Reasoning on Imperative Programs Through Characteristic Formulae

Arthur Charguéraud

INRIA

Dagstuhl Seminar

Dagstuhl, 2010/08/31

Motivation

– **Verification of Caml programs**

Call-by-value, sequential, deterministic

First class functions, polymorphic recursion,

Data types, pattern matching,

Mutable references, null pointers, strong updates

– **Through interactive Coq proofs**

Total correctness, higher-order logic specifications

Working with heap predicates and the frame rule

Describing Caml programs in Coq

generator

Caml program without annotation

Characteristic formula (CF) expressed in Coq

- **Not a deep embedding**
  → Formulae do not refer to Caml syntax
- **Not a shallow embedding**
  → Caml functions not represented as Coq functions
- **Related to Berger, Honda and Yoshida’s TCAPs**
  → Sound and complete description of programs

Interpretation of CF

Hoare triple: $\{H\}\ t\ \{Q\}$

where $(H : \text{Hprop})$ and $(Q : A \to \text{Hprop})$

where "Hprop" abbreviates "Heap $\to$ Prop"

Total correctness interpretation: in a piece of heap satisfying H, the term t terminates and returns a value and a heap satisfying Q.

Characteristic formulae: $\|t\|\ H\ Q$

where $\|t\| : \text{Hprop} \to (A \to \text{Hprop}) \to \text{Prop}$

Difference: $\{H\}\ t\ \{Q\}$ is a three place relation that refers to the syntax of t, whereas $\|t\|$ is a predicate expressed directly in terms of basic higher-order logic connectives (e.g., $\wedge$, $\Rightarrow$, $\forall$, $\exists$).

Implementation and examples

Implementation of CFML:

– CF generator: 3.000 lines of Caml (+ type-checker)

– Coq lemmas, tactics and notation: 4.000 lines of Coq

½ of Okasaki's purely functional data structures:

– 825 lines of Caml verified through 5.000 lines of Coq

– local ratio "proofs/(code+spec)" between 1.0 and 2.0

– the entire set of files compiles in about 2 minutes

Recently extended to imperative programs:

– In-place list reversal

– Copy of a mutable tree

– Append and CPS-append functions

– Higher-order iterators: iter, map, fold (on-going)

– Landin's knot

CF for a let-binding

Hoare logic:

$$\frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\,x\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}}$$

Characteristic formula:

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

Introduction of notation:

$$(\mathbf{let}\ x = \mathcal{F}_1\ \mathbf{in}\ \mathcal{F}_2) \quad \equiv$$
$$\lambda H.\lambda Q.\ \ \exists Q'.\ \mathcal{F}_1\,H\,Q' \ \ \wedge\ \ \forall x.\ \mathcal{F}_2\,(Q'\,x)\,Q$$

Characteristic formula generator:

$$[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!] \quad \equiv \quad (\mathbf{let}\ x = [\![t_1]\!]\ \mathbf{in}\ [\![t_2]\!])$$

Interest of characteristic formulae

$$[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2]\!] \quad \equiv \quad (\mathbf{let}\ x = [\![t_1]\!]\ \mathbf{in}\ [\![t_2]\!])$$

1) CF generation is really straightforward

$\rightarrow$ all you need is to is type-check the input program

2) CF generation is entirely compositional

$\rightarrow$ reasoning on programs is thus also compositional

3) Proof obligations read like source code

$\rightarrow$ they take the form ‖t‖ H Q, so it displays as the source code followed by a pre- and a post-condition

Integration of the frame rule

Hoare logic:

$$\frac{\{H_1\}\ t\ \{Q_1\}}{\{H_1 * H_2\}\ t\ \{Q_1 \star H_2\}}$$

where $Q_1 \star H_2$ is defined as "$\lambda v.\,(Q_1\, v) * H_2$"

Updated definition:

$$\begin{aligned}(\textbf{let}\ x = \mathcal{F}_1\ \textbf{in}\ \mathcal{F}_2) \quad &\equiv \\ &\mathsf{frame}\,(\lambda H.\,\lambda Q.\ \ \exists Q'.\ \mathcal{F}_1\, H\, Q' \ \wedge\ \forall x.\ \mathcal{F}_2\,(Q'\, x)\, Q)\end{aligned}$$

Predicate presentation:

$$\mathsf{frame}\,\mathcal{F} \quad \equiv \quad \lambda H\, Q.\ \exists H_1\, H_2\, Q_1.\ \begin{cases} H = H_1 * H_2 \\ \mathcal{F}\, H_1\, Q_1 \\ Q = Q_1 \star H_2 \end{cases}$$

The predicate "local" generalizes "frame". It supports the rules of consequence and of garbage collection, as well as extraction of quantifiers and propositions.

Translation of types

The Caml type T is reflected as the Coq type ⟨T⟩

$$
\begin{array}{lcl}
\langle \mathsf{int} \rangle & \equiv & \mathsf{Int} \\
\langle T_1 \times T_2 \rangle & \equiv & \langle T_1 \rangle \times \langle T_2 \rangle \\
\langle T_1 + T_2 \rangle & \equiv & \langle T_1 \rangle + \langle T_2 \rangle \\
\langle T_1 \rightarrow T_2 \rangle & \equiv & \mathsf{Func} \\
\langle \mathsf{ref}\, T \rangle & \equiv & \mathsf{Loc}
\end{array}
$$

Model : a Coq value of type Func corresponds to the source code of a well-typed Caml function

A heap is a map from location to dependent pairs made of a type and a value of that type.

Breaking the circularity

Models for higher-order stores are tricky

$$\text{Type} \approx \text{World} \to \text{Pred(Values)}$$
$$\text{World} \approx \text{Loc} \to \text{Type}$$

The circularity is here somehow avoided:

– references are represented by their memory location

– functions are represented by their source code

The equations become:

$$\text{Rtype} \approx \langle\, \text{CamlType} \,\rangle \qquad (\text{Rtype} \subset \text{CoqType})$$
$$\text{Heap} \approx \text{Loc} \to (\, \Sigma A{:}\text{Rtype}.\ A)$$

Values of type Loc or Func may be stored in the heap since they do not refer to the type "Heap" in any way.

Specification of functions

The abstract predicate **AppReturns f x H Q** asserts that the application of **f** to the value **v** admits **H** as pre-condition and **Q** as post-condition.

Example: for any location r and integer n,

```
AppReturns incr r (r ~~> n) (fun _ => r ~~> n+1)
```

Type of AppReturns:

$\forall$A. $\forall$B. Func $\rightarrow$ A $\rightarrow$ Hprop $\rightarrow$ (B $\rightarrow$ Hprop) $\rightarrow$ Prop

Characteristic formulae for applications:

$$[\![f\ v]\!] \quad \equiv \quad \mathsf{AppReturns}\, f\, v$$

CF for function definitions

Instances of the predicate AppReturns:

– have to be provided for reasoning on applications,

– are provided by the CF of a function definition.

Characteristic formulae for (recursive) functions:

$$\begin{aligned} [\![\mathsf{let}\ f\ x = t\ \mathsf{in}\ t']\!] \quad &\equiv \\ &\lambda H Q.\, \forall f.\ (\forall x\, H'\, Q'.\ [\![t]\!]\ H'\ Q' \ \Rightarrow\ \mathsf{AppReturns}\ f\ x\ H'\ Q') \ \Rightarrow\ [\![t']\!]\ H\ Q \end{aligned}$$

Remark: no explicit treatment of recursivity; recursive functions are proved correct by induction.

Characteristic formula generation

$H_1 \rhd H_2$ is entailment on heap predicates

$$[\![v]\!] \equiv$$
$$\text{local}\,(\lambda H Q.\ H \rhd Q\, v)$$

$$[\![f\ v]\!] \equiv$$
$$\text{local}\,(\lambda H Q.\ \text{AppReturns}\ f\ v\ H\ Q)$$

$$[\![\text{crash}]\!] \equiv$$
$$\text{local}\,(\lambda H Q.\ \text{False})$$

$$[\![\text{if}\ v\ \text{then}\ t_1\ \text{else}\ t_2]\!] \equiv$$
$$\text{local}\,(\lambda H Q.\ (v = \text{true} \Rightarrow [\![t_1]\!]\, H\, Q) \wedge (v = \text{false} \Rightarrow [\![t_2]\!]\, H\, Q))$$

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

$$[\![\text{let}\ f\ x = t_1\ \text{in}\ t_2]\!] \equiv$$
$$\text{local}\,(\lambda H Q.\ \forall f.\ \mathcal{H} \Rightarrow [\![t_2]\!]\, H\, Q)$$
$$\text{where } \mathcal{H} \text{ is } (\forall x\, H'\, Q'.\ [\![t_1]\!]\, H'\, Q' \Rightarrow \text{AppReturns}\ f\ x\ H'\, Q')$$

Characteristic formula generation

For-loop: invariant of type "int $\rightarrow$ Hprop"

$$[\![\text{for } i = a \text{ to } b \text{ do } t_1]\!] \;\equiv$$

$$\text{local}\,(\lambda HQ.\ \exists I.\ \begin{cases} H \rhd I\,a \\ \forall i \in [a,b].\ [\![t_1]\!]\,(I\,i)\,(\#\,I\,(i+1)) \\ I\,(\max a\,(b+1)) \rhd Q\,tt \end{cases})$$

While-loop: invariants of type "A $\rightarrow$ Hprop" and of type "A $\rightarrow$ bool $\rightarrow$ Hprop", for some type A.

$$[\![\text{while } t_1 \text{ do } t_2]\!] \;\equiv\; \text{local}\,(\lambda HQ.$$

$$\exists A.\ \exists I.\ \exists J.\ \exists(\prec).\ \begin{cases} \text{well-founded}(\prec) \\ \exists X_0.\ H \rhd I\,X_0 \\ \forall X.\ [\![t_1]\!]\,(I\,X)\,(J\,X) \\ \forall X.\ [\![t_2]\!]\,(J\,X\,\text{true})\,(\#\,\exists\!\!\!\exists Y.\ (I\,Y) * [Y \prec X]) \\ \forall X.\ J\,X\,\text{false} \rhd Q\,tt \end{cases}$$

Integration of CF in Coq

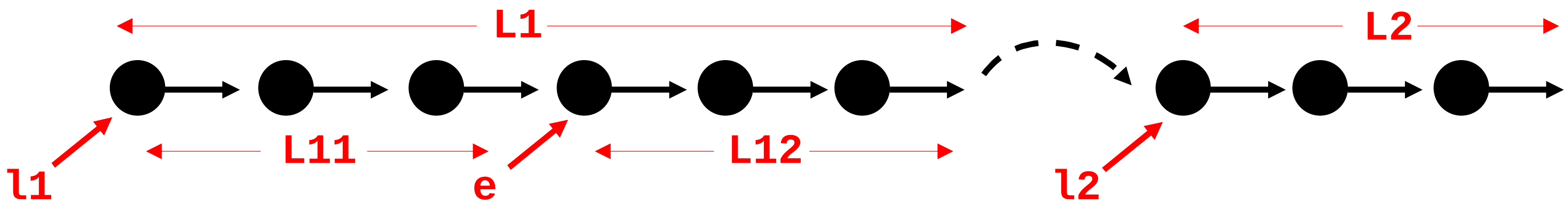
For one single closed term:

```
Definition CF := ‖t‖.      (* generated *)
Lemma specif : CF H Q.     (* specification *)
Proof. ... Qed.            (* verification *)
```

For a set of top-level functions:

```
Axiom f : Func.            (* generated *)
Axiom f_CF :               (* generated *)
   forall x H Q, ‖t‖ H Q -> AppReturns f x H Q
Lemma f_spec : ... .       (* specification *)
Proof.                     (* verification *)
   apply f_CF.
   ...
Qed.
... the rest of the script may refer to f ...
```

Example: destructive append




```
Lemma append_spec : forall A,
  Spec append (l1:loc) (l2:loc) |R>>
    forall (L1 L2:list A),
    R (l1 ~> Mlist L1 \* l2 ~> Mlist L2)
      (fun l' => l' ~> Mlist (L1 ++ L2)).
Proof.
  xcf. intros.
  xif. ...
  ...
  xwhile (fun L12 => Hexists L11 e,
    [L1 = L11 ++ L12] \* (l1 ~> MlistSeg e L11)
    \* (h ~~> e) \* (e ~> Mlist L12) \* [L2 <> nil])
    (@list_sub A).
  ...
Qed.
```

Soundness and completeness

Soundness theorem: $\lfloor V \rfloor$ takes Coq values into Caml

$$\begin{cases} \vdash t : T \\ [\![t]\!]\ H\ Q \\ H\ h_i \\ h_i \perp h_k \end{cases} \Rightarrow \exists V\ h_f\ h_g. \begin{cases} \vdash \lfloor V \rfloor : T \\ t_{/\lfloor h_i \rfloor + \lfloor h_k \rfloor} \Downarrow \lfloor V \rfloor_{/\lfloor h_f \rfloor + \lfloor h_k \rfloor + \lfloor h_g \rfloor} \\ h_f \perp h_k \perp h_g \\ Q \lfloor V \rfloor\ h_f \end{cases}$$

Completeness theorem: (slightly simplified)

$$\begin{cases} \vdash t : T \\ t_{/m} \Downarrow v_{/m'} \end{cases} \Rightarrow [\![t]\!]\ (\mathsf{mgh}\ m)\ (\mathsf{mgp}\ v\ m')$$

Completeness, special case:

$$t_{/\emptyset} \Downarrow n_{/m} \quad \Rightarrow \quad [\![t]\!]\ [\,]\ (\lambda x.\,[x = n])$$

Related work

Comparison with:

– Hoare Logic and Separation Logic

– Honda, Berger and Yoshida's TCAPs

– Shallow embeddings and Ynot

– Deep embeddings

Hoare Logic and Separation Logic

Compared with Hoare Logic:

– No need to apply inductive reasoning rules

– CF not intented for VCG but for interactive proofs

– Total correctness is completely primitive in CF

Compared with Separation Logic:

– All the reasoning takes place in "Hprop", not "Heap"

– Heap predicates are implemented in Coq (standard)

– The frame rule takes the form of a predicate

Honda, Berger and Yoshida's TCAPs

Total characteristic assertion pair:

– (H_w,Q_s) TCAP if H_w is weakest pre, Q_s strongest post

– Algorithm for generating the TCAP of any PCF term

– Idea: to prove {H}t{Q}, it suffices to check

$$H \Rightarrow H_w \qquad \text{and} \qquad H \wedge Q_s \Rightarrow Q$$

$\rightarrow$ TCAP are sound and complete and do not refer to t

Characteristic formulae build on a similar idea.

– TCAPs are expressed in an ad-hoc logic, where values of the logic are PCF values (including functions) and equality is observational equivalence and
$\rightarrow$ I represent functions using the type Func

– Functions are specified with $\{H\}\ f \bullet v = x\ \{H'\}$
$\rightarrow$ Same as the proposition AppReturns f v H (λx'. H')

Shallow embeddings

Representing Caml programs as Coq definitions

$\rightarrow$ CF also benefit from program values being represented as Coq values (e.g. Caml list as Coq list)

$\rightarrow$ CF view functions as object of type Func, and not as Coq functions, which must be total

$\rightarrow$ CF are very flexible w.r.t. the syntax of the source language

Compared with Ynot:

$\rightarrow$ CF need not involve a monad for side-effects

$\rightarrow$ CF separates code from specifications, whereas specifications are imbricated in the code in Ynot

$\rightarrow$ CF offer a simple direct treatment of ghost variables

Deep embeddings

Representing Caml syntax and semantics in Coq

(I have applied this standard technique to pure-Caml)

1) Axiomatized semantics: $t_{/h} \Downarrow v'_{/h'}$

2) Special case of functions: $(f\ v)_{/h} \Downarrow v'_{/h'}$

3) Function specification: **AppEval f v h v' h'**

4) Lift v and v' but not f: **AppEval f V h V' h'**

5) Use heap predicates: **AppReturns f V H Q**

Deep embeddings

CF brought three major improvements:

1) translation of Caml values into Coq becomes implicit

2) the application of reasoning rules becomes implicit,
(in particular no need to compute reduction contexts)

3) reasoning tactics are much simpler to implement

CF can be viewed as an abstract layer built on top of a deep embeeding.

Thanks!

Further information on characteristic formulae for pure programs:

Program Verification Through Characteristic Formulae

Arthur Charguéraud, to appear at ICFP'10

http://arthur.chargueraud.org/research/2010/cfml

More complete presentation and treatment of imperative programs in:

My thesis

Expected in a week or two.

Please come to me if you wish a copy of the current draft.