Characteristic Formulae for the Verification of Imperative Programs

Arthur Charguéraud

Max Planck Institute for Software Systems

Harvard University 2012/02/01

Overview

Proving the 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

Properties of characteristic formulae (CF)

 $\longrightarrow$ **characteristic formulae** $\longrightarrow$ 

$\rightarrow$ CF are built automatically

$\rightarrow$ CF are built compositionaly

$\rightarrow$ CF are of linear size

$\rightarrow$ CF are displayed in a way that resembles source code

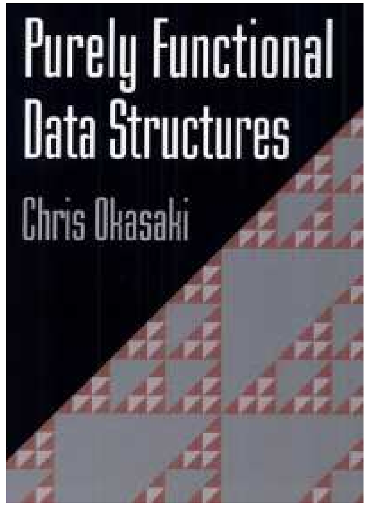
$\rightarrow$ CF can be manipulated using solely high-level tactics

$\rightarrow$ CF are not just sound but also complete

$\rightarrow$ CF support local reasoning and modular verification

Verification using CFML

Half of Okasaki's book



→ Batched queue, Bankers queue, Physicists queue, Real-time queue, Implicit queue, Bootstrapped queue, Hood-Melville queue, Leftist heap, Pairing heap, Lazy pairing heap, Splay heap, Binominal heap, Unbalanced set, Red-black set, Bottom-up merge sort, Catenable lists, Binary random-access lists

Imperative higher-order programs

→ Dijkstra's shortest path algorithm, Counter generator, Append for mutable lists, CPS-append, Iterators on mutable lists, Sparse arrays, Union-Find, Composition function, Landin's knot (recursion through the store)

Structure of the talk

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

(2) Example: verification of Dijkstra's algorithm

(3) Technical insight: how to construct CF

CF: what they are

Generation of characteristic formulae

$$C \longrightarrow [\![C]\!]$$

source code without any modification nor annotation

characteristic formula, expressed in higher-order logic using $\forall, \exists, \wedge, \Rightarrow, \ldots$

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

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

heap predicates (heap $\to$ Prop)

application in higher-order logic

total correctness Hoare triple

$\to$ 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)**

$\rightarrow$ the source code is not annotated with invariants

$\rightarrow$ instead, invariants are provided in interactive proofs

– **Not a dynamic logic**

$\rightarrow$ there is no ad-hoc logic construct to embed source code

$\rightarrow$ allows to stay in a standard logic and use existing tools

– **Not a deep embedding**

$\rightarrow$ no inductive datatype is used to represent code syntax

$\rightarrow$ avoids low-level details and issues related to binders

– **Not a shallow embedding**

$\rightarrow$ Caml functions are not represented as Coq functions

$\rightarrow$ avoids a mismatch between partial and total functions

Comparison with Ynot

– Heap predicates similar to those from Ynot

– Source language is Caml, not Coq + monad operators

$\rightarrow$ more flexible, in particular for binders

$\rightarrow$ support verification of existing code

– Verification is not established through type-checking the code with dependent types

– Auxiliary variables are never mixed with the code

Structure of the talk

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

(2) Example: verification of Dijkstra's algorithm

$\rightarrow$ source code

$\rightarrow$ specification

$\rightarrow$ invariant

$\rightarrow$ main mathematical lemma

$\rightarrow$ verification proof script

$\rightarrow$ example of a proof obligation

(3) Technical insight: how to construct CF

Source code

```
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)
```

mutable data structures

loop

pattern matching

higher-order function

abstract data structure (argument of the functor)

Material generated by CFML

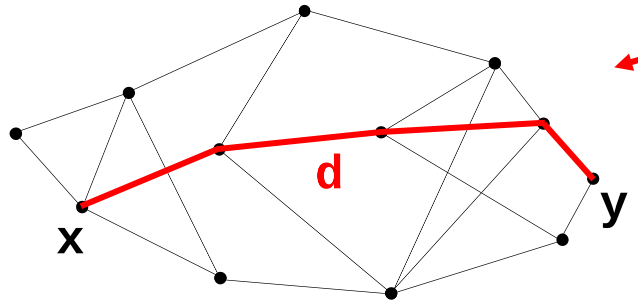
```
Axiom dijkstra : func.

Axiom dijkstra_cf :
   (@CFPrint.tag tag_top_fun _ _ (@CFPrint.tag tag_body _ _
   (forall K :  (CFHeaps.loc -> (int -> (int -> ((CFHeaps.hprop
   -> ((_ -> CFHeaps.hprop) -> Prop)) -> Prop)))), ((is_spec_3
   K) -> ((forall g : CFHeaps.loc, (forall s : int, (forall e :
   int, ((((K g) s) e) (@CFPrint.tag tag_let_trm (Label_create
   'n) _ (local (fun H : CFHeaps.hprop => (fun Q : (_ ->
   CFHeaps.hprop) => (Logic.ex (fun Q1 : (int -> CFHeaps.hprop)
   => ((Logic.and (((@CFPrint.tag tag_apply _ _ ((((@app_1
   CFHeaps.loc) int) ml_array_length)...
  (** goes on for about 100 more lines *)
```

func is the abstract data type used to represent functions

$\rightarrow$ **These axioms are justified by the soundness theorem**

Specification for Dijkstra's shortest path



mathematical graph *G*

```
Theorem dijkstra_spec : ∀ g x y G,
   nonnegative_edges G ->
   x \in nodes G ->
   y \in nodes G ->
   (App dijkstra g x y)
      (g ~> GraphAdjList G)
      (fun d => [d = dist G x y]
                \* g ~> GraphAdjList G)
```

application

pre-condition

post-condition

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 ~> GraphAdjList G
\* v ~> Array V
\* b ~> Array B
\* q ~> Pqueue q
\* [inv Q B V].
```

G : graph int
V : array bool
B : array intbar
Q : multiset (int*int)

```
Record inv Q B V : Prop := {
 Bdist: ∀x, x \in nodes G -> V\(x) = true ->
         B\(x) = dist G s x;
 Bbest: ∀x, x \in nodes G -> V\(x) = false ->
         B\(x) = mininf weight (crossing V x);
 Qcorr: ∀x, (x,d) \in Q ->
         x \in nodes G /\ ∃p, crossing V x p /\ weight p = d;
 Qcomp: ∀x p, x \in nodes G -> crossing V x p ->
         ∃d, (x,d) \in Q /\ d <= weight p;
 SizeV: length V = n;
 sizeB: length B = n }
```

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
  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 Nx Ed Dy Eq [Inv SV SB]. sets_
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_fin
lets Ny: (has_edge_in_nodes_r Ed).
sets p: ((x,y,w)::px).
asserts W: (weight p = dy). subst p.
tests (V'\(y)) as C; case_If as Nlt.
(* subcase y visited, distance impro
false. rewrite~ Bd in Nlt. forwards M
 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.
```

specialized CFML tactic

loop invariant

termination measure

reference to one mathematic lemma

→ 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
```

hypotheses

```
(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)
```

characteristic formula

pre-condition

post-condition

Summary of CFML

What you need to use CFML

$\rightarrow$ learn Coq in case you don't know it yet

$\rightarrow$ learn about the 25 tactics specific to CFML

$\rightarrow$ take a Caml program and feed it to CFML

$\rightarrow$ write down the specification of the program

$\rightarrow$ write down the invariants of your program (hardest part)

$\rightarrow$ complete the proofs interactively

Not supported (yet)

$\rightarrow$ modulo and floatting-point arithmetics

$\rightarrow$ exceptions, objects and concurrency

Structure of the talk

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

(2) Example: verification of Dijkstra's algorithm

(3) Technical insight: how to construct CF

$\rightarrow$ characteristic formula for sequences

$\rightarrow$ function definitions and function calls

$\rightarrow$ integration of the frame rule

CF construction for sequence

Hoare logic rule for sequence

$$\frac{\{H\}\, C_1\, \{H''\} \qquad \{H''\}\, C_2\, \{H'\}}{\{H\}\, (C_1\, ;\, C_2)\, \{H'\}}$$

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

$$[\![C_1\, ;\, C_2]\!] \;\equiv$$
$$\lambda H. \lambda H'. \;\exists H''. \, [\![C_1]\!]\, H\, H'' \;\wedge\; [\![C_2]\!]\, H''\, H'$$

Notation system for CF

Definition from previous slide

$$
\begin{aligned}
[\![C_1\,;\,C_2]\!] \;\equiv\; & \\
& \lambda H.\,\lambda H'.\;\; \exists H''.\; [\![C_1]\!]\; H\; H'' \;\;\wedge\;\; [\![C_2]\!]\; H''\; H'
\end{aligned}
$$

Definition of a Coq notation

$$
\begin{aligned}
(\mathcal{F}_1 \;;;\; \mathcal{F}_2) \;\equiv\; & \\
& \lambda H.\,\lambda H'.\;\; \exists H''.\; \mathcal{F}_1\; H\; H'' \;\;\wedge\;\; \mathcal{F}_2\; H''\; H'
\end{aligned}
$$

Characteristic formula for sequence, revisited

$$
[\![C_1\,;\,C_2]\!] \;\equiv\; [\![C_1]\!] \;;;\; [\![C_2]\!]
$$

Generalization

Other language constructs are handled in a similar way

$$[\![C_1\ ;\ C_2]\!] \;\equiv\; [\![C_1]\!]\ ;;\ [\![C_2]\!]$$

$$[\![\mathsf{while}\ C_1\ \mathsf{do}\ C_2]\!] \;\equiv\; \mathbf{While}\ [\![C_1]\!]\ \mathbf{Do}\ [\![C_2]\!]$$

...

As a result:

$\longrightarrow$ CF are fully compositional

$\longrightarrow$ CF are easy to generate

$\longrightarrow$ 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

$$\begin{aligned} \text{Axiom } f &: \ \text{func}. \\ \text{Axiom } F &: \ \forall x\, H\, H'.\ [\![C]\!]\, H\, H' \ \Rightarrow\ \text{App } f\, x\, H\, H'. \end{aligned}$$

where *func* is an abstract type and *App* an abstract predicate

$$\begin{aligned} \text{func} &: \ \text{Type} \\ \text{App} &: \ \forall A.\ \text{func} \to A \to (\text{Heap} \to \text{Prop}) \to (\text{Heap} \to \text{Prop}) \to \text{Prop} \end{aligned}$$

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

$$\mathsf{function}\ f\,(n)\,\{\ \mathsf{if}\ n > 0\ \mathsf{then}\ \{x := x + 1;\ f\,(n - 1)\}\ \}$$

We prove its specification by induction on *n*

$$\forall n.\ \forall a.\ n \geq 0\ \Rightarrow\ \mathsf{App}\, f\, n\, (x \hookrightarrow a)\, (x \hookrightarrow a + n)$$

Frame rule

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

$$\frac{\{H_1\}\ C\ \{H_1'\}}{\{H_1 * H_2\}\ C\ \{H_1' * H_2\}}$$

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

$$[\![C_1\ ;\ C_2]\!] \;\equiv\; \mathsf{local}\,(\lambda H.\,\lambda H'.\ \ \ldots)$$

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

$$\mathsf{local}\,\mathcal{F} \;\equiv\; \lambda H.\,\lambda H'.\ \exists H_1 H_1' H_2.\ \begin{cases} \mathcal{F}\,H_1\,H_1' \\ H = H_1 * H_2 \\ H' = H_1' * H_2 \end{cases}$$

Further information

Homepage

$\rightarrow$ examples at http://arthur.chargueraud.org/softs/cfml

$\rightarrow$ download and try CFML (open source)

ICFP'11 paper

$\rightarrow$ more on how to build characteristic formulae

$\rightarrow$ examples of first-class imperative functions

Thesis

$\rightarrow$ more on representation predicates (e.g., GraphAdjList)

$\rightarrow$ proofs of soundness and completeness

Thanks!