Characteristic Formulae for Mechanized Program Verification

Arthur Charguéraud

Advisor: François Pottier

PhD Defense, Université Paris Diderot Paris, 2010/12/16

Big programs everywhere

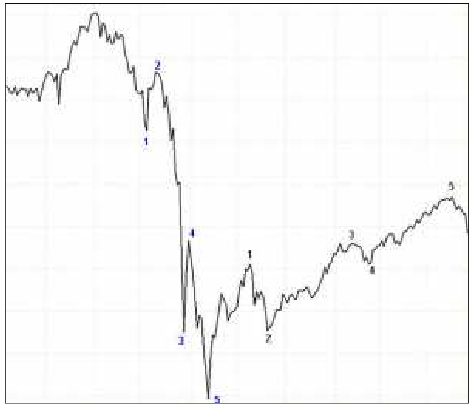
Programs are everywhere

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

User writes:

– definitions

– statement of theorems

– key steps of reasoning

Proof assistant checks:

– well-formedness of definitions and statements

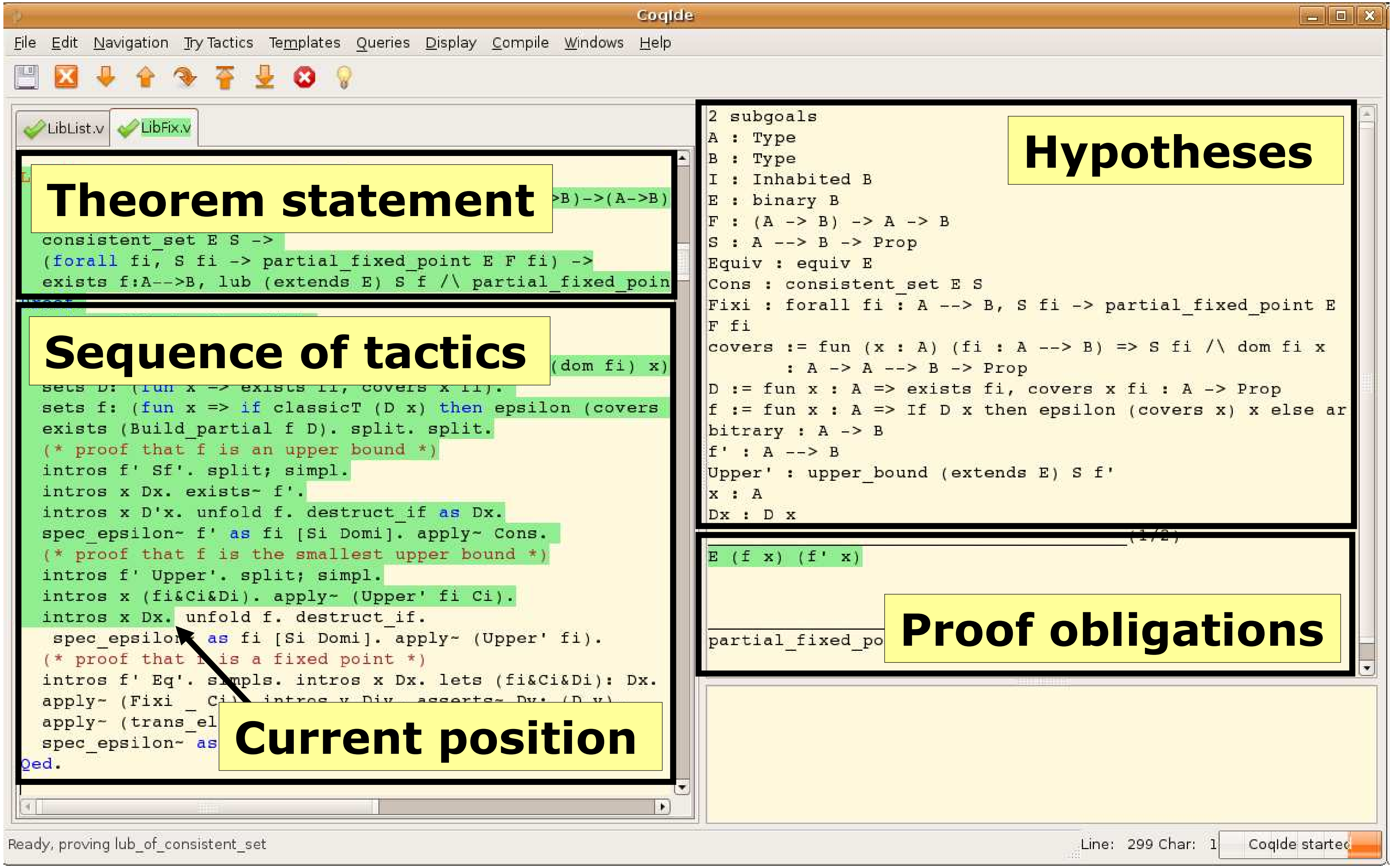
– legitimacy of each step of reasoning

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

No mistake possible:

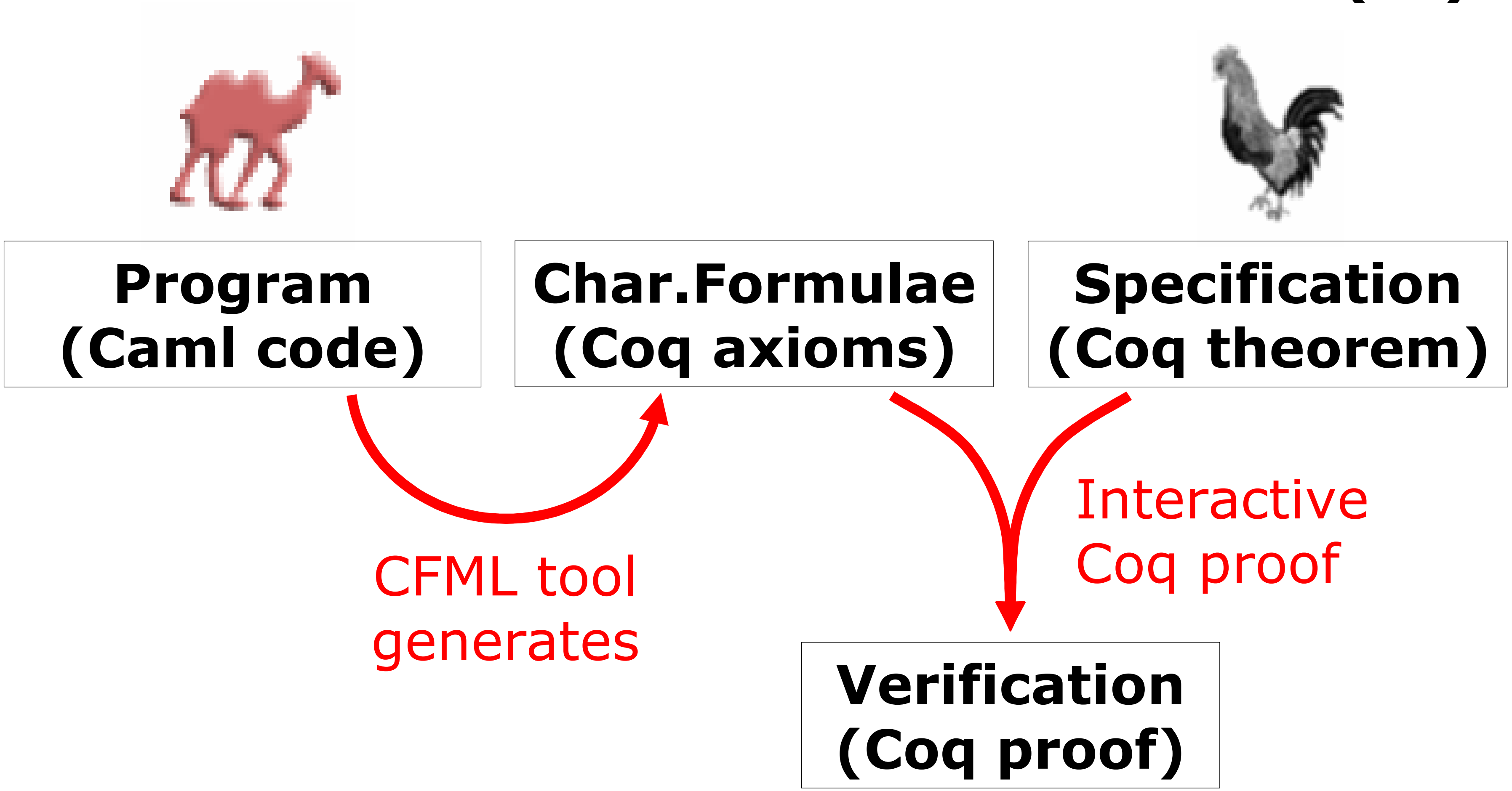
If all the steps involved in the proof of theorem are accepted, then the theorem is true

Coq at a glance



Characteristic formulae

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



- 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?

$\rightarrow$ A problem studied over the past 50 years

$\rightarrow$ Five main approches, summarized next

1–Verification Condition Generators

In the traditional "Verification Condition Generator" approach, no correctness theorem is stated explicitly

source code
specification
invariants

} **generation** → proof obligations

→ **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

source code	**Extraction** ←	logical definition	e.g. Compcert, Ynot
source code	**Decompilation** →	logical definition	e.g. Loop, Myreen's PhD
source code	**Proof of equiv.** ↔	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)**

3– Dynamic logics

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

$\rightarrow$ **Key tool: interactive verification of real code**

$\rightarrow$ **Need to build a new proof assistant: overwhelming implementation effort**

$\rightarrow$ **Custom tool using custom logic: less trustworthy than a standard proof assistant**

4– Deep embeddings

"Such piece of syntax, when executed according to such reduction rules, admits such specification"

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

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

$\rightarrow$ **Very expressive: can prove any true property**

$\rightarrow$ **Far from perfect: the explicit representation of syntax exposes many technical details**

$\rightarrow$ **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
$\rightarrow$ accomodates a standard proof assistant

2) CF with Separation Logic style specification
$\rightarrow$ supports modular verification

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

$\rightarrow$ **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

h : heap   heap := fmap loc dyn
                    dyn := {A:Type; v:A}

Heap predicate *H*: description of a heap state

H : hprop   hprop := heap $\rightarrow$ Prop

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 = `let x = !r + 1 in s := x + 2`

where t_1 = `!r + 1` and t_2 = `s := x + 2`

H = `(r ~~> 3) \* (s ~~> 9)`

Q' = `fun v => [v = 4] \* (r ~~> 3) \* (s ~~> 9)`

The Hoare triple **{*H*} *t_1* {*Q'*}** is true

Q' x = `[x = 4] \* (r ~~> 3) \* (s ~~> 9)`

Q = `fun _:unit => (r ~~> 3) \* (s ~~> 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 ~~> 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:

$$\forall H. \forall Q. \qquad [\![t]\!]\, H\, Q \quad \Longleftrightarrow \quad \{H\}\, t\, \{Q\}$$

$\rightarrow$ obtain a predicate capturing the behavior of a program but not referring to the syntax of its code

$\rightarrow$ **translates source code into logical predicates**

Note that $[\![t]\!]$ has type "hprop $\rightarrow$ (A $\rightarrow$ hprop) $\rightarrow$ Prop"

CF for let-expressions

Rule:

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

Goal: $\forall H. \forall Q. \quad [\![t]\!]\ H\ Q \quad \Longleftrightarrow \quad \{H\}\ t\ \{Q\}$

Definition:

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

$$\lambda H. \lambda Q. \ \ \exists Q'. \ [\![t_1]\!]\ H\ Q' \ \ \wedge \ \ \forall x. \ [\![t_2]\!]\ (Q'\ x)\ Q$$

Notation system for CF

CF for let-binding:

$$[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!] \;\equiv$$

$$\lambda H.\, \lambda Q.\;\; \exists Q'.\;\; [\![t_1]\!]\, H\, Q' \;\;\wedge\;\; \forall x.\;\; [\![t_2]\!]\, (Q'\, x)\; Q$$

Definition of a Coq notation:

$$(\mathbf{Let}\;\, x = \mathcal{F}_1 \;\,\mathbf{in}\;\, \mathcal{F}_2) \;\equiv$$

$$\lambda H.\, \lambda Q.\;\; \exists Q'.\;\; \mathcal{F}_1\, H\, Q' \;\;\wedge\;\; \forall x.\;\; \mathcal{F}_2\, (Q'\, x)\; Q$$

CF for let-binding, reformulated:

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

→ **translate a source code into a logical predicate**

Summary of CF generation

$$
\begin{array}{lcl}
[\![v]\!] & \equiv & \mathbf{Ret}\ v \\
[\![f\ v]\!] & \equiv & \mathbf{App}\ f\ v \\
[\![\mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2]\!] & \equiv & \mathbf{If}\ v\ \mathbf{then}\ [\![t_1]\!]\ \mathbf{else}\ [\![t_2]\!] \\
[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2]\!] & \equiv & \mathbf{Let}\ x\ =\ [\![t_1]\!]\ \mathbf{in}\ [\![t_2]\!] \\
[\![\mathsf{let\ rec}\ f\ x = t_1\ \mathsf{in}\ t_2]\!] & \equiv & \mathbf{Let\ rec}\ f\ x = [\![t_1]\!]\ \mathbf{in}\ [\![t_2]\!] \\
[\![\mathsf{crash}]\!] & \equiv & \mathbf{Crash} \\
[\![\mathsf{while}\ t_1\ \mathsf{do}\ t_2]\!] & \equiv & \mathbf{While}\ [\![t_1]\!]\ \mathbf{Do}\ [\![t_2]\!] \\
[\![\mathsf{for}\ i = a\ \mathsf{to}\ b\ \mathsf{do}\ t]\!] & \equiv & \mathbf{For}\ i = a\ \mathbf{To}\ b\ \mathbf{Do}\ [\![t]\!]
\end{array}
$$

→ 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.

$$\left\{ \begin{array}{l} [\![t]\!]\, H\, Q \\ H\, h \end{array} \right. \quad \Rightarrow \quad \exists v.\, \exists h'.\, \left\{ \begin{array}{l} t_{/h} \Downarrow v_{/h'} \\ Q\, v\, h' \end{array} \right.$$

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

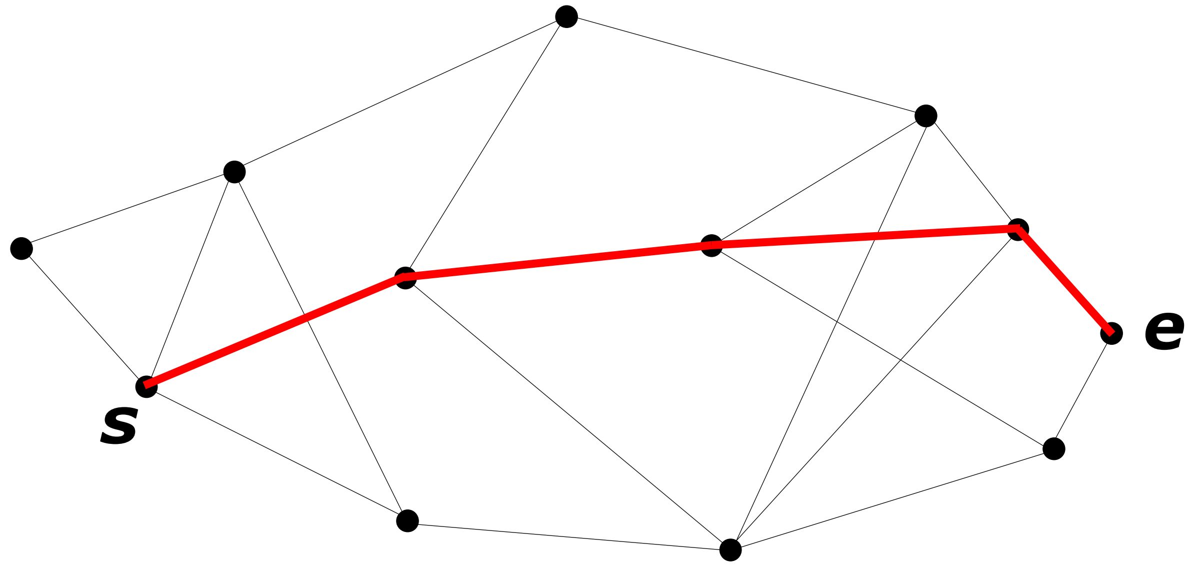
$$t_{/\emptyset} \Downarrow n_{/h} \quad \Rightarrow \quad [\![t]\!]\, [\,]\, (\lambda x.\, [x = n])$$

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 : (int*int) pqueue — ordering the nodes to treat

where intbar = Finite of int | Infinite

Implementation

```
val dijkstra : ((int*int)list)array -> int -> int -> intbar
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 structures

loop

pattern matching

higher-order function

abstract data structure

Material generated by CFML

```
Module Dijkstra (Pqueue : PqueueSig).

Axiom dijkstra : func.
```

func = datatype used to represent functions

```
Axiom dijkstra_cf :
  (@CFPrint.tag tag_top_fun _ _ (@CFPrint.tag tag_body _ _ (forall K :
  (CFHeaps.loc -> (int -> (int -> ((CFHeaps.hprop -> ((_ -> CFHeaps.hprop) ->
  Prop)) -> Prop)))), ( ... : CFHeaps.loc, (forall s :
  int, (forall e : int, ... ag tag_let_trm (Label_create
  'n) _ (local (fun H ... (_ -> 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 *)

End Dijkstra.
```

characteristic formula

→ **Axioms are justified by the soundness theorem**

Verification of functors

→ **Modular verification of modular code**

Shortest path specification

```
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).
```

mathematical graph

pre-condition

post-condition

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

Main invariant

```
Definition hinv Q B V : hprop :=
   g ~> GraphAdjList G  (* G : graph int *)
\* v ~> Array V         (* V : array bool *)
\* b ~> Array B         (* B : array intbar *)
\* q ~> Pqueue Q        (* Q : multiset(int*int) *)
\* [inv Q B V].

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 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.
(* 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 CF

maths-style reasoning in terms of multisets

All the nontrivial reasoning is there

180 lines of proofs in total for the invariant (a third in this lemma)

8 seconds to check

Verification of the code

```
Theorem dijkstra_spec : ∀ g x y G, ... (App dijkstra
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
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
(* ---- loop conditio
unfold data. xapps. x
(* ---- loop body --
...
Qed.
```

x-tactics

invariants

termination

lemma application

40 lines of proofs + 8 lines of invariants

20 seconds to check

Example of a 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 : int,
      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
______________________________________(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

- **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

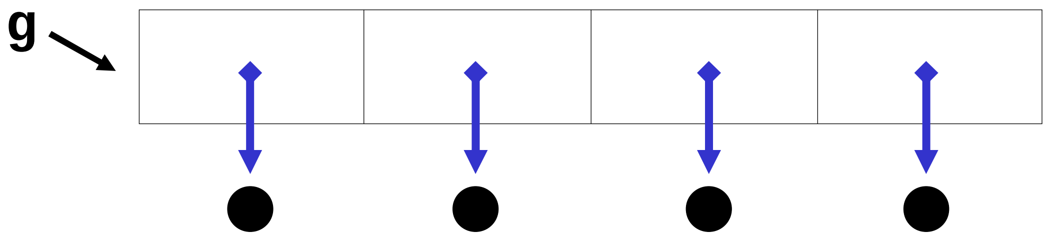
```
g ~> GraphAdjList G
```

Representation predicates are user-defined:

`x ~> S X` is equivalent to `S X x`

```
Definition GraphAdjList (G:graph int) (g:loc) :=
 Hexists (N:array(list(int*int))),
      g ~> Array N
   \* [∀x, x \in nodes G <-> index N x]
   \* [∀x y w, x \in nodes G ->
         (x,y,w) \in edges G <-> Mem (y,w) (N\(x))]
```

Basic data structures




```
Caml type:
(edge list) array

where:
edge = int*int
```

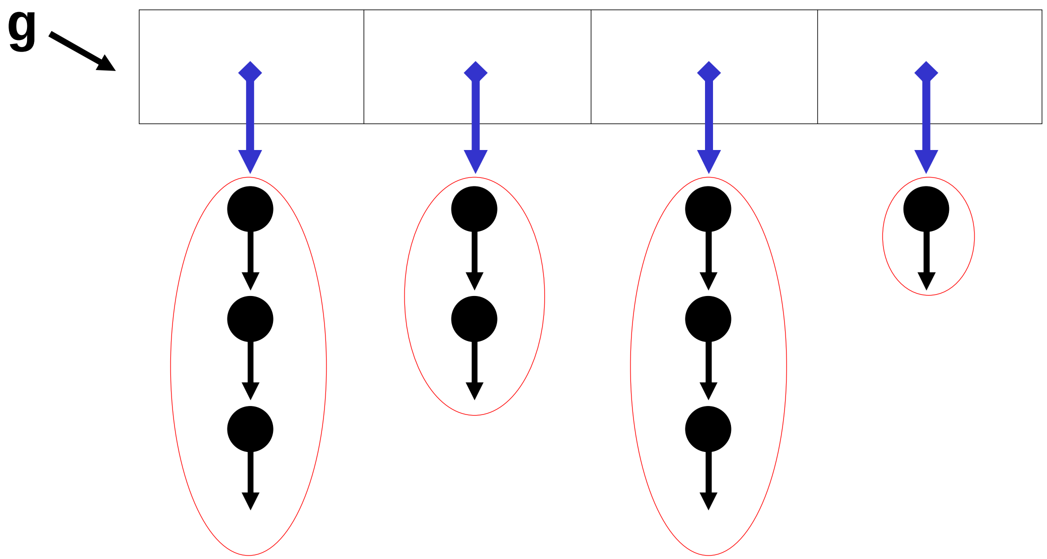
Representation in Coq

```
g ~> Array N

(g : loc) (N : array (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

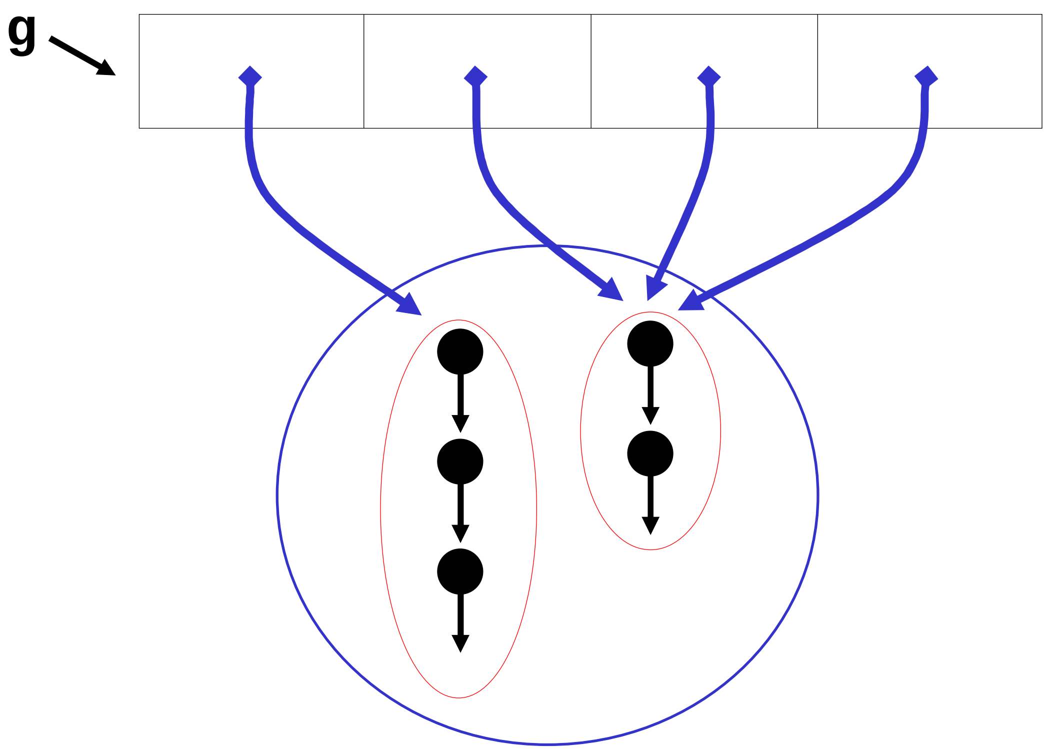
```
g ~> ArrayOf Mlist N

(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 ~> Array N) \* (GroupOf Mlist M)
(g : loc) (N : array loc) (M : 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 1st 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)
– Birkedal et al (2009, 2010)	(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

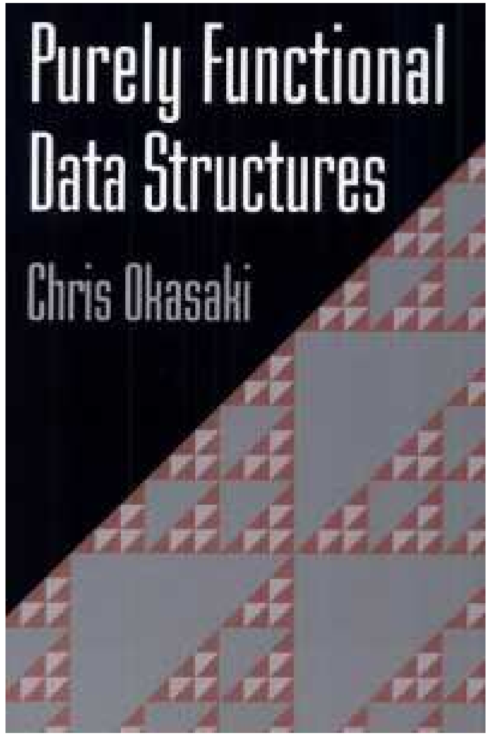
Trees: unbalanced, red-black

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)

$\rightarrow$ **proofs** $\approx$ **code + spec + invariants** (in nb. of lines)

$\rightarrow$ *Program Verification Through Characteristic formulae*
(published at ICFP 2010)

Verified imperative programs

Algorithms:

– Dijsktra'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

already works for a toy imperative language

Source code

OCaml compiler ↓ Mechanically-verified compiler

already works for the C language

Machine code

Conclusion

– **A new, pratical 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/