Separation Logic Foundations

Arthur Charguéraud

From Hoare Logic to Separation Logic

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

Hoare triple :

H and Q describe the whole of the memory state

Separation Logic triple :

H and Q describe only a piece of the memory state,
a piece that includes the resources necessary to the execution of t

The frame rule

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

Separation Logic in practice

1. **Automated proofs**
   - The user provides code
   - The tool locates potential bugs
2. **Semi-automated proofs**
   - The user provides code, specifications, and invariants
   - The tool leverages automated solvers (SMT)
3. **Interactive proofs**
   - The user provides code, specifications, and proof scripts
   - Proofs are developed using a proof assistant (e.g., Coq)

Separation Logic in proof assistants

Project	Tool	Language	Target
Ynot	Coq	ML	Data structures
Sel4	Isabelle	C, assembly	OS micro-kernel
Flint	Coq	Assembly	OS (operating system)
Bedrock	Coq	Assembly	OS for robots
VST	Coq	C	Concurrent protocols
CakeML	HOL	SML	ML runtime
CFML	Coq	OCaml	Data structures, algorithms
Iris	Coq	Rust	Concurrent libraries

(non-exhaustive list)

Benefits

- Expressiveness (almost) without limits of higher-order logic
- Unified framework to prove both code and mathematical lemmas
- Proof scripts easy to maintain upon code update
- High degree of confidence in the correctess of the tool

Construction

All these tools are constructed following the same schema

1. Formalization of the syntax and semantics of the source language
2. Definition of Separation Logic predicates as higher-order logic ones
3. Definition of triples, and statements and proofs of reasoning rules
4. Infrastructure (lemmas, tactics, notation) enabling concise proofs

Goal of this course: explain this construction for

- a minimalistic imperative programming language,
- the simplest variant of Separation Logic.

Step 1

Syntax and semantics of the source language

Grammar of the source language

```
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
  ..
```

Parsing of concrete programs

```
let rec mlength p = (* length of a C-style mutable list *)
  if p == null
    then 0
    else 1 + mlength p.tail
```

Corresponding Coq definition

```
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"))))).
```

Coq definition using notation and coercions

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

Semantics of the source language, in big-step style

```
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)
  ...
```

Step 2

Predicates and entailments in Separation Logic

Separation Logic predicates

```
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.
```

Order relation on predicates

Entailment

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

An order relation

```
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.
```

Fundamental properties of the star

```
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 of postconditions

A precondition describes an input state

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

A postcondition describes an output state and an output value

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

Generalization of star and of entailment

```
                                              (* 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.
```

Step 3

Definition of triples, statement and proof of reasoning rules

Definition of triples, in total correctness

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

Hoare triple

```
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'.
```

Separation Logic triple

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

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

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

Structural rules of Separation Logic

Main rules

```
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').
```

Extraction rules

```
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.
```

Reasoning rules, e.g., for sequences

Hoare Logic rule

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

Separation Logic rule

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

Rules specifying primitive operations

```
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 _ ⇒ [])
```

Demo

Example of a verification proof by hand in Separation Logic

Step 4

Infrastructure for more concise proof scripts

Characteristic formula generator

Weakest precondition calculus

- targets Hoare logic
- targets code annotated with invariants

Characteristic formulae

- targets Separation Logic, including the frame rule
- targets code with outout any annotation

Technical challenges for the generator

- definition has to be structurally recursive, and compute within Coq
- output of the generator should be human readable

Overview of the ingredients of the generator

1. **To cope with the lack of annotations**
   `wpgen` leverages the notion of semantic `wp`.
2. **To accomodate for the frame rule**
   `wpgen` integrates a predicate called `mkstruct`.
3. **To be structurally recursive**
   `wpgen` performs substitutions in a lazy manner.
4. **To improve readability of the output**
   `wpgen` introduces intermediate definitions and notation.

Semantic weakest precondition

Characterisation 1

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

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

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

Characterisation 2

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

Characterisation 3

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

Separation Logic in wp-style

Reasoning rules for terms

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

Only two structural rules

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

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

Definition of the generator (1/5)

Weakest precondition calculus for unannotated terms

```
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.
```

Definition of the generator (2/5)

Reformulation with a context, to make termination obvious

```
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.
```

Definition of the generator (3/5)

Swapping `match` and `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.
```

Definition of the generator (4/5)

Introduction of auxiliary definitions

```
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.
```

Example of an auxiliary definition

`wpgen_let F F'` is a definition for `fun Q ⇒ F (fun v ⇒ F' v Q)`.

`Let v := F1 in F2` is a notation for `wpgen_let F1 (fun v ⇒ F2)`.

`wpgen (trm_let x t1 t2)` displays in the form `Let x := F1 in F2`.

Definition of the generator (5/5)

Integration of the frame rule

```
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).
```

Definition de mkstruct

Required properties

```
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).
```

Realization

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

Soundness of the characteristic formulae generator

Soundness theorem

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

Recall the equivalence

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

How to invoke the generator

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

Zoom on a key ingredient for implementing the infrastructure

The magic wand

Intuition

```
Definition hwand (H1 H2:hprop) : hprop := ... (* noté H1 -★ H2 *)
Parameter hwand_elim : H1 ★ (H1 -★ H2) ⊢ H2.
```

One possible definition

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

Generalization to postconditions: $Q1 \mathrel{\dot{-\!\!\star}} Q2$

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

Structural rules revisited with the magic wand

Without the wand

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

With the wand

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

  triple t H Q.
```

With the wand, in wp-style

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

Demo

Example verification proof using `wpgen` and tactics

Conclusions

Summary of the construction

1. Syntax with val and trm, and semantics with eval
2. Predicates `[]`, `[P]` and $\texttt{p} \mapsto \texttt{v}$ and $\texttt{H1} * \texttt{H2}$ and $\exists \texttt{x}, \texttt{H}$, with $\vdash$ and $\underset{\cdot}{\vdash}$
3. Triples `hoare` and `triple`, statements and proofs of rules
4. Infrastructure: `wp`, `wpgen`, $\multimap\!\star$, `Enc`, x-tactics

The CFML tool

Language extensions

- `for`-loops and `while`-loops
- mutual recursion
- records and arrays
- algebraic data types and pattern matching
- simple functors

Logical extensions

- Affine predicates, to reflect for the action of the garbage-collect
- Time credits, to reason about amortized asymptotic complexity

Example of a CFML proof

Problem

- Incremental cycle detection

Algorithm

- by Bender, Fineman, Gilbert, and Tarjan (2016)
- of complexity $O(m \cdot \min(m^{1/2}, n^{2/3}))$
- involving forward DFS, and backward DFS at bounded depth

Implementation and verification

- About 200 lines of dense OCaml code
- Proof of correctness and asymptotic complexity
- See Armaël Guéneau's PhD thesis for details

For additional information

The course **Separation Logic Foundations** , entirely in Coq:

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