Shallow embedding of a logic in Coq

Jérôme Vouillon

Universite Paris Diderot - Paris 7, CNRS

Motivation

Hoare-style assertions

$$\{P\}\ C\ \{Q\}$$

How to specify the language of assertions?

Reasoning about assertions

- fairly complete set of lemmas (inference rules)
- easy to use lemmas

Formalization

- Robust
- Compact

Motivation

Hoare-style assertions

$$\{P\}\ C\ \{Q\}$$

How to specify the language of assertions?

Reasoning about assertions

- fairly complete set of lemmas (inference rules)
- easy to use lemmas

Formalization

- Robust
- Compact

A General Framework for Certifying Garbage Collectors and Their Mutators (McCreight, Shao, Lin, Li) : 1.8 MiB

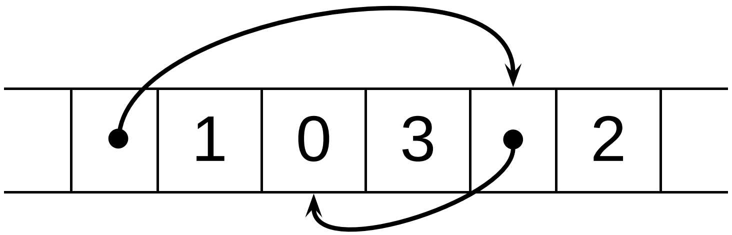
Summary

- Separation logic
- Using the logic
- Formalization of the logic
- Applications

Separation Logic

Separation Logic

Reasoning about program heaps

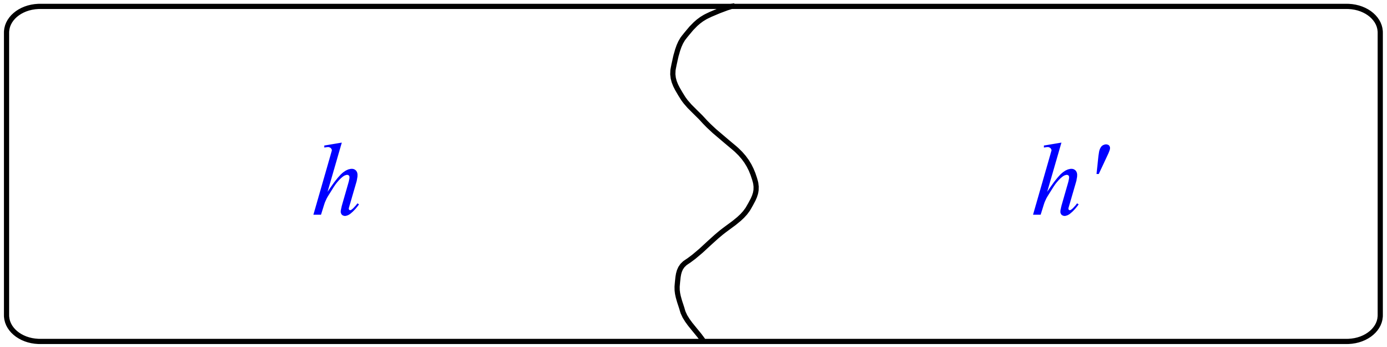


Describe heap portions ("heaplets")

- `emp` : "the heaplet is empty"
- $x \mapsto y$: "the heaplet has exactly one cell x, holding y"
- $A * B$: "the heaplet can be divided so that A is true of one partition and B of the other"

Separation Logic

Good compositional properties

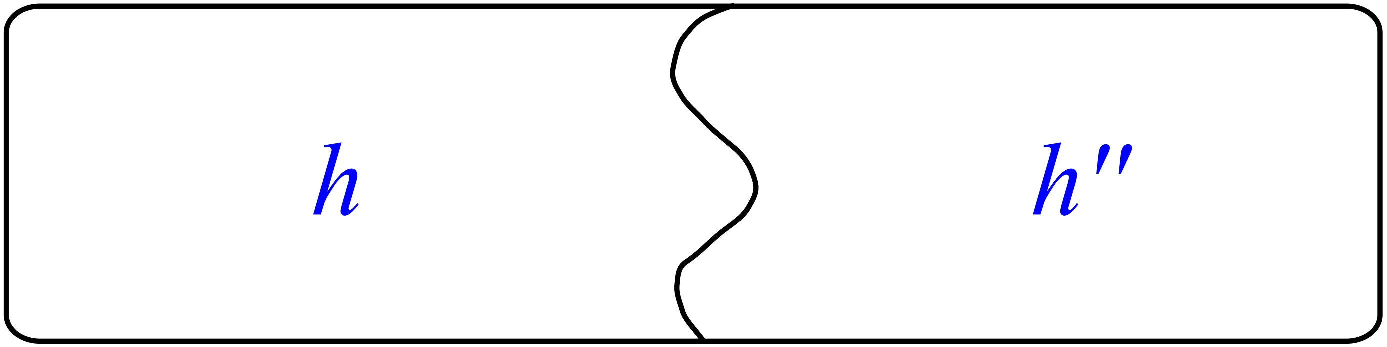


$h \Vdash A$ $\qquad$ $h' \Vdash B$

$$h + h' \Vdash A * B$$

Separation Logic

Good compositional properties



$h \Vdash A$ $\qquad$ $h'' \Vdash C$

$$h + h'' \Vdash A * C$$

Applications

Typing assembly code

```
Lemma mov_type :
  forall (r1 r2 : register) (v : word) (c : instr_seq) (p : prop),
  instructions (seq (mov r2 r1) c) ** (p ** r1 ↦ v ** r2 ↦_) ⊢
  next (instructions c ** (p ** r1 ↦ v ** r2 ↦ v)).
```

Typing higher order programs with effects (Ynot)

```
Definition snew (A : Type) (v : A) (p : prop) :
   {{ p }} y {{ p ** ∃l, {| y = Val l |} ** l ↦ dyn _ v }}.
```

Using a logic

Sequents

General sequents $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \qquad\qquad \frac{\Gamma \vdash A}{!\Gamma \vdash !A}$$

Sequents

General sequents $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \qquad\qquad \frac{\Gamma \vdash A}{!\Gamma \vdash !A}$$

Inconvenient to use

Sequents

General sequents $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \qquad\qquad \frac{\Gamma \vdash A}{!\Gamma \vdash !A}$$

Inconvenient to use

Binary sequents $A \vdash B$

- Simple rules
- Can use Coq rewriting tactic.

Some rules of the logic

Sample rules

$$A \vdash A \qquad \frac{A \vdash B \qquad B \vdash C}{A \vdash C} \qquad A * B \vdash B * A$$

$$A * (B * C) \vdash (A * B) * C \qquad \frac{A \vdash B \qquad C \vdash D}{A * C \vdash B * D}$$

$$A * \mathtt{emp} \vdash A \qquad A \vdash A * \mathtt{emp}$$

Proof technique

Mostly by rewriting

Goal: $C \vdash D$

If $A \vdash B$, rewrite A into B in C.

Additional tactics

- normalize the hypotheses
- reorder the hypotheses

Shallow embedding

Deep embedding

```
Inductive prop : Type :=
  emp : prop
  conj : prop -> prop -> prop
  ...
```

Shallow embedding

```
Definition prop := heaplet -> Prop.
Definition emp : prop := fun h => is_empty h.
```

Higher-order abstract syntax

```
Definition forall (A : Type) (f : A -> prop) :=
  fun h => forall x, f x h.
Notation "∀ x , p" := (Forall (fun x => p)).
```

Formalization

State and permissions

Machine state σ

```
Parameter state : Type.
```

Permissions π (here, a set of locations)

```
Parameter location : Type.
Definition domain := location -> Prop.
```

Permissions can be concatenated: $\pi \approx \pi_1 \uplus \pi_2$

Worlds

Worlds w

```
Record world : Type :=
  make_world { w_state : state;
               w_perms : domain }
```

Equivalence between worlds: $w \approx w'$

Worlds can be concatenated: $w \approx w_1 \uplus w_2$

Semantics: $w \Vdash A$ iff $w \in A$

```
Definition prop := world -> Prop.
```

Logic

Define

$$A \vdash B$$

$$w \Vdash A_1 * A_2$$

$$w \Vdash \mathtt{emp}$$

```
Definition sequent p q := forall w, p w -> q w.
Definition conj p1 p2 :=
  fun w =>
    exists w1, exists w2,
    concat w w1 w2 /\ p1 w1 /\ p2 w2.
Definition emp w := forall l, ~ w_perms w l.
```

Logic rules

Derived lemmas

$$A * B \vdash B * A \qquad A * (B * C) \vdash (A * B) * C \qquad \frac{A \vdash B \qquad C \vdash D}{A * C \vdash B * D}$$

$$\frac{A * B \vdash C}{A \vdash B \mathrel{-\!\!*} C} \qquad \frac{A \vdash B \mathrel{-\!\!*} C}{A * B \vdash C} \qquad A \vdash A * \mathtt{emp}$$

Logic rules

Derived lemmas

$$A * B \vdash B * A \qquad A * (B * C) \vdash (A * B) * C \qquad \frac{A \vdash B \qquad C \vdash D}{A * C \vdash B * D}$$

$$\frac{A * B \vdash C}{A \vdash B \mathrel{-\!\!*} C} \qquad \frac{A \vdash B \mathrel{-\!\!*} C}{A * B \vdash C} \qquad A \vdash A * \texttt{emp} \qquad A * \texttt{emp} \vdash A$$

Extensionality

$$A * \texttt{emp} \vdash A \quad ?$$

$$\left.\begin{array}{l} w \approx w_1 \uplus w_2 \\ w_1 \Vdash A \\ w_2 \Vdash \texttt{emp} \end{array}\right\} w \Vdash A \quad ?$$

Need extensionality

If $w \approx w'$ and $w \Vdash A$, then $w' \Vdash A$

Common solution

Heaplet

A world is a partial heap h
If $h \approx h'$, then $h = h'$.

Possibly infinite maps (Ni, Shao)

```
Definition heap := location -> option word.
Axiom ext_eq :
  forall A B (f g : A -> B),
  (forall x, f x = g x) -> f = g.
```

Finite maps (Marti, Affeldt, Yonezawa)

Unique representation using sorted lists

Variant

Canonical elements (McCreight, Shao, Lin, Li)

$h \Vdash A$ iff $\mathrm{canon}(h) \in A$

If $h \approx h'$, then $\mathrm{canon}(h) = \mathrm{canon}(h')$.

Variant

Canonical elements (McCreight, Shao, Lin, Li)

$h \Vdash A$ iff $\text{canon}(h) \in A$

If $h \approx h'$, then $\text{canon}(h) = \text{canon}(h')$.

Disavantage: cannot use arbitrary worlds

A first generic solution

Define $w \Vdash A$ as:

for all w', $w \approx w'$ implies $w' \in A$

Then, $w \Vdash A$ and $w \approx w'$ implies $w' \Vdash A$.

Last iteration

Kripke semantics

Accessibility relation R between worlds.
Propositions are closed under R.

```
Definition R_persistent p :=
   forall w w', p w -> R w w' -> p w'.

Record prop : Type :=
  make_prop { prop_def :> world -> Prop;
              prop_pers : R_persistent prop_def }.
```

(Also considered by Appel and Blazy)

Definitions

```
Definition Conj_def (p1 p2 : prop) :=
  fun w =>
    exists w1, exists w2,
    concat w w1 w2 /\ p1 w1 /\ p2 w2.

Lemma conj_persistent :
  forall p1 p2, R_persistent (Conj_def p1 p2).
...

Definition Conj p1 p2 :=
   make_prop (conj p1 p2) (conj_persistent p1 p2).
```

Indexing

Indexing

Use pairs (σ, n) of state σ and integer $n \geq 0$

n decreases strictly at each step

All execution traces are finite

$\Rightarrow$ induction principle

Index closure

If $(\sigma, n, \pi) \Vdash A$ and $n' \leq n$, then $(\sigma, n', \pi) \Vdash A$

$\Rightarrow$ define R accordingly

Using modules

Modular design

Use functors

- for enriching worlds (with permissions)

```
Module Heap_world := Heap.World World.
```

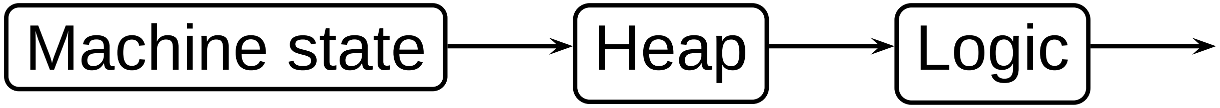
- for defining the core logic

```
Module Logic := Linear.Logic World.
```

- for extending the logic

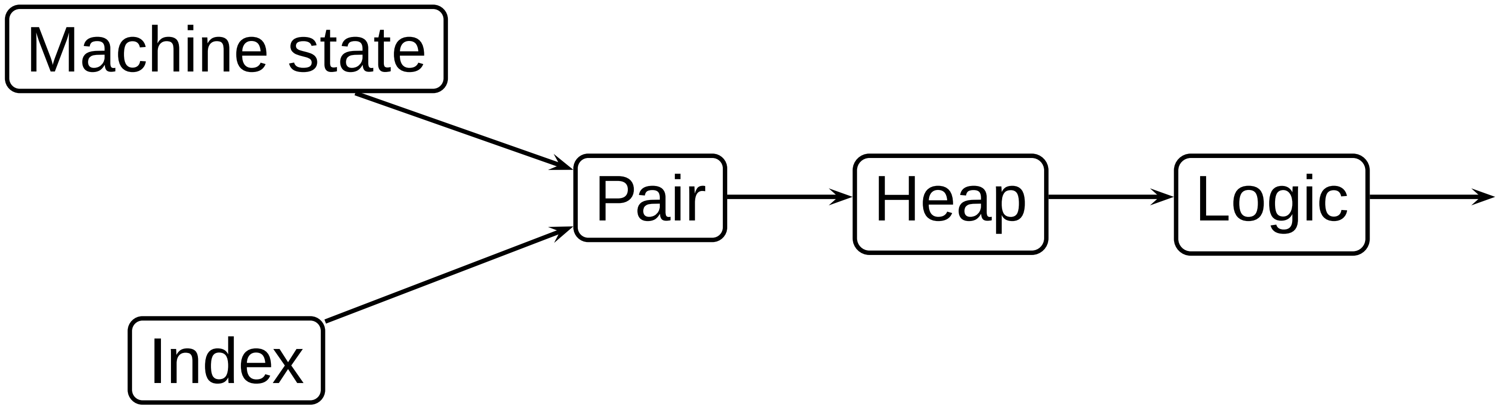
```
Module LaterMod := Logic.Later Indexed_world.
```

Simple instantiation




```
Module Heap_world := Heap.World Machine_world.
Module Logic := Linear.Logic Heap_world.
```

Complex instantiation




```
Module I_world := Heap.Pair Machine_world Indexed_world.
Module I_step := I_world.W.Step Machine_world.W Indexed_world.
Module I_later := I_world.W.Later Indexed_world.

Module HI_world := Heap.World I_world.
Module HI_step := HI_world.Step I_step.
Module HI_later := HI_world.Later I_later.

Module MD := Modal.Logic HI_world.
Module MS := MD.Step HI_step.
Module ML := MD.Later HI_later.
```

Limit of Modules

Module that depends on

```
W
Logic W
```

First possibility

```
Module F (W : WORLD).
  Module L := Logic W.
   ...
End F.
```

Second possibility

```
Module F (W : WORLD) (L : LOGIC with ...).
   ...
End F.
```

Formalization: conclusion

Assessement

Not too costly

About 20% overhead in proof size
(due to Kriepke semantics and modular design)

Robust

No fancy heaplet representation

Applications

Ynot: Reasoning with the Awkward Squad,
Nanevski, Morrisett, Shinnar, Govereau, Birkedal.

Definition of the logic: around 75 lines

Proof of some primitives

```
Definition snew (A : Type) (v : A) p :
  {{ p }} y {{ p ** ∃l, {| y = Val l |} ** l ↦ dyn _ v }}.
Definition sfree (l : loc) p : {{ p ** l ↦_ }} y {{ p }}.
```

(around 30 lines for each proofs)

Append function

Certified Assembly Programming with Embedded Code Pointers, Zhaozhong Ni and Zhong Shao

Destructive list append in CPS

```
append:  bgti r0, 0, else
         ld r31, r2(0)
         ld r0, r2(1)
         free r2, 2
         jmp r31
else:    alloc r3, 2
         st r3(0), r0
         st r3(1), r2
         ld r0, r0(1)
         alloc r2, 2
         st r2(1), r3
         movi r3, k
         st r2(0), r3
         jd append
k:       ld r2, r0(0)
         ld r3, r0(1)
         free r0, 2
         st r2(1), r1
         mov r1, r2
         ld r31, r3(0)
         ld r0, r3(1)
         free r3, 2
         jmp r31
```

Append function

Proof size in bytes:

	Ni and Shao	This work
Specification	100470	77093
“list-append” example	83296	9704

Room for improvement

Important points

- Binary sequents $p \vdash q$
- Kripke semantics
- Modularity