Characteristic Formulae for the Verification of Imperative Programs

Arthur Charguéraud
Max Planck Institute for Software Systems
Overview

Goal: prove correctness of arbitrarily-complex programs

→ take an existing program written, say, in Caml

→ specify and verify it using Coq

characteristic formulae

used to describe the semantics of the code in the logic
Contribution

Last year's paper (ICFP'10)
→ targets purely-functional programs
→ used CFML to verify half of Okasaki's book

This year's paper (ICFP'11)
→ generalizes the results to imperative programs
  – including local reasoning (frame rule)
  – including higher-order imperative functions
→ extended CFML to verify, e.g., Dijkstra's shortest path, Union-Find, iterators on mutable lists, functions with local state, functions in CPS form, recursion through the store
Features

Extension to the imperative setting preserves the nice properties of characteristic formulae (CF)

→ CF are built compositionally from the source code
→ CF are of linear size w.r.t. the source code
→ CF are displayed in a way that resembles source code
→ CF can be manipulated using solely high-level tactics
→ CF are not just sound but also complete
Structure of the talk

(1) Introduction: what CF are and what they are not
(2) Technical insight: how to construct CF
(3) Example: verification of Dijkstra's algorithm
CF: what they are

Generation of characteristic formulae

\[ C \quad \rightarrow \quad [C] \]

source code without any modification nor annotation  
characteristic formula, expressed in higher-order logic using ∀,∃,∧,⇒,...

CF are sound and complete w.r.t. Hoare logic

\[ \forall H. \forall H'. \quad [C] \ H \ H' \quad \iff \quad \{H\} \ C \ \{H'\} \]

heap predicates (heap → Prop)  
application in higher-order logic  
total correctness  
Hoare triple

→ capturing that, in any heap satisfying \(H\), the execution of the code \(C\) terminates and leaves a heap that satisfies \(H'\).
CF: what they are not

- Not a verification condition generator (VCG)
  → the source code is not annotated with invariants
  → instead, invariants are provided in interactive proofs

- Not a dynamic logic
  → there is no ad-hoc logic construct to embed source code
  → allows to stay in a standard logic and use existing tools

- Not a deep embedding
  → no inductive datatype used to represent code syntax
  → avoids low-level details and issues related to binders

- Not a shallow embedding
  → Caml functions are not represented as Coq functions
  → avoids a mismatch between partial and total functions
Structure of the talk

(1) Introduction: what CF are and what they are not

(2) Technical insight: how to construct CF
   → characteristic formula for sequences
   → function definitions and function calls
   → integration of the frame rule

(3) Example: verification of Dijkstra's algorithm
CF construction for sequence

Hoare logic rule for sequence

\[
\begin{array}{c}
\{ H \} \ C_1 \ \{ H'' \} \\
\{ H'' \} \ C_2 \ \{ H' \}
\end{array}
\]

\[
\{ H \} \ (C_1 ; C_2) \ \{ H' \}
\]

Property of characteristic formulae

\[
\forall H. \forall H'. \ [C_1 ; C_2] \ H \ H' \iff \ \{ H \} \ (C_1 ; C_2) \ \{ H' \}
\]

Characteristic formula for sequence

\[
[C_1 ; C_2] \equiv \lambda H. \lambda H'. \ \exists H''. \ [C_1] \ H \ H'' \ \land \ [C_2] \ H'' \ H'
\]
Notation system for CF

Definition from previous slide

\[
\begin{array}{c}
\left[ C_1 ; C_2 \right] \\
\equiv \\
\lambda H. \lambda H'. \ \exists H''. \ \left[ C_1 \right] H H'' \ \land \ \left[ C_2 \right] H'' H'
\end{array}
\]

Definition of a Coq notation

\[
\begin{array}{c}
\left( F_1 ; F_2 \right) \\
\equiv \\
\lambda H. \lambda H'. \ \exists H''. \ F_1 H H'' \ \land \ F_2 H'' H'
\end{array}
\]

Characteristic formula for sequence, revisited

\[
\left[ C_1 ; C_2 \right] \equiv \left[ C_1 \right] ; ; \left[ C_2 \right]
\]
Generalization

Other language constructs are handled in a similar way

\[
[ C_1 ; C_2 ] \equiv [ C_1 ] ;; [ C_2 ]
\]
\[
[ \text{while} \ C_1 \ \text{do} \ C_2 ] \equiv \text{While} [ C_1 ] \ \text{Do} \ [ C_2 ]
\]

It results that

→ CF are fully compositional
→ CF are easy to generate
→ CF are of linear size

Moreover, thanks to tactics, notation need not be unfolded
Function definitions

For the following definition

\[
\text{function } f(x) \{ C \}
\]

we generate two axioms

Axiom \( f \) : \text{func.}

Axiom \( F \) : \forall x \ H \ H'. [C] H H' \Rightarrow \text{App } f \ x \ H \ H'.

where \text{func} is an abstract type and \text{App} an abstract predicate

\[
\text{func} : \text{Type} \\
\text{App} : \forall A. \text{func} \rightarrow A \rightarrow (\text{Heap} \rightarrow \text{Prop}) \rightarrow (\text{Heap} \rightarrow \text{Prop}) \rightarrow \text{Prop}
\]

Characteristic formula for function calls

\[
[ f(v) ] \equiv \lambda H. \lambda H'. \text{App } f \ v \ H \ H'
\]
Example of a recursive function

We don't need to add anything to support recursion: the specification of a recursive functions can be proved by induction, using Coq's induction principles.

Consider the following recursive function

\[ f(n) \begin{cases} \text{if } n > 0 \text{ then } & \{ x := x + 1; f(n - 1) \} \end{cases} \]

We prove its specification by induction on \( n \)

\[ \forall n. \forall a. \ n \geq 0 \Rightarrow \text{App } f \ n \ (x \leftrightarrow a) (x \leftrightarrow a + n) \]
Frame rule

The frame rule is not syntax directed; how to integrate it?

\[
\frac{\{ H_1 \} \ C \ \{ H'_1 \}}{\{ H_1 \ast H_2 \} \ C \ \{ H'_1 \ast H_2 \}}
\]

Insert a predicate at the head of every node in the CF

\[
[C_1 \ ; \ C_2] \equiv \text{local} (\lambda H. \lambda H'. \ldots)
\]

This \textit{local} predicate supports application of the frame rule at any time and it can be eliminated by framing the empty heap

\[
\text{local } \mathcal{F} \equiv \lambda H. \lambda H'. \exists H_1 H'_1 H_2.
\]

\[
\begin{align*}
\mathcal{F} & \quad H_1 & \quad H'_1 \\
H & = & \ H_1 \ast H_2 \\
H' & = & \ H'_1 \ast H_2
\end{align*}
\]
Structure of the talk

(1) Introduction: what CF are and what they are not

(2) Technical insight: how to construct CF

(3) Example: verification of Dijkstra's algorithm
   → quick overview of the source code
   → specification of the algorithm
   → statement of the invariant
   → main mathematical lemma
   → verification proof script
   → example of a proof obligation
let dijkstra g s e =
  let n = Array.length g in
  let b = Array.make n Infinite in
  let v = Array.make n false in
  let q = Pqueue.create() in
  b.(s) <- Finite 0;
  Pqueue.push (s,0) q;
  while not (Pqueue.is_empty q) do
    let (x,dx) = Pqueue.pop q in
    if not v.(x) then begin
      v.(x) <- true;
      let update (y,w) =
        let dy = dx + w in
        if (match b.(y) with
          | Finite d -> dy < d
          | Infinite -> true)
        then (b.(y) <- Finite dy; Pqueue.push (y,dy) q) in
        List.iter update g.(x);
    end;
  done;
  b.(e)
Specification for Dijkstra's shortest path

**Theorem** dijkstra_spec : \( \forall g \ x \ y \ G, \)
nonnegative_edges G \(\rightarrow\)
x \(\in\) nodes G \(\rightarrow\)
y \(\in\) nodes G \(\rightarrow\)
(App dijkstra g x y)
(g \(\rightarrow\) GraphAdjList G)
(fun d \(\rightarrow\) [d = dist G x y]
\* g \(\rightarrow\) GraphAdjList G)

Remark: the representation predicate GraphAdjList is a user-defined predicate (it is not built in the system)
Main invariant

**Definition** \( hinv \ Q \ B \ V : \ hprop := \)

\[
g \rightarrow \ GraphAdjList \ G
\]

\*[v \rightarrow \ Array \ V]

\*[b \rightarrow \ Array \ B]

\*[q \rightarrow \ Pqueue \ Q]

\*[[inv \ Q \ B \ V]].

**Record** \( inv \ Q \ B \ V : \ Prop := \{
\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)

\(\)
Main lemma about the invariant

Lemma inv_update : forall L V B Q x y,
  x \in nodes G ->
  has_edge G x y w ->
  dy = dx + w ->
  Finite dx = dist G s x ->
  inv (V\(x:=true)) B Q (new_crossing x L V) ->
  If len_gt (B\(y)) dy
  then inv (V\(x:=true)) (B\(y:=Finite dy)) (\{(y, dy)\} \u Q) ...
  else inv (V\(x:=true)) B Q (new_crossing x ((y,w):: L) V) .

Proof.
introv NxE Dy Eq [Inv SV SB]. sets_eq V': (V\(x:= true)).
lets NegP: nonneg_edges_to_path Neg. intros z. lets [Bd Bb Hc Hk]: Inv z. tests (z = y).
(* case z = y *)
forwards~ (px&Px&Wx&Mx): (@mininf_finite_inv (path int)) (eq_sym Eq).
lets Ny: (has_edge_in_nodes_r Ed).
sets p: ((x,y,w)::px).
tests (V'\(y)) as C; case_If as Nlt.
(* subcase y visited, distance improved *)
false. rewrite~ Bd in Nlt. forwards M: mininf_len_g t Nlt p; subst~ p.
rewrite weight_cons in M. math.
(* subcase y visited, distance not improved *)
...

no reference to charact. formulae

maths-style reasoning
in terms of multisets

all the nontrivial reasoning is there

180 lines across several lemmas
(1/3 of the lines in this lemma)

8 seconds to type-check in Coq
Proof script for Dijkstra's algorithm

Theorem dijkstra_spec : ∀ g x y G, ... (App dijkstra g x y) ...
Proof.
  xcf. introv Pos Ns De. unfold GraphAdjList at 1.
  hdata_simpl. xextract as N Neg Adj. xapp.
  intros Ln. rewrite <- Ln in Neg.
  xapps. xapps. xapps. xapps*. xapps.
  set (data := fun B V Q => g ~> Array N \*
  v ~> Array V \* b ~> Array B \* q ~> Heap Q).
  set (hinv := fun VQ => let '(V,Q) := VQ in
  Hexists B, data B V Q \* [inv G n s V B Q (crossing G s V)]).
  xseq (# Hexists V, hinv (V,\{})).
  set (W := lexico2
  (binary_map (count (= true)) (upto n))
  (binary_map card (downto 0))).
  xwhile_inv W hinv. refine (ex_intro' (_,_)).
  unfold hinv,data. hsimpl. applys_eq~ inv_start 2.
  permut_simpl. intros [V Q]. unfold hinv.
  xextract as B Inv. xwhile_body.
  unfold data. xapps. xret.
...
Qed.

→ 48 lines of proofs, including 8 lines of invariants; checked in 8 seconds
A typical proof obligation

Pos: nonnegative_edges G
Ns: s \in nodes G
Ne: e \in nodes G
Neg: nodes_index G n
Adj: forall x y w,
    x \in nodes G -> Mem (y, w) (N\(x)) = has_edge G x y w
Nx: x \in nodes G
Vx: ~ V\(x)
Dx: Finite dx = dist G s x
Inv: inv G n s V' B Q (new_crossing G s x L' V)
EQ: N\(x) = rev L' ++ (y, w) :: L
Ew: has_edge G x y w
Ny: y \in nodes G

(Let dy := Ret dx + w in
Let _x38 := App ml_array_get b y ; in
If_ Match
  (Case _x38 = Finite d [d] Then Ret (dy '< d) Else
   (Case _x38 = Infinite Then Ret true Else Done))
  Then (App ml_array_set b y (Finite dy) ;) ;;
    App push (y, dy) h ; Else (Ret tt))
(q ~> Pqueue Q \* b ~> Array B \* v ~> Array V' \* g ~> Array N)
(fun _:unit => hinv' L)
Summary

What you need to use CFML
→ learn Coq in case you don't know it yet
→ learn about the 25 tactics specific to CFML
→ take a Caml program and feed it to CFML
→ write down the specification of the program
→ write down the invariants of your program (hardest part)
→ complete the proofs interactively

Future work
→ support modulo and floating-point arithmetics
→ support exceptions, objects and concurrency
→ verify the characteristic formula generator itself
Further information

Online

→ examples at http://arthur.chargueraud.org/softs/cfml
→ download CFML (open source)

In the paper

→ more on how to build characteristic formulae
→ examples of first-class imperative functions

In my thesis

→ more on representation predicates (e.g., GraphAdjList)
→ proofs of soundness and completeness

Thanks!