctx) (t:trm) : formula :=
  match t with
  | trm_val v ⇒ wpgen_val v
  | trm_var x ⇒ wpgen_var E x
  | trm_app v1 v2 ⇒ wp t
  | trm_let x t1 t2 ⇒
      wpgen_let (wpgen E t1) (fun v ⇒ wpgen ((x,v)::E) t2)
  ...
  end.
```

Exemple de définition auxiliaire

`wpgen_let F F'` est une définition pour `fun Q ⇒ F (fun v ⇒ F' v Q)`.

Définition du générateur (4/5)

Insertion des définitions auxiliaires

```
Fixpoint wpgen (E:ctx) (t:trm) : formula :=
  match t with
  | trm_val v ⇒ wpgen_val v
  | trm_var x ⇒ wpgen_var E x
  | trm_app v1 v2 ⇒ wp t
  | trm_let x t1 t2 ⇒
      wpgen_let (wpgen E t1) (fun v ⇒ wpgen ((x,v)::E) t2)
  ...
  end.
```

Exemple de définition auxiliaire

`wpgen_let F F'` est une définition pour `fun Q ⇒ F (fun v ⇒ F' v Q)`.

`Let v := F1 in F2` est une notation pour `wpgen_let F1 (fun v ⇒ F2)`.

`wpgen (trm_let x t1 t2)` s'affiche sous la forme `Let x := F1 in F2`.

Définition du générateur (5/5)

Intégration de la règle d'encadrement

```
Definition mkstruct : formula→ formula := ...

Fixpoint wpgen (E:ctx) (t:trm) : formula :=
  mkstruct (match t with
    | trm_val v ⇒ wpgen_val v
    | trm_var x ⇒ wpgen_var E x
    | trm_app v1 v2 ⇒ wp (isubst E t)
    | trm_let x t1 t2 ⇒
       wpgen_let (wpgen E t1) (fun v ⇒ wpgen ((x,v)::E) t2)
    ...
    end).
```

Définition de mkstruct

Propriétés nécessaires

```
Parameter mkstruct_erase :
  F Q ⊢ mkstruct F Q.

Parameter mkstruct_monotone :
  Q1 ⊢ Q2 →
  mkstruct F Q1 ⊢ mkstruct F Q2.

Parameter mkstruct_frame :
  (mkstruct F Q) ⋆ H ⊢ mkstruct F (Q ⋆ H).
```

Définition de mkstruct

Propriétés nécessaires

```
Parameter mkstruct_erase :
  F Q ⊢ mkstruct F Q.

Parameter mkstruct_monotone :
  Q1 ⊢* Q2 →
  mkstruct F Q1 ⊢ mkstruct F Q2.

Parameter mkstruct_frame :
  (mkstruct F Q) ⋆ H ⊢ mkstruct F (Q ⋆* H).
```

Réalisation

```
Definition mkstruct (F:formula) : formula :=
  fun (Q:val→ hprop) ⇒ ∃Q1 H, (F Q1) ⋆ H ⋆ [Q1 ⋆* H ⊢* Q].
```

Preuve du générateur de formules caractéristiques

Théorème de correction

```
Lemma wpgen_sound :
  wpgen nil t Q ⊢ wp t Q.
```

Preuve du générateur de formules caractéristiques

Théorème de correction

```
Lemma wpgen_sound :
  wpgen nil t Q ⊢ wp t Q.
```

Rappel de l'équivalence

```
Parameter wp_equiv :
  (H ⊢ wp t Q) ↔ (triple t H Q).
```

Utilisation du générateur

```
Lemma triple_of_wpgen :
  H ⊢ wpgen nil t Q →
  triple t H Q.
```

Zoom sur deux techniques : les encodeurs et la baguette magique

Technique des encodeurs

Intérêt des encodeurs

```
Lemma triple_ref :
  triple (val_ref v) [] (fun (r:val) ⇒ ∃(p:loc), [r = val_loc p] ⋆ p ↦ v)

Lemma Triple_ref :
  Triple (val_ref v) [] (fun (p:loc) ⇒ p ↦ v)
```

Technique des encodeurs

Intérêt des encodeurs

```
Lemma triple_ref :
  triple (val_ref v) [] (fun (r:val) ⇒ ∃(p:loc), [r = val_loc p] ⋆ p ↦ v)

Lemma Triple_ref :
  Triple (val_ref v) [] (fun (p:loc) ⇒ p ↦ v)
```

Classe de types des encodeurs

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

  Instance Enc_loc : Enc loc := Build_Enc val_loc.
```

Technique des encodeurs

Intérêt des encodeurs

```
Lemma triple_ref :
  triple (val_ref v) [] (fun (r:val) ⇒ ∃(p:loc), [r = val_loc p] ⋆ p ↦ v)

Lemma Triple_ref :
  Triple (val_ref v) [] (fun (p:loc) ⇒ p ↦ v)
```

Classe de types des encodeurs

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

  Instance Enc_loc : Enc loc := Build_Enc val_loc.
```

Triplets avec encodeurs

```
  Definition Triple (t:trm) (H:hprop) (A:Type) {Enc A} (Q:A→ hprop) :=
    triple t H (fun (v:val) ⇒ ∃(V:A), [v = enc V] ⋆ Q V).
```

La baguette magique

Intuition

```
Definition hwand (H1 H2:hprop) : hprop := ... (* noté H1 -* H2 *)

Parameter hwand_elim : H1 ⋆ (H1 -⋆ H2) ⊢ H2.
```

Une définition possible

```
Definition hwand (H1 H2:hprop) : hprop :=
  ∃H0, H0 ⋆ [H1 ⋆ H0 ⊢ H2].
```

Généralisation aux postconditions : $Q1 \mathrel{-\!\!\dot{\star}} Q2$

```
Definition qwand (Q1 Q2:val→ hprop) : hprop :=
  ∃H0, H0 ⋆ [Q1 ⋆̇ H0 ⊢̇ Q2].
```

Règles structurelles avec la baguette magique

Sans la baguette

```
Lemma consequence_frame_rule :
  triple t H1 Q1 →
  H ⊢ H1 ⋆ H2 →
  Q1 ⋆· H2 ⊢· Q →
  triple t H Q.
```

Avec la baguette

```
Lemma ramified_frame_rule :
  triple t H1 Q1 →
  H ⊢ H1 ⋆ (Q1 -⋆· Q) →

  triple t H Q.
```

Règles structurelles avec la baguette magique

Sans la baguette

```
Lemma consequence_frame_rule :
  triple t H1 Q1 →
  H ⊢ H1 ⋆ H2 →
  Q1 ⋆∴ H2 ⊢∴ Q →
  triple t H Q.
```

Avec la baguette

```
Lemma ramified_frame_rule :
  triple t H1 Q1 →
  H ⊢ H1 ⋆ (Q1 -⋆∴ Q) →

  triple t H Q.
```

Avec la baguette en style wp

```
Lemma wp_ramified :
  (wp t Q1) ⋆ (Q1 -⋆∴ Q2) ⊢ (wp t Q2).
```

Démo

Vérification de programmes avec `wpgen` et tactiques

Conclusions et perspectives

Résumé de la construction

1. Syntaxe avec val et trm, et sémantique avec eval
2. Prédicats [] et [P] et `p` $\mapsto$ `v` et `H1` $*$ `H2` et $\exists$`x`, `H`, avec $\vdash$ et $\models$
3. Triplets `hoare` et `triple`, énoncés et preuves des règles
4. Infrastructure : `wp`, `wpgen`, $-\!\!\star$, `Enc`, x-tactiques

L'outil CFML

Extensions du langage

- Boucles `for` et `while`
- Récursion mutuelle
- Structures et tableaux
- Types algébriques et filtrage
- Foncteurs simples

Extensions de la logique

- Prédicats affines, pour refléter l'action du ramasse-miette (GC)
- Crédits temps, pour l'analyse de complexité amortie asymptotique

Exemple de preuve CFML

Problème

- Détection incrémentale de cycle

Algorithme

- Bender, Fineman, Gilbert, et Tarjan (2016)
- Complexité $O(m \cdot \min(m^{1/2}, n^{2/3}))$
- Avec DFS avant et DFS arrière à profondeur bornée

Implémentation et vérification

- Environ 200 lignes de code OCaml
- Preuve de correction et de complexité
- Cf. thèse d'Armaël Guéneau

Pour aller plus loin

Le cours « **Separation Logic Foundations** » entièrement en Coq :

`http://arthur.chargueraud.org/teach/verif`

Merci !