Characteristic Formulae for Mechanized Program Verification

Arthur Charguéraud

Advisor: François Pottier

PhD Defense, Université Paris Diderot

Paris, 2010/12/16
Big programs everywhere

Programs are everywhere
Programs are ever-more complex

→ 10 million lines of code in your pocket
What if one of those lines was incorrect?

Cell phones are not the only devices that may crash...
Bugs everywhere

If suffices to have one single line incorrect to end up with a buggy system. How can we prevent that?

1) Code review
   → extremely hard for humans to catch all bugs

2) Test
   → find some bugs, but others remain undetected

3) Static analysis (e.g. type checking)
   → find all the bugs of a particular kind

4) Mechanized verification
   → use a machine to prove the absence of bug
Specification

Definition: a specification is a description of what a program is intended to compute, regardless of how the program computes its result

Examples of specifications:
- the definition let n = ... produces a value \( n \) that is the smallest prime number greater than 90
- the function let f x = ..., when given a nonnegative integer \( x \), returns an integer equal to \( x! \)
- the function let incr r = ..., when called in a state where the location \( r \) contains an integer \( n \), changes the memory so that the location \( r \) contains \( n+1 \)
Correctness as a theorem

The statement "such program is free of bug" can be formulated as a formal theorem:

"Such program admits such specification"

→ In general, we cannot expect a machine to automatically prove theorems of this form

→ Some form of human intervention is needed

→ One possibility is to use a proof assistant (e.g., Coq, Isabelle, HOL4, ...)


# Proof assistants

<table>
<thead>
<tr>
<th>User writes:</th>
<th>Proof assistant checks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>– definitions</td>
<td>– well-formedness of definitions and statements</td>
</tr>
<tr>
<td>– statement of theorems</td>
<td>– legitimacy of each step of reasoning</td>
</tr>
<tr>
<td>– key steps of reasoning</td>
<td></td>
</tr>
</tbody>
</table>

The user does not always need to give all the details: easy steps of reasoning can be proved automatically.

No mistake possible:

**If all the steps involved in the proof of theorem are accepted, then the theorem is true**
Coq at a glance

Theorem statement

Sequence of tactics

Hypotheses

Proof obligations

Current position
Characteristic formulae

In this thesis: a new, practical approach to program verification based on **Characteristic Formulae (CF)**

**Program (Caml code)** → **Char.Formulae (Coq axioms)** → **Specification (Coq theorem)** → **Verification (Coq proof)**

CFML tool generates

Interactive Coq proof
– Introduction
– CF in the design space
– Theory: construction of CF
– Practice: Dijkstra's algorithm
– Representation predicates
– Conclusion
Interpreting the theorem

"Such piece of code admits such specification"

How to state and prove such a theorem?

→ A problem studied over the past 50 years

→ Five main approaches, summarized next
In the traditional "Verification Condition Generator" approach, no correctness theorem is stated explicitly.

- Quite effective when proofs can be automated.
- If not, need more invariants (but it takes time).
- Or need a proof assistant (but obligations are not so easy to read and not robust on change).

(Examples of modern VCGs: Why, Boogie, Jahob, VCC)
2– Shallow embeddings

"such logical definition admits such specification"

Three ways to relate the logical definition to the code

- **Extraction**
  - source code
  - logical definition
  - e.g. Compcert, Ynot

- **Decompilation**
  - source code
  - logical definition
  - e.g. Loop, Myreen's PhD

- **Proof of equiv.**
  - source code
  - logical definition
  - e.g. Sel4

→ Large-scale projects successfully formalized
→ Partial functions and side-effects need to be encapsulated in a monad (like in Haskell code)
Create new mathematical logics in which the statement "Such piece of code admits such specification" has a meaning.

Example: the Key tool, and other dynamic logics

→ **Key tool**: interactive verification of real code
→ **Need to build a new proof assistant**: overwhelming implementation effort
→ **Custom tool using custom logic**: less trustworthy than a standard proof assistant
"Such piece of syntax, when executed according to such reduction rules, admits such specification"

e.g. Mehta & Nipkow, Shao et al, etc...

→ During the 2nd year of my PhD, I built a deep embedding of the pure fragment of Caml in Coq

→ Very expressive: can prove any true property

→ Far from perfect: the explicit representation of syntax exposes many technical details

→ Characteristic formulae can be viewed as an abstract layer built on top of a deep embedding, keeping the expressiveness but hiding the details
5– Characteristic formulae

"the characteristic formula of this piece of code is a predicate that holds of such specification"

Origins of Characteristic Formulae:

– Hennessy-Milner logic (1980): two processes are bisimilar iff their characteristic formulae are equivalent

– Graf & Sifakis (1986): there exists an algorithm for computing the characteristic formula of any process

– Honda, Berger & Yoshida (2004, 2006): one can build a most-general specification (i.e. Hoare triple) of any PCF program, without referring to a representation of syntax. (Specifications expressed in an ad-hoc logic.)
Overview of the contribution

1) CF expressed in a standard higher-order logic
   → accommodates a standard proof assistant

2) CF with Separation Logic style specification
   → supports modular verification

3) CF are of linear size and easy to read
   → allows the approach to scale up

→ Thesis: generating the characteristic formula of a program and exploiting that formula in an interactive proof assistant provides an effective approach to proving that the program satisfies its specification
- Introduction
- CF in the design space
- Theory: construction of CF
  - specification language
  - description of values in Coq
  - CF for let-bindings
  - notation system for CF
  - soundness and completeness
- Practice: Dijkstra's algorithm
- Representation predicates
- Conclusion
### Specification

**Heap** $h$: finite map from locations to values

\[
\begin{align*}
    h &: \text{heap} \\
    \text{heap} &: = \text{fmap} \ \text{loc} \ \text{dyn} \\
    \text{dyn} &: = \{A:\text{Type}; v:A\}
\end{align*}
\]

**Heap predicate** $H$: description of a heap state

\[
\begin{align*}
    H &: \text{hprop} \\
    \text{hprop} &: = \text{heap} \rightarrow \text{Prop}
\end{align*}
\]

**Hoare triple**: \{\(H\)\} \(t\) \{\(Q\)\} asserts that, in an initial heap satisfying the predicate \(H\), the evaluation of the term \(t\) terminates and produces a value \(v\) such that the final heap satisfies the predicate \((Q \ v)\).

\(H\) is the *pre-condition* and \(Q\) is the *post-condition*
Example of specification

$$t = \text{let } x = !r + 1 \text{ in } s := x + 2$$

$$H = (r \rightsquigarrow 3) \times (s \rightsquigarrow 9)$$

$$Q' = \text{fun } v \Rightarrow [v = 4] \times (r \rightsquigarrow 3) \times (s \rightsquigarrow 9)$$

The Hoare triple \{H\} \(t_1\) \{Q'\} is true

$$Q' x = [x = 4] \times (r \rightsquigarrow 3) \times (s \rightsquigarrow 9)$$

$$Q = \text{fun } _:\text{unit} \Rightarrow (r \rightsquigarrow 3) \times (s \rightsquigarrow 6)$$

The Hoare triple \{Q' x\} \(t_2\) \{Q\} is true

Thus, the Hoare triple \{H\} \(t\) \{Q\} is true
Representation of values

Caml values are represented as Coq values

- Base values are translated directly: a Caml value of type `bool list` becomes a Coq value of type `list bool`

- A Caml reference of type `T ref` is described in Coq as a value of type `loc` (\( r \) has type `loc` in \( r \rightsquigarrow 3 \))

- A Caml function of type `\( T_1 \rightarrow T_2 \)` is described in Coq as a value of an abstract type called `func`, and it is specified with help of an abstract predicate called `App`

Note: for simplicity, the type "int" is mapped to "Z"
Characteristic formulae

The characteristic formula of a term \( t \), written \([t]\), is a higher-order predicate such that:

\[
\forall H. \forall Q. \quad [t] \; H \; Q \iff \{H\} \; t \; \{Q\}
\]

→ obtain a predicate capturing the behavior of a program but not referring to the syntax of its code
→ translates source code into logical predicates

Note that \([t]\) has type "hprop \(\rightarrow (A \rightarrow \text{hprop}) \rightarrow \text{Prop}\)"
CF for let-expressions

Rule:
\[
\begin{align*}
\{H\} \ t_1 \ \{Q'\} \quad & \quad \forall x. \ \{Q' \ x\} \ t_2 \ \{Q\} \\
\{H\} \ (\text{let } x = t_1 \ \text{in } t_2) \ \{Q\}
\end{align*}
\]

Goal:
\[
\forall H. \forall Q. \ \llbracket t \rrbracket \ H \ Q \iff \ \{H\} \ t \ \{Q\}
\]

Definition:
\[
\llbracket \text{let } x = t_1 \ \text{in } t_2 \rrbracket \ \equiv \\
\lambda H. \lambda Q. \ \exists Q'. \ \llbracket t_1 \rrbracket \ H \ Q' \land \forall x. \ \llbracket t_2 \rrbracket (Q' \ x) \ Q
\]
Notation system for CF

CF for let-binding:

\[
\begin{align*}
\llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket & \equiv \\
\lambda H. \lambda Q. & \exists Q'. \ [t_1] H Q' \land \forall x. \ [t_2] (Q' x) Q
\end{align*}
\]

Definition of a Coq notation:

\[
\begin{align*}
\left( \text{Let } x = F_1 \text{ in } F_2 \right) & \equiv \\
\lambda H. \lambda Q. & \exists Q'. \ F_1 H Q' \land \forall x. \ F_2 (Q' x) Q
\end{align*}
\]

CF for let-binding, reformulated:

\[
\llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket \equiv \left( \text{Let } x = \llbracket t_1 \rrbracket \text{ in } \llbracket t_2 \rrbracket \right)
\]

→ translate a source code into a logical predicate
Summary of CF generation

- \[v\] \(\equiv\) Ret \(v\)
- \[f\ v\] \(\equiv\) App \(f\ v\)
- \([\text{if } v\ \text{then } t_1\ \text{else } t_2]\) \(\equiv\) If \(v\) then \([t_1]\) else \([t_2]\)
- \([\text{let } x = t_1\ \text{in } t_2]\) \(\equiv\) Let \(x = [t_1]\) in \([t_2]\)
- \([\text{let rec } f\ x = t_1\ \text{in } t_2]\) \(\equiv\) Let rec \(f\ x = [t_1]\) in \([t_2]\)
- \([\text{crash}]\) \(\equiv\) Crash
- \([\text{while } t_1\ \text{do } t_2]\) \(\equiv\) While \([t_1]\) Do \([t_2]\)
- \([\text{for } i = a\ \text{to } b\ \text{do } t]\) \(\equiv\) For \(i = a\) To \(b\) Do \([t]\)

→ Characteristic formulae are easy to generate
→ Characteristic formulae are of linear size
→ Characteristic formulae read like source code
→ The user never needs to unfold the definitions
Soundness and completeness

**Soundness:** if the CF of a program holds of a specification, then the program satisfies this spec.

\[
\begin{cases}
[t] H Q \\
H h
\end{cases} \quad \Rightarrow \quad \exists v. \exists h'. \begin{cases}
t/h \downarrow v/h' \\
Q v h'
\end{cases}
\]

**Completeness:** if a program satisfies a specification, then the CF of that program holds of that specification

\[
t/\emptyset \downarrow n/h \quad \Rightarrow \quad [t] [] (\lambda x. [x = n])
\]

**Meaning:** characteristic formulae tell all the truth, and nothing but the truth, about the behavior of a program.
– Introduction
– CF in the design space
– Theory: construction of CF
– **Practice: Dijkstra's algorithm**
  – overview of the source code
  – material generated by CFML
  – specification and invariants
  – overview of the proof scripts
– **Representation predicates**
– Conclusion
Dijkstra's shortest path algorithm

Path of minimum weight from a node \( s \) to a node \( e \)

\( v : \) bool array marking of treated nodes
\( b : \) intbar array storing best known distances
\( q : (\text{int}^*\text{int}) \) pqueue ordering the nodes to treat

where intbar = Finite of int | Infinite
val dijkstra : ((int*int)list)array -> int -> int ->  int

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)
Module Dijkstra (Pqueue : PqueueSig).

Axiom dijkstra : func.

Axiom dijkstra Cf :

(func = datatype used to represent functions)

characteristic formula

(** goes on for about 100 more lines *)

End Dijkstra.

→ Axioms are justified by the soundness theorem
Verification of functors

→ Modular verification of modular code
Shortest path specification

\begin{definition}
\textbf{Theorem} dijkstra\_spec : \( \forall g \, x \, y \, G, \)
\begin{align*}
&\text{nonnegative\_edges } G \rightarrow \\
&x \in \text{nodes } G \rightarrow \\
y \in \text{nodes } G \rightarrow \\
&\text{(App dijkstra } g \, x \, y) \\
&(g \Rightarrow \text{GraphAdjList } G) \\
&(\text{fun } d \Rightarrow [d = \text{dist } G \, x \, y] \\
&\quad \ast g \Rightarrow \text{GraphAdjList } G)
\end{align*}
\end{definition}

→ Not very far from an informal specification: can be understood without knowledge of Coq
Main invariant

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

\[
g \mapsto \text{GraphAdjList} \ G \quad (\ast \ G : \ \text{graph} \ \text{int} \ \ast)
\]

\[
\star \ v \mapsto \text{Array} \ V \quad (\ast \ V : \ \text{array} \ \text{bool} \ \ast)
\]

\[
\star \ b \mapsto \text{Array} \ B \quad (\ast \ B : \ \text{array} \ \text{intbar} \ \ast)
\]

\[
\star \ q \mapsto \text{Pqueue} \ Q \quad (\ast \ Q : \ \text{multiset} \ (\text{int} * \text{int}) \ \ast)
\]

\[
\star \ \lbrack \text{inv} \ Q \ B \ V \rbrack.
\]

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

**Bdist:** \( \forall x, \ x \ \in \ \text{nodes} \ G \rightarrow \ V \left( x \right) = \text{true} \rightarrow \)

\( B \left( x \right) = \text{dist} \ G \ s \ x; \)

**Bbest:** \( \forall x, \ x \ \in \ \text{nodes} \ G \rightarrow \ V \left( x \right) = \text{false} \rightarrow \)

\( B \left( x \right) = \text{mininf} \ \text{weight} \ \left( \text{crossing} \ V \ x \right); \)

**Qcorr:** \( \forall x, \ (x,d) \ \in \ Q \rightarrow \)

\( x \ \in \ \text{nodes} \ G /\ \exists p, \ \text{crossing} \ V \ x \ p /\ \text{weight} \ p = d; \)

**Qcomp:** \( \forall x \ p, \ x \ \in \ \text{nodes} \ G \rightarrow \ \text{crossing} \ V \ x \ p \rightarrow \)

\( \exists d, \ (x,d) \ \in \ Q /\ d \leq \ \text{weight} \ p; \)

**SizeV:** \( \text{length} \ V = n; \)

**sizeB:** \( \text{length} \ B = n \} \)
Main lemma about 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 Nx Ed 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.
(* 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 visisted, 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 visisted, distance not improved *)
...
Verification of the code

Theorem dijkstra_spec : \forall 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.
(* -- initial state satisfies the invariant -- *)
refine (ex_intro' (_,_)). unfold hinv,data. hsimpl.
   applys_eq~ inv_start 2. permut_simpl.
(* -- verification of the loop -- *)
intros [V Q]. unfold [V Q]. unfold hinv,data. hsimpl.
   (* ---- loop condition ---- *)
   unfold data. xapps. xret.
   (* ---- loop body ---- *)
... Qed.

invariants
termination
lemma application
x-tactics

40 lines of proofs + 8 lines of invariants
20 seconds to check
Example of a proof obligation

```plaintext
Pos : nonnegative_edges G
Ns : s \in nodes G
Ne : e \in nodes G
Neg : nodes_index G n
Adj : \forall x y w : int,
     x \in nodes G \rightarrow Mem (y, w) (N\(x)) = has_edge G x y w
Nx : x \in nodes G
Vx : \sim 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

(1/6)
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)
```

well-named hypotheses (for robustness)
char. formula
pre-condition
post-condition

Example of a proof obligation
well-named hypotheses (for robustness)
char. formula
pre-condition
post-condition
– Introduction
– CF in the design space
– Theory: construction of CF
– Practice: Dijkstra's algorithm

– **Representation predicates**
  – definition of "GraphAdjList"
  – composition of predicates
  – treatment of sharing
  – relationship with capabilities

– Conclusion
Representation of graphs

**Representation predicates**: relate a data structure with the mathematical structure it describes

\[ g \sim \text{GraphAdjList}(G) \]

**Representation predicates are user-defined:**

\[ x \sim S\,X \quad \text{is equivalent to} \quad S\,X\,x \]

**Definition**

\[
\text{GraphAdjList}(G:\text{graph}\,\text{int})\,(g:\text{loc}) := \\
\quad \text{Hexists}\,(N:\text{array}(\text{list}(\text{int}*\text{int}))), \\
\quad \quad g \sim \text{Array}\,N \\
\quad \quad \text{[}\forall x, \; x \in \text{nodes}\,G \leftrightarrow \text{index}\,N\,x]\]

\[
\text{[}\forall x\;y\;w, \; x \in \text{nodes}\,G \rightarrow \text{edges}\,G \leftrightarrow \text{Mem}\,(y,w)\,(N\!(x))\text{]}]
\]
Basic data structures

Caml type: (edge list) array
where:
edge = int*int

Representation in Coq

\( g \mapsto \text{Array } N \)

\((g : \text{loc}) \ (N : \text{array} \ (\text{list edge}))\)

"array" here denotes a Coq finite map of domain [0..n(}
Recursive ownership

Caml type:  
(edge mlist) array

where:
'a mlist
= ('a * mlist) ref

Representation in Coq

g ~> ArrayOf Mlist N

(g : loc) (N : array (list edge))

g ~> Array N = g ~> ArrayOf Id N

No limits, e.g., t ~> ArrayOf (MlistOf Array) T
Sharing

Caml type:
(edge mlist) array

where:
'a mlist
= ('a * mlist) ref

Representation in Coq:

\((g \leadsto \text{Array } N) \ast (\text{GroupOf Mlist } M)\)

\((g : \text{loc}) \ (N : \text{array } \text{loc}) \ (M : \text{fmap loc (list edge)})\)
Capabilities

Representation predicates like `ArrayOf` and `GroupOf` are the Coq counterpart of the "capabilities" involved in the type system developed in the 1\textsuperscript{st} year of my PhD.

→ *Functional Translation of a Calculus of Capabilities*  
(published at ICFP 2008, with François Pottier)

This type system has been used by:
- Pottier (2008)  
  (antiframe rule)
- Pilkiewicz & Pottier (2010)  
  (monotonic state)
- Protzenko & Pottier (ongoing)  
  (language design)
  (Kripke model)
– Introduction
– CF in the design space
– Theory: construction of CF
– Practice: Dijkstra's algorithm
– Representation predicates
– Conclusion
  – examples formalized
  – future work
  – summary
Purely functional data structures

**Trees:** unbalanced, red-black

**Heaps:** splay, leftist, binomial, pairing

**Queues:** batched, lazy, realtime, bootstrapped, HoodMelville

**Dequeues:** bankers

**Lists:** concatenable, random access

Covers more than half of the book (825 lines of Caml)

→ **proofs ≈ code + spec + invariants** (in nb. of lines)

→ *Program Verification Through Characteristic formulae* (published at ICFP 2010)
Verified imperative programs

**Algorithms:**
- Dijkstra's shortest path
- Union-find (implements a partial equivalence relation)
- Sparse arrays (arrays without initialization overhead)

**Tricky functions:**
- Reynold's CPS-append function for mutable lists
- Landin's knot (recursion through the store)
Future work

Direct extensions:
- support more language features (e.g., exceptions)
- generalize the proof to non-deterministic programs

Additional reasoning rules:
- complexity analysis (time credits)
- hidden state (anti-frame rule)
- concurrency (shared invariants)

Other languages as target:
- low-level languages (C or assembly)
- object-oriented languages (e.g., Java)
Towards a fully-verified chain

**Specification**
- Mechanically-verified proofs

**Characteristic formula**
- Soundness proof on paper
- Mechanically-verified characteristic formulae

**Source code**
- OCaml compiler
- Mechanically-verified compiler

**Machine code**
- already works for a toy imperative language
- already works for the C language
Conclusion

- A new, practical approach to program verification
- **Soundness** and **completeness** proofs
- **Implementation**: CFML, from Caml to Coq
- **Examples**: verification can be achieved at fairly reasonable cost even for complex algorithms

→ **Thesis**: generating the characteristic formula of a program and exploiting that formula in an interactive proof assistant provides an effective approach to proving that the program satisfies its specification
The end!

Further information and examples: http://arthur.chargueraud.org/