Shallow embedding of a logic in Coq

Jérôme Vouillon

Université Paris Diderot - Paris 7, CNRS
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
Reasoning about program heaps

Describe heap portions ("heaplets")

- \texttt{emp} : "the heaplet is empty"
- \( x \mapsto y \) : "the heaplet has exactly one cell \( x \), holding \( y \)"
- \( A \ast B \) : "the heaplet can be divided so that \( A \) is true of one partition and \( B \) of the other"
Good compositional properties

\[ h \vdash A \quad h' \vdash B \]

\[ h + h' \vdash A \ast B \]
Separation Logic

Good compositional properties

\[ h \vdash A \quad h'' \vdash C \quad h + h'' \vdash A \ast C \]
Typing assembly code

Lemma mov_type :
for all (r1 r2 : register) (v : word) (c : instr_seq) (p : prop),
instructions (seq (mov r2 r1) c) ** (p ** r1 \mapsto v ** r2 \mapsto \_ ) \vdash
next (instructions c ** (p ** r1 \mapsto v ** r2 \mapsto v)).

Typing higher order programs with effects (Ynot)

Definition snew (A : Type) (v : A) (p : prop) :
{} p {} y {} p ** \exists l, \{ y = Val l \} ** l \mapsto dyn _ v {}).
Using a logic
General sequents $A_1, ..., A_n \vdash B_1, ..., B_m$

\[
\frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \quad \frac{\Gamma \vdash A}{!\Gamma \vdash !A}
\]
Sequents

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

$$
\frac{
\Gamma \vdash A, \Delta \hspace{1cm} \Gamma', A \vdash \Delta'
}{
\Gamma, \Gamma' \vdash \Delta, \Delta'
}
\hspace{2cm}
\frac{
\Gamma \vdash A
}{
!\Gamma \vdash !A
}

Inconvenient to use
General sequents $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$

\[
\frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \quad \frac{\Gamma \vdash A}{!\Gamma \vdash !A}
\]

*Inconvenient to use*

Binary sequents $A \vdash B$

- Simple rules
- Can use Coq rewriting tactic.
Some rules of the logic

Sample rules

\[
\begin{align*}
A & \vdash A \\
A \vdash B & \quad B \vdash C \\
\frac{}{A \vdash C} & \\
A \ast B & \vdash B \ast A
\end{align*}
\]

\[
\begin{align*}
A \ast (B \ast C) & \vdash (A \ast B) \ast C \\
A \ast \text{emp} & \vdash A \\
A & \vdash A \ast \text{emp}
\end{align*}
\]
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

\textbf{Inductive} \texttt{prop} : Type :=
\begin{align*}
\texttt{emp} : \texttt{prop} \\
\texttt{conj} : \texttt{prop} \to \texttt{prop} \to \texttt{prop} \\
\ldots
\end{align*}

Shallow embedding

\textbf{Definition \texttt{prop}} := \texttt{heaplet} \to \texttt{Prop}.
\textbf{Definition \texttt{emp}} : \texttt{prop} := \texttt{fun h} \Rightarrow \texttt{is\_empty h}.

Higher-order abstract syntax

\textbf{Definition \texttt{forall} (A : Type) (f : A \to \texttt{prop}) :=}
\begin{align*}
\texttt{fun h} \Rightarrow \texttt{forall x, f x h}.
\end{align*}
\textbf{Notation} \texttt{"\(\forall\) x, p" := (Forall (fun x => p)).}
Formalization
State and permissions

Machine state $\sigma$

Parameter $\text{state} : \text{Type}$.

Permissions $\pi$ (here, a set of locations)

Parameter $\text{location} : \text{Type}$.
Definition $\text{domain} := \text{location} \rightarrow \text{Prop}$.

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

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

Equivalence between worlds: $w \approx w'$

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

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

Definition prop := world -> Prop.
Define

\[ A \vdash B \]

\[ w \vdash A_1 \ast A_2 \]

\[ w \vdash \text{emp} \]

Definition \texttt{sequent} \( p \; q := \forall w, \; p \; w \rightarrow q \; w \).

Definition \texttt{conj} \( p1 \; p2 := \)

\[
\text{fun} \; w \Rightarrow \\
\exists w1, \exists w2, \\
\text{concat} \; w \; w1 \; w2 \; \text{/\} \; p1 \; w1 \; \text{/\} \; p2 \; w2.
\]

Definition \texttt{emp} \( w := \forall l, \; \sim w_{\text{perms}} \; w \; l \).
Logic rules

Derived lemmas

\[ A \ast B \vdash B \ast A \quad A \ast (B \ast C) \vdash (A \ast B) \ast C \quad \frac{A \vdash B \quad C \vdash D}{A \ast C \vdash B \ast D} \]

\[ A \ast B \vdash C \quad A \vdash B \rightarrow C \quad \frac{A \ast C \vdash B \ast D}{A \vdash A \ast \text{emp}} \]
Derived lemmas

\[
\begin{align*}
A \ast B & \vdash B \ast A & A \ast (B \ast C) & \vdash (A \ast B) \ast C & A \vdash B \quad C \vdash D \\
A \ast C & \vdash B \ast D
\end{align*}
\]

\[
\begin{align*}
A \ast B & \vdash C & A \vdash B \ast C & A \vdash A \ast \text{emp} & A \ast \text{emp} & \vdash A \\
A \vdash B \ast C & A \ast B \vdash C
\end{align*}
\]
Extensionality

\[ A \ast \text{emp} \vdash A \ ? \]

\[
\begin{cases}
    w \approx w_1 \uplus w_2 \\
    w_1 \vdash A \\
    w_2 \vdash \text{emp}
\end{cases}
\]

\[ w \vdash A \ ? \]

Need extensionality

If \( w \approx w' \) and \( w \vdash A \), then \( w' \vdash A \)
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
Canonical elements (McCreight, Shao, Lin, Li)

\[ h \models A \iff \text{canon}(h) \in A \]

If \( h \approx h' \), then \( \text{canon}(h) = \text{canon}(h') \).
**Canonical elements** (McCreight, Shao, Lin, Li)

\[ h \models A \text{ iff } \text{canon}(h) \subseteq A \]

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

*Disadvantage: cannot use arbitrary worlds*
A first generic solution

Define $w \models A$ as:

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

Then, $w \models A$ and $w \approx w'$ implies $w' \models A$. 
Kripke semantics

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

Definition $R_{\text{persistent}} p :=$
  \[
  \forall w w', p w \rightarrow R w w' \rightarrow p w'.
  \]

Record prop : Type :=
  make_prop { prop_def :> world -> Prop;
               prop_pers : $R_{\text{persistent}}$ prop_def }.

(Also considered by Appel and Blazy)
**Definition**  \( \text{Conj\_def} \) (\( p_1 \ p_2 : \text{prop} \)) :=
\[
  \text{fun } w => \\
  \quad \text{exists } w_1, \text{exists } w_2, \\
  \quad \text{concat } w \ w_1 \ w_2 \ /\!\!/ \ p_1 \ w_1 \ /\!\!/ \ p_2 \ w_2.
\]

**Lemma**  \( \text{conj\_persistent} \):
\[
  \forall p_1 \ p_2, \text{R\_persistent} \ (\text{Conj\_def} \ p_1 \ p_2).
\]
...

**Definition**  \( \text{Conj} \) \( p_1 \ p_2 :=\)
\[
  \text{make\_prop} \ (\text{conj} \ p_1 \ p_2) \ (\text{conj\_persistent} \ p_1 \ p_2).
\]
Indexing
Use pairs \((\sigma, n)\) of state \(\sigma\) and integer \(n \geq 0\)

\(n\) decreases strictly at each step

All execution traces are finite

\[\Rightarrow\] induction principle

Index closure

If \((\sigma, n, \pi) \not\models A\) and \(n' \leq n\), then \((\sigma, n', \pi) \not\models A\)

\[\Rightarrow\] define \(R\) accordingly
Using modules
Use functors

- for enriching worlds (with permissions)
  

- for defining the core logic
  
  Module Logic := Linear.Logic World.

- for extending the logic
  
  Module LaterMod := Logic.Later Indexed_world.
Simple instantiation

Module Logic := Linear.Logic Heap_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

\[ W \]

\text{Logic \( W \)}

First possibility

Module \text{F} (W : WORLD).

Module \text{L} := \text{Logic \( W \)}.

\ldots

End \text{F}.

Second possibility

Module \text{F} (W : WORLD) (L : \text{LOGIC with \ldots}).

\ldots

End \text{F}.
Formalization: conclusion
Not too costly

About 20% overhead in proof size
(due to Kripke 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 :

```plaintext
{{ p }} y {{ p ** ∃ l, { y = Val l } ** l ↦→ dyn _ v }).
```

Definition `sfree` (l : loc) p :

```plaintext
{{ 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:

<table>
<thead>
<tr>
<th></th>
<th>Ni and Shao</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td>100470</td>
<td>77093</td>
</tr>
<tr>
<td>“list-append” example</td>
<td>83296</td>
<td>9704</td>
</tr>
</tbody>
</table>

*Room for improvement*
Important points

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