Program Verification Using Characteristic Formulae

Arthur Charguéraud

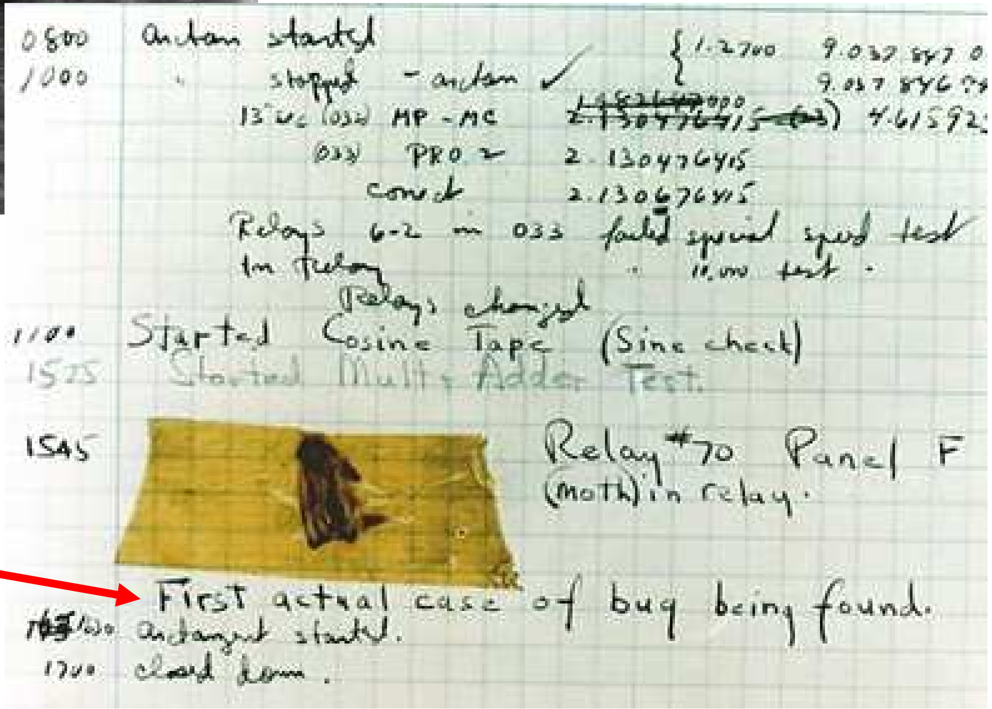
Journée du LRI

2013/07/01

Bugs in the old days

Mark-I computer (3Hz)

"First actual case of bug being found."

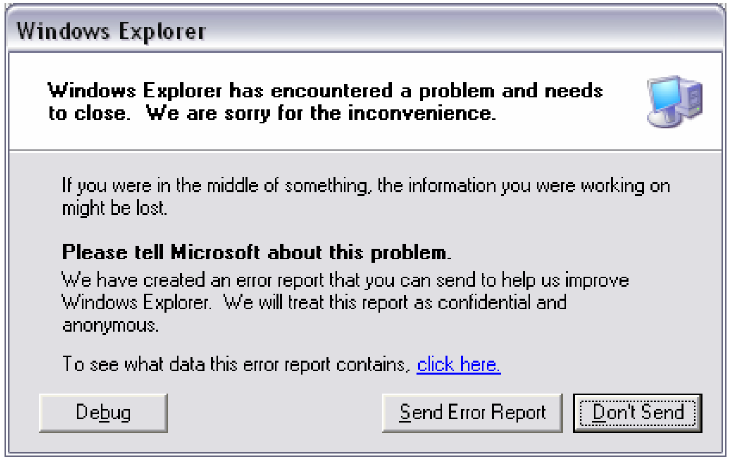


0800 Arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 0 / 9.037 846 9
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~) 4.61592
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10,000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Arctangent started.
1700 closed down.

Modern bugs

Bugs at every levels:

- applications
- operating system
- drivers
- hardware



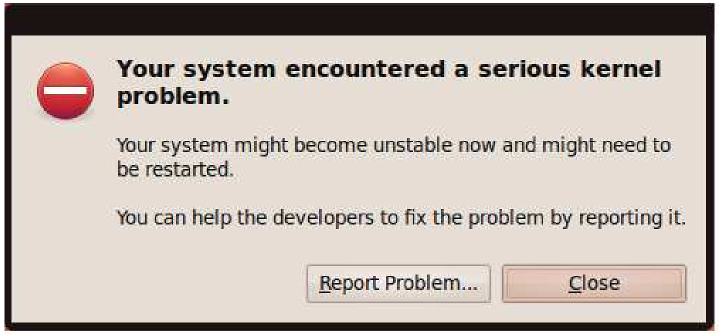
Windows Explorer has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

We have created an error report that you can send to help us improve Windows Explorer. We will treat this report as confidential and anonymous.

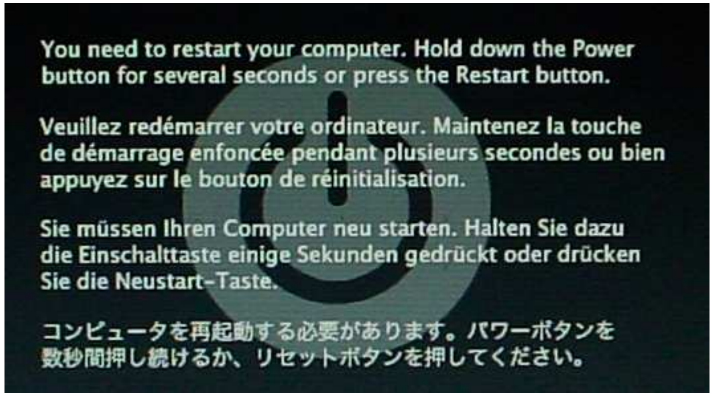
To see what data this error report contains, click here.



Your system encountered a serious kernel problem.

Your system might become unstable now and might need to be restarted.

You can help the developers to fix the problem by reporting it.



You need to restart your computer. Hold down the Power button for several seconds or press the Restart button.

Veuillez redémarrer votre ordinateur. Maintenez la touche de démarrage enfoncée pendant plusieurs secondes ou bien appuyez sur le bouton de réinitialisation.

Sie müssen Ihren Computer neu starten. Halten Sie dazu die Einschalttaste einige Sekunden gedrückt oder drücken Sie die Neustart-Taste.

コンピュータを再起動する必要があります。パワーボタンを数秒間押し続けるか、リセットボタンを押してください。

Bugs in the real world

Appearance of bugs

Example:

Visible bugs

- system freezing → windows blue screen
- erratic behaviors → music that loops

Silent bugs

- incorrect results → false numeric results
- security holes → stolen informations

→ security holes are not always due to bugs,
but bugs can sometimes be exploited by attackers

Bug hunting

Code review

- 10 million lines of code... one bug is enough
- some pieces of code are very complex

Testing

- can find many bugs, but not all
- too many cases to test

More bug hunting

Static analysis

- finds all the bugs of a particular form
- successful example: type checking

Mechanized verification

- build a proof of the absence of bugs
- have this proof checked by a program (a theorem prover)

Specification of a program

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

Examples:

– 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 of a program

Intuition: "**the program *P* is free of bugs**".

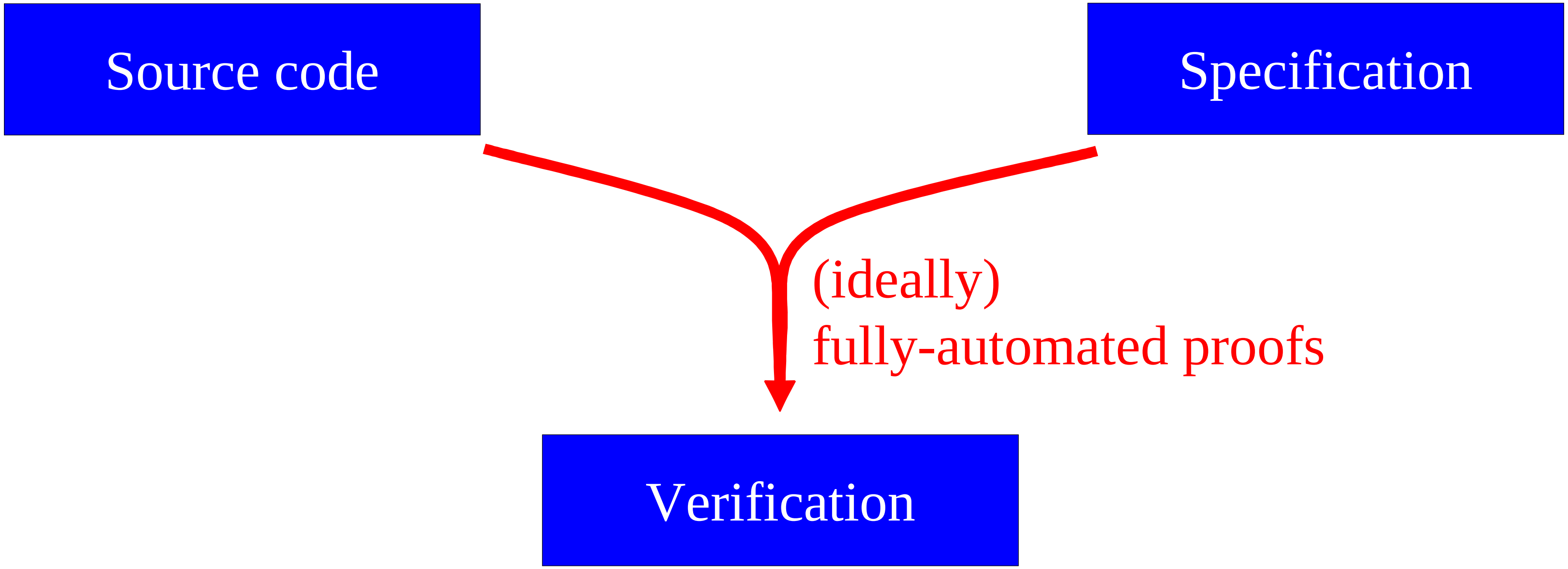
Formalization: "**the program *P* satisfies the specification *S***".

Two aspects are critical:

→ the adequacy of the specification

→ the correctness of the theorem prover

The verifying compiler



Source code

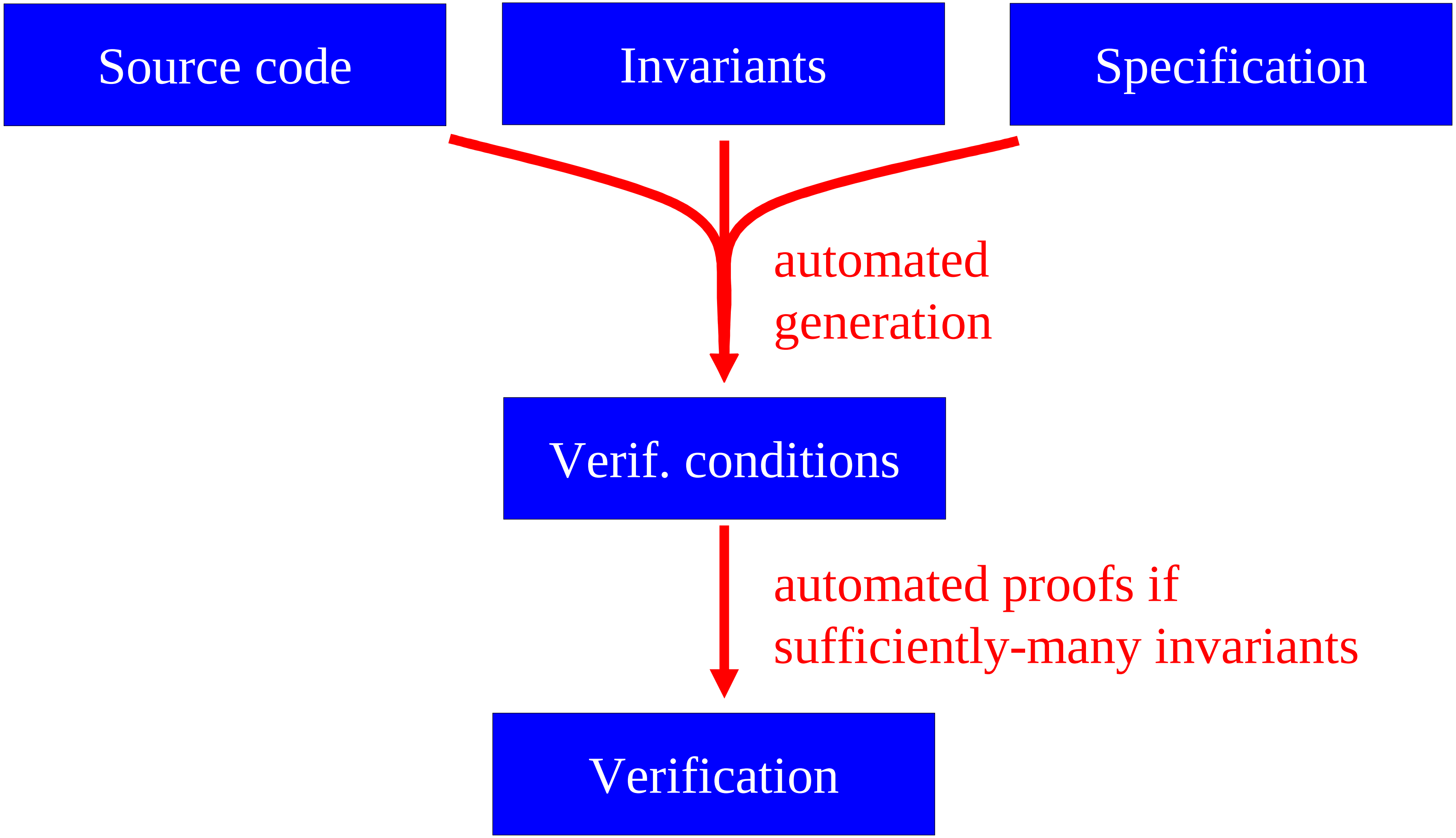
Specification

(ideally)
fully-automated proofs

Verification

→ one of the unfulfilled promises of Artificial Intelligence...

Verification condition generation



Source code

Invariants

Specification

automated
generation

Verif. conditions

automated proofs if
sufficiently-many invariants

Verification

Machine-checked mathematical proofs

Feit-Thompson theorem

"all odd groups are solvable", or equivalently,

"every nonabelian finite simple group has even order"

→ Conjecture by Burnside (1911)

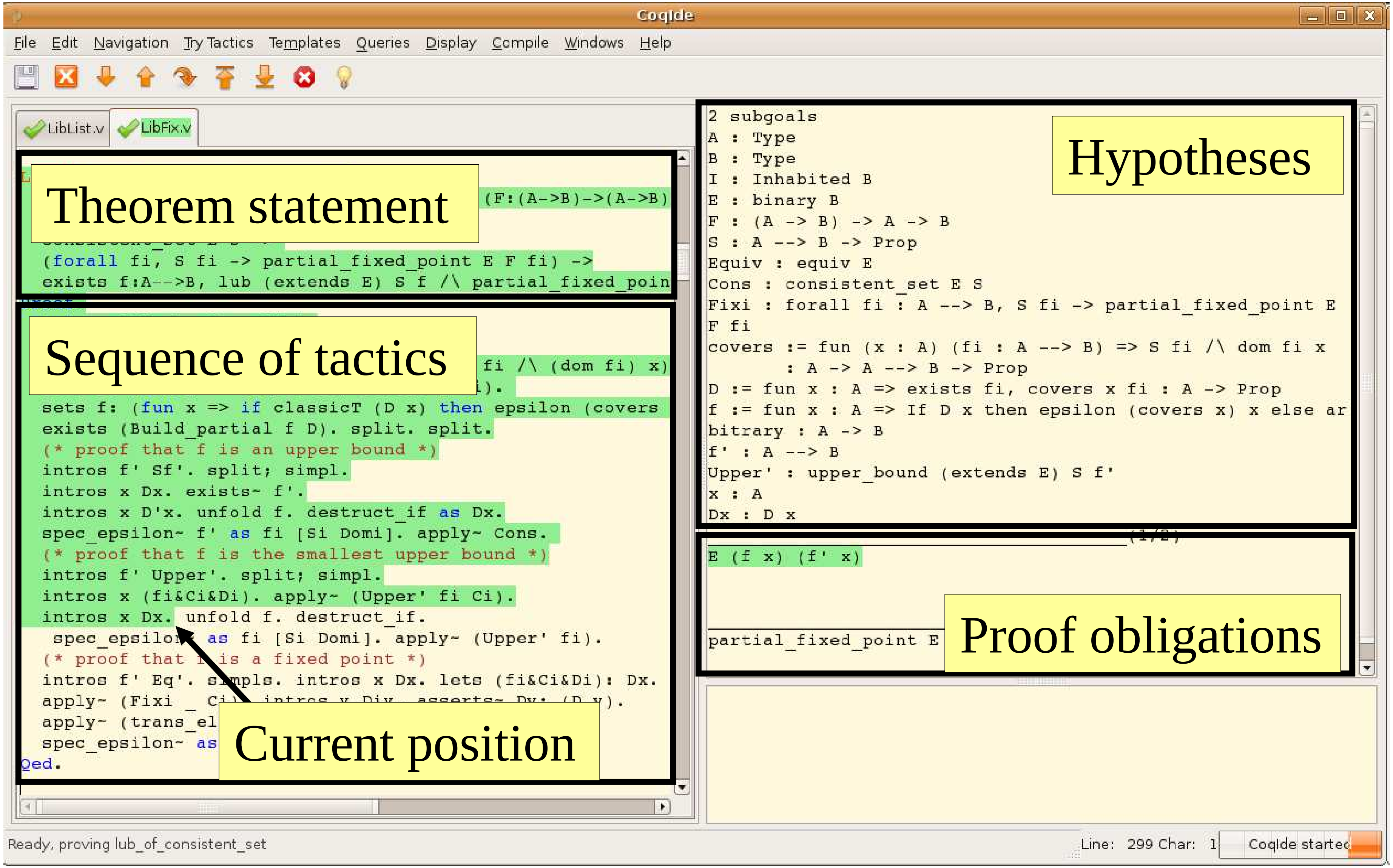
→ Original proof (1962): 255 pages

→ Revised proof (1995): two books

→ Machine-checked proof by Gonthier et al (2012)

170,000 lines in the Coq proof assistant

Coq at a glance



Theorem statement

```
                                                      (F:(A->B)->(A->B)
  (forall fi, S fi -> partial_fixed_point E F fi) ->
  exists f:A-->B, lub (extends E) S f /\ partial_fixed_poin
```

Sequence of tactics

```
                                                      fi /\ (dom fi) x)
                                                     i).
  sets f: (fun x => if classicT (D x) then epsilon (covers
  exists (Build_partial f D). split. split.
  (* proof that f is an upper bound *)
  intros f' Sf'. split; simpl.
  intros x Dx. exists~ f'.
  intros x D'x. unfold f. destruct_if as Dx.
  spec_epsilon~ f' as fi [Si Domi]. apply~ Cons.
  (* proof that f is the smallest upper bound *)
  intros f' Upper'. split; simpl.
  intros x (fi&Ci&Di). apply~ (Upper' fi Ci).
  intros x Dx. unfold f. destruct_if.
   spec_epsilon~ as fi [Si Domi]. apply~ (Upper' fi).
  (* proof that f is a fixed point *)
  intros f' Eq'. simpls. intros x Dx. lets (fi&Ci&Di): Dx.
  apply~ (Fixi _ Ci).
  apply~ (trans_el
  spec_epsilon~ as
Qed.
```

Current position

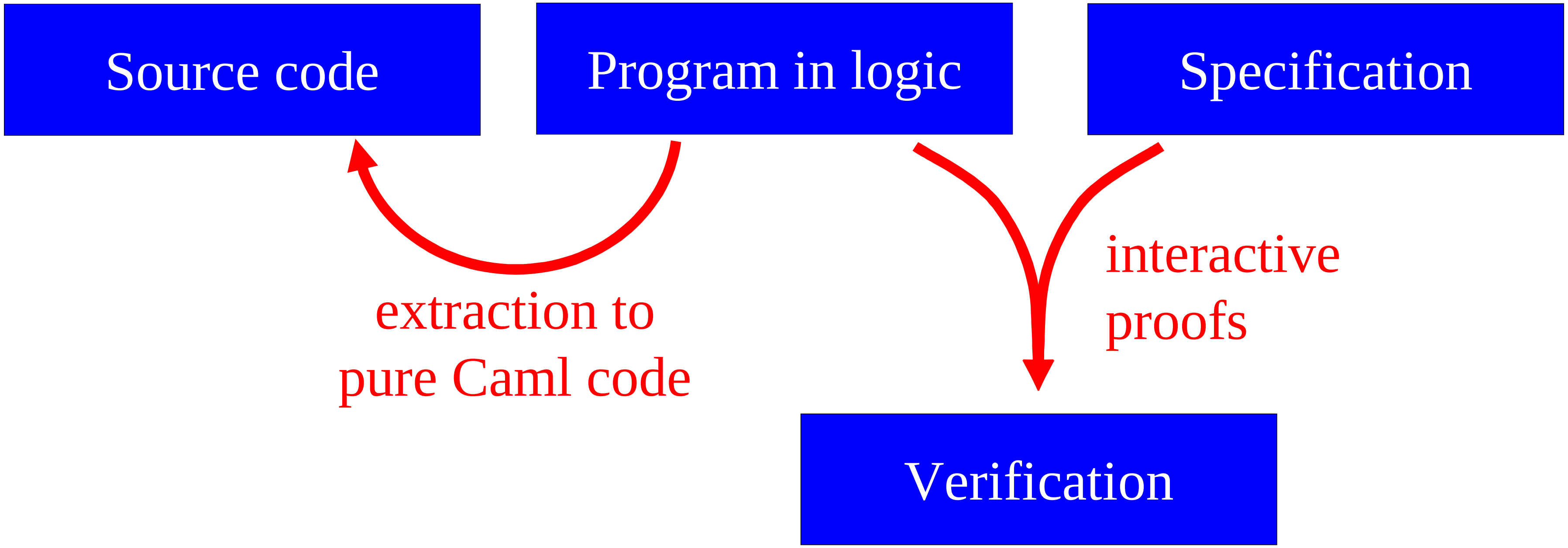
Hypotheses

```
2 subgoals
A : Type
B : Type
I : Inhabited B
E : binary B
F : (A -> B) -> A -> B
S : A --> B -> Prop
Equiv : equiv E
Cons : consistent_set E S
Fixi : forall fi : A --> B, S fi -> partial_fixed_point E
F fi
covers := fun (x : A) (fi : A --> B) => S fi /\ dom fi x
       : A -> A --> B -> Prop
D := fun x : A => exists fi, covers x fi : A -> Prop
f := fun x : A => If D x then epsilon (covers x) x else ar
bitrary : A -> B
f' : A --> B
Upper' : upper_bound (extends E) S f'
x : A
Dx : D x
______________________________________________(1/2)
E (f x) (f' x)

______________________________________________(2/2)
partial_fixed_point E
```

Proof obligations

```
Ready, proving lub_of_consistent_set                    Line:  299 Char:  1    CoqIde started
```

Extraction from Coq



Source code

Program in logic

Specification

extraction to
pure Caml code

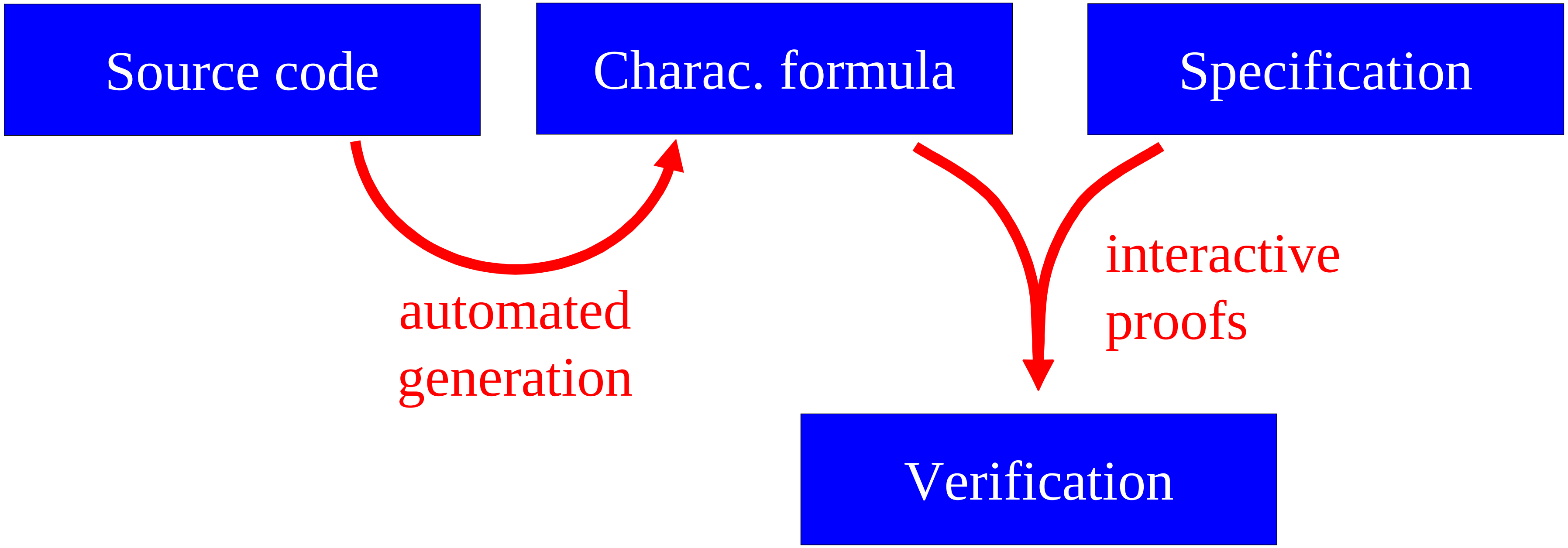
interactive
proofs

Verification

→ e.g. Leroy's verified C compiler (Compcert)

→ this approach only applies to purely-functional programs

Verification using characteristic formulae


Source code

Charac. formula

Specification

automated
generation

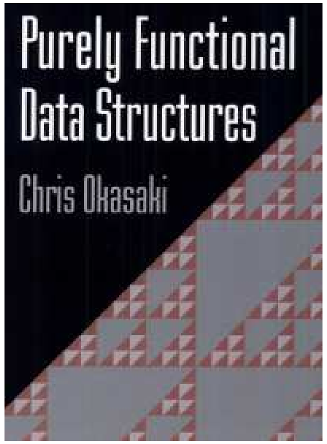
interactive
proofs

Verification

→ supports imperative programs in potentially any language

→ implementation for Caml in a tool called CFML

Programmes verified using CFML


Purely Functional Data Structures
Chris Okasaki

Purely functional data structures

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, finger trees

Imperative programs

→ Dijkstra shortest path, Union-Find, sparse array, mutable lists and trees
→ functions with local state (gensym)
→ higher-order functions (List.iter, compose)
→ CPS functions (CPS-append)
→ functions stored in memory cells (Landin's knot)

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)
```

Formulae generated by CFML

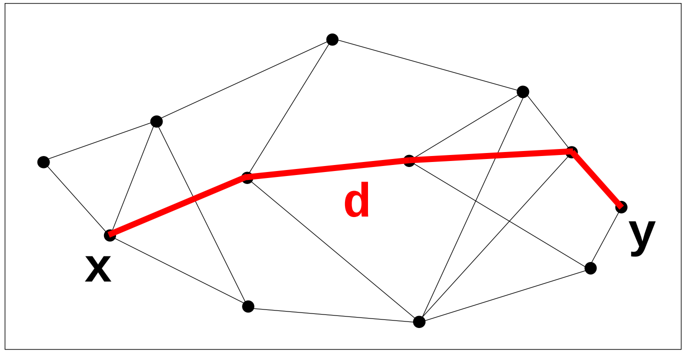
```
Axiom dijkstra : func.

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 *)

Displayed in Coq in a more readable form:

```
...
  (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))
```

Specification



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

Proof script

```
Theorem dijkstra_spec : ∀ g x y G, ... (App dijkstra g x y) ...
Proof.
xcf. introv Pos Ns Ne. 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 tactic

loop invariant

termination measure

lemma about the invariant

– 180 lines in auxiliary lemmas

– 48 lines in the proof of this theorem

Proof obligations

hypotheses

```
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
```

---

characteristic formula

```
(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))
```

pre-condition

```
(q ~> Pqueue Q \* b ~> Array B \* v ~> Array V' \* g ~> Array N)
```

```
(fun _:unit => hinv' L)
```

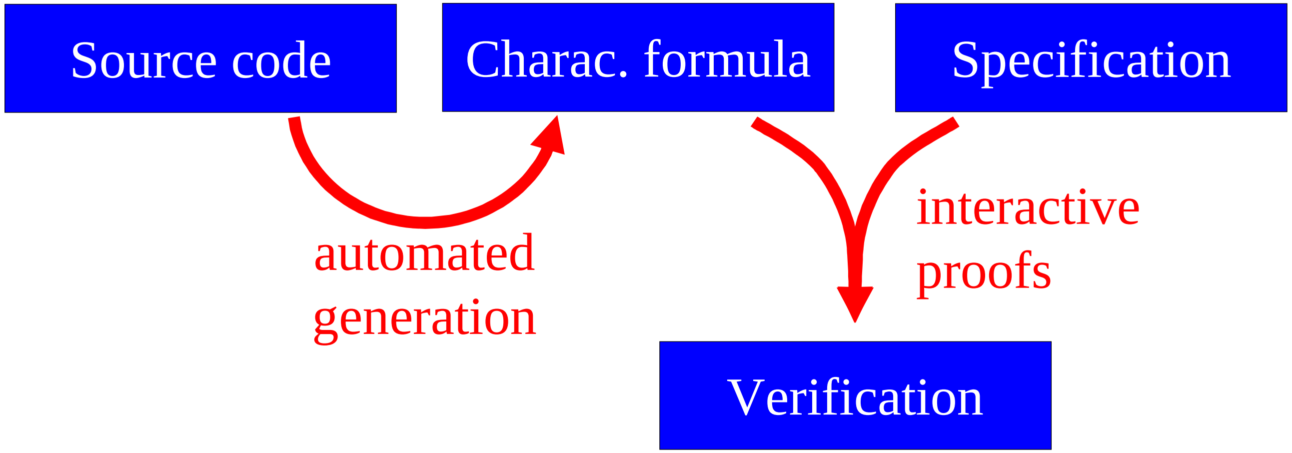
post-condition

Summary

→ bugs will have more and more impact on our daily life

→ use mechanized proofs to prove the absence of bugs

→ characteristic formulae support complex imperative programs



Source code

Charac. formula

Specification

automated generation

interactive proofs

Verification

On-going work:

→ adding support for exceptions and floating point arithmetic

→ developing characteristic formulae for C programs