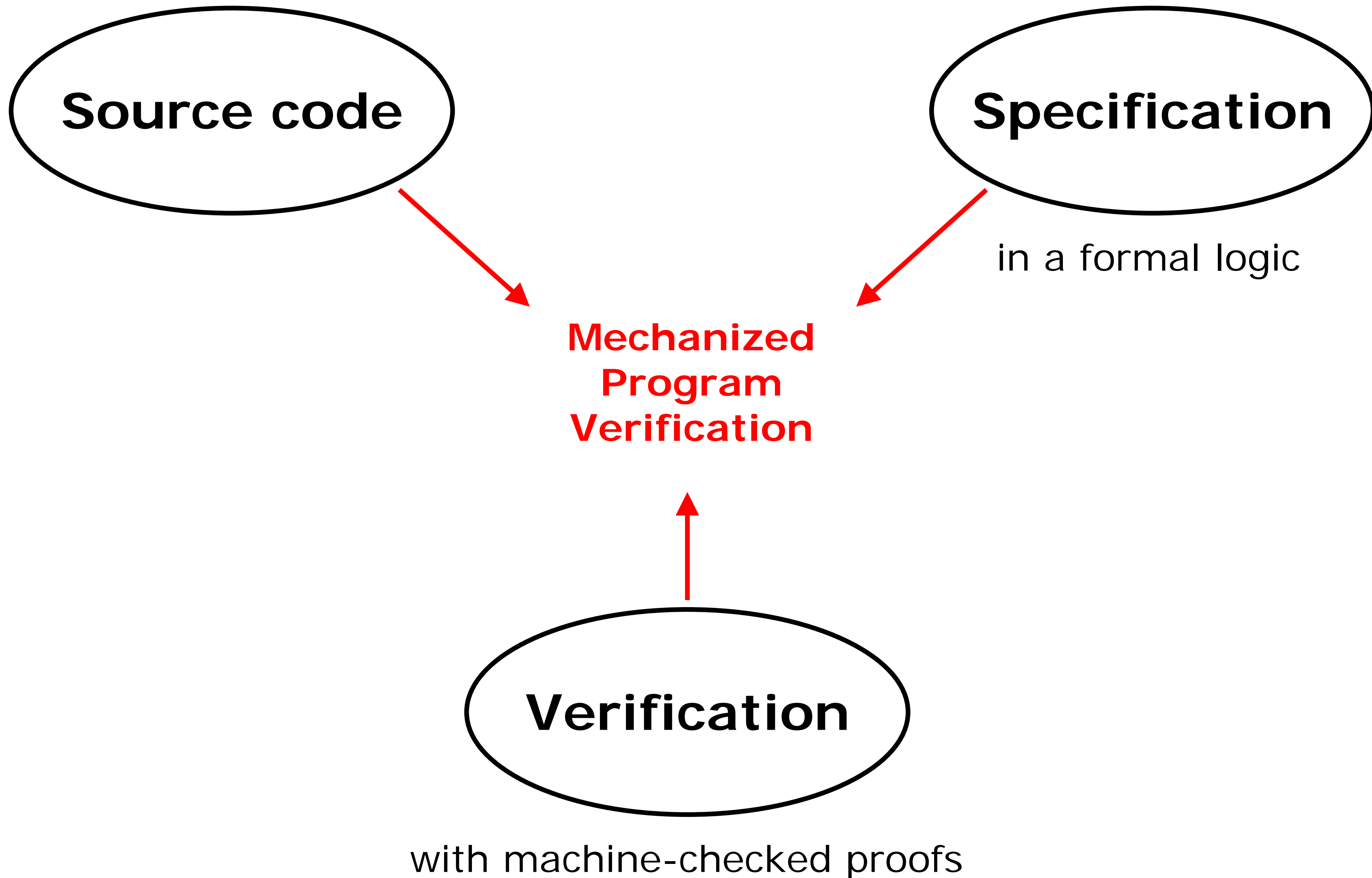
Program Verification Through Characteristic Formulae

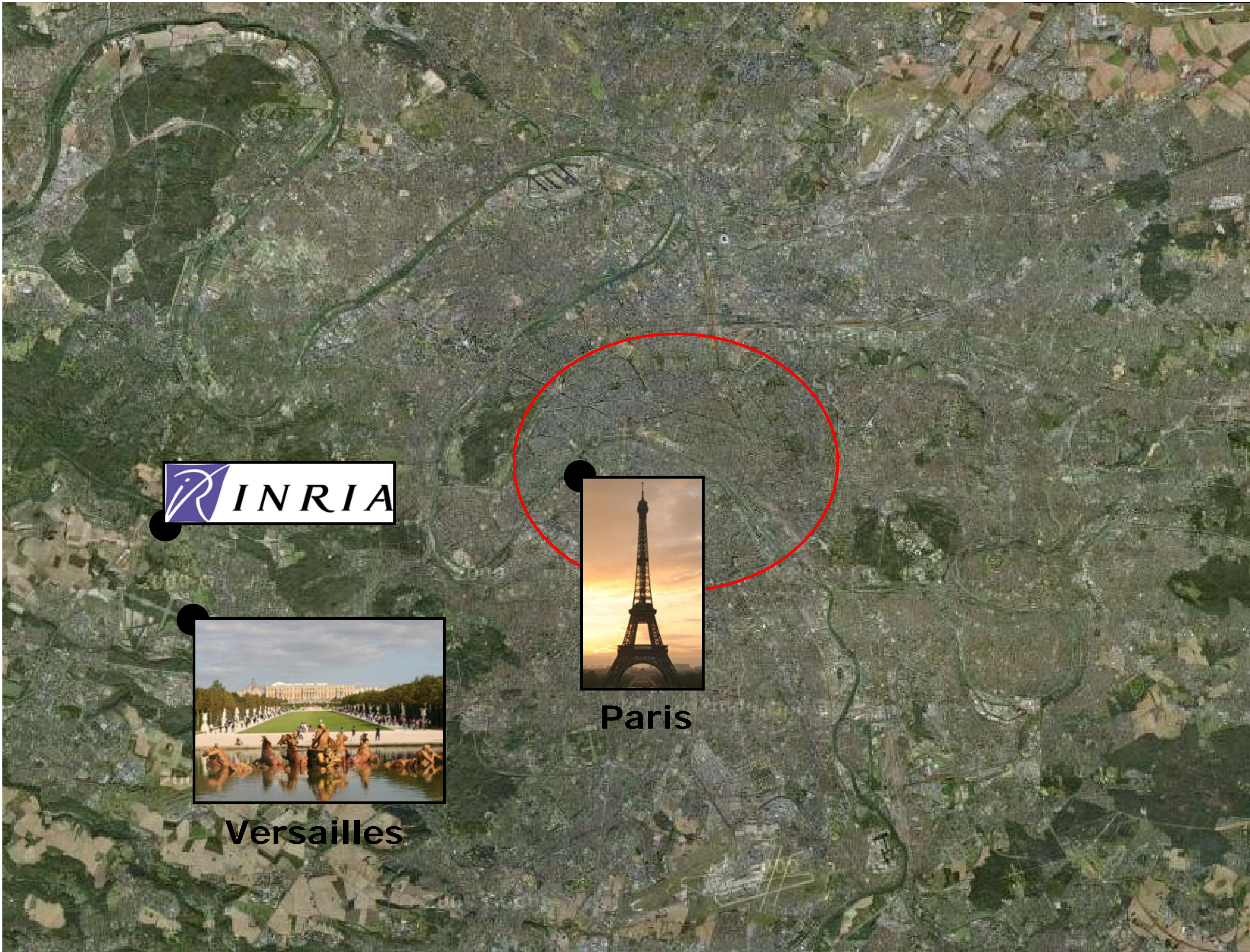
Arthur Charguéraud

INRIA

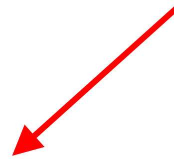
MPI-SWS Saarbrücken, 2010/03/24

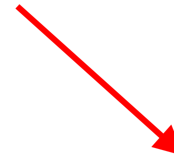
Program verification




INRIA
Paris
Versailles

Tools from INRIA





OCaml

A functional
programming language

Coq

A proof assistant
(higher-order logic)

→ We use Coq to specify and verify OCaml programs

Example: Okasaki's red-black trees

```
module RedBlackSet (Element : Ordered) : Fset = struct
 type color = Red | Black
 type tree = Empty | Node of color * tree * elem * tree

 let rec member x = function
 | Empty -> false
 | Node (_,a,y,b) -> if Element.lt x y then member x a
                     else if Element.lt y x then member x b else true

 let balance = function
 | (Black, Node (Red, Node (Red, a, x, b), y, c), z, d)
 | (Black, Node (Red, a, x, Node (Red, b, y, c)), z, d)
 | (Black, a, x, Node (Red, Node (Red, b, y, c), z, d))
 | (Black, a, x, Node (Red, b, y, Node (Red, c, z, d))) ->
     Node (Red, Node (Black, a, x, b), y, Node (Black, c, z, d))
 | (c,a,x,y) -> Node (c,a,x,y)

 let rec insert x s =
   let rec ins = function
     | Empty -> Node (Red, Empty, x, Empty)
     | Node (col, a, y, b) as s ->
       if Element.lt x y then balance (col, ins a, y, b)
       else if Element.lt y x then balance (col, a, y, ins b)
       else s in
  let Node (_, a, y, b) = ins s in Node (Black, a, y, b)
```

Subset of OCaml considered

Support for the core language and modules:

– algebraic datatypes

– pattern matching

– recursion and mutual recursion

– higher-order functions

– lazy evaluation (when code terminates even in cbv)

– modules and functors (when expressible in Coq)

But no support for:

– side effects (on-going work)

– exception-catching (future work)

– objects (used by few OCaml programmers)

Interactive proofs with Coq

CoqIde

File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help

LibList.v LibFix.v

```
Lemma lub_of_consistent_set :
  forall A B {I:Inhabited B} (E:binary B) (F:(A->B)->(A->B))
  equiv E ->
  consistent_set E S ->
  (forall fi, S fi -> partial_fixed_point E F fi) ->
  exists f:A-->B, lub (extends E) S f /\ partial_fixed_poin
Proof.
  introv I Equiv Cons Fixi.
  (* construct a function f *)
  sets covers: (fun (x:A) (fi:A-->B) => S fi /\ (dom fi) x)
  sets D: (fun x => exists fi, covers x fi).
  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). intros y Diy. asserts~ Dy: (D y).
  apply~ (trans_elim (f y)). unfold f. destruct_if.
  spec_epsilon~ as fj [Sj Domj]. apply~ Cons.
Qed.
```

```
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 F (Build_partial f D)
```

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

Interest of using Coq

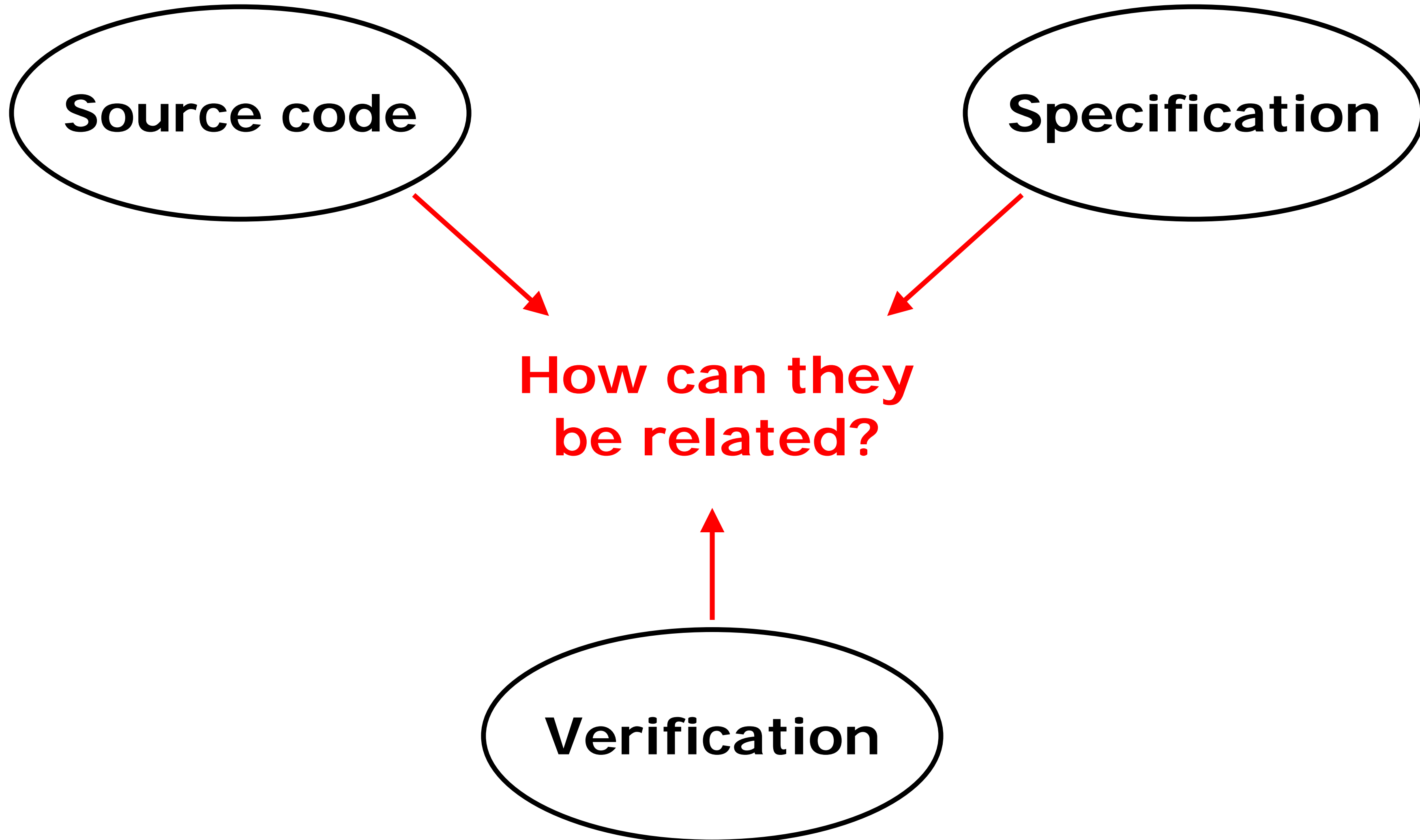
– **Very expressive:** Coq is based on the Calculus of Construction (in rough System F with full dependent types), extended with inductive and coinductive definitions.

→ We can always formalize in Coq the invariant that we have in mind in a relatively natural way.

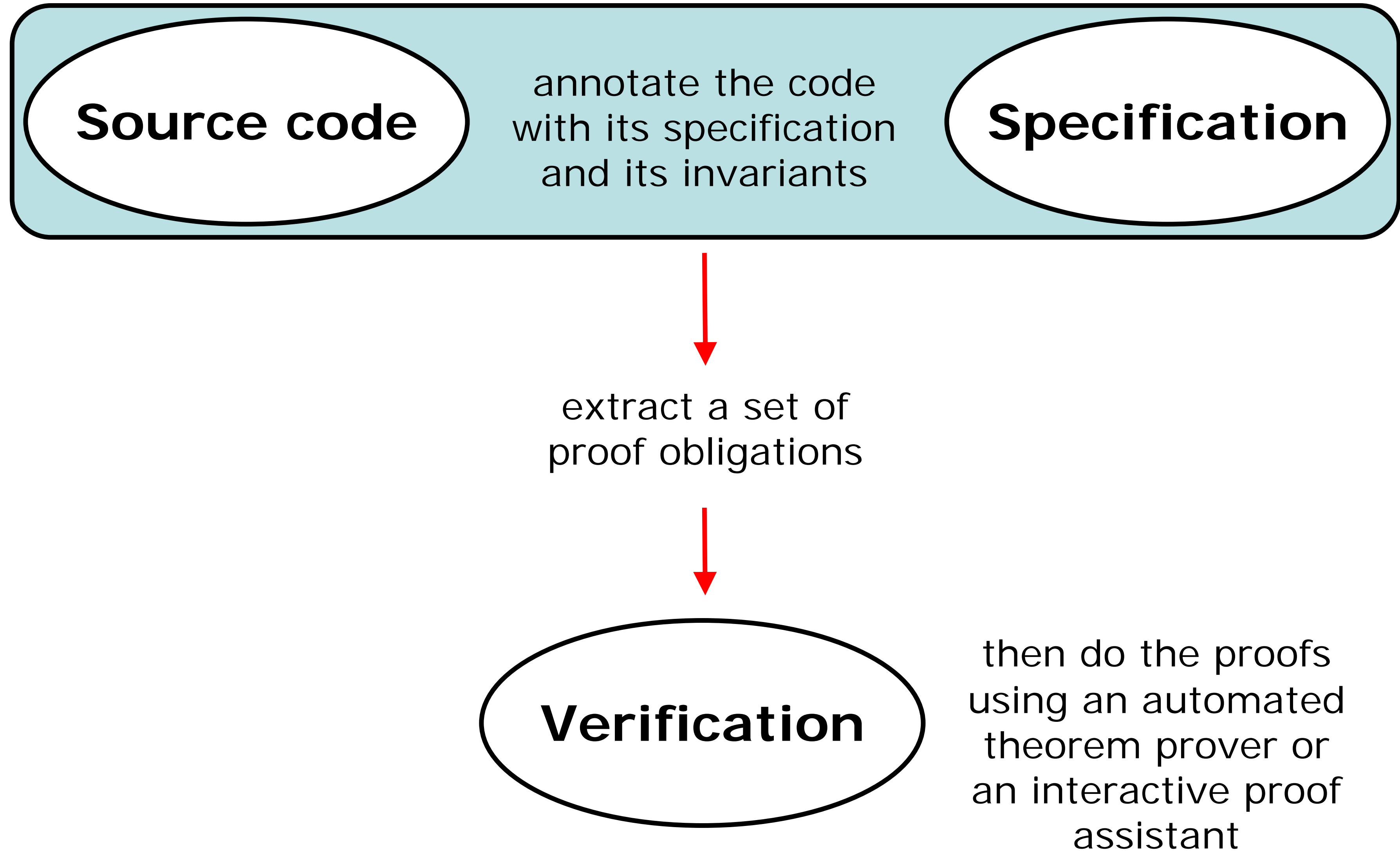
– **Customizable automation, notation and tactics:** allows the user to conduct arbitrarily-complex proofs with reasonable effort.

→ Makes it possible to work at a relatively high-level

Verification approaches



1) Verification Condition Generator



A VCG for OCaml

```
let rec union (a, b) where (wf_bst_avl a) and (wf_bst_avl b)
  returns c where (wf_bst_avl c)
              and (elements c) = (elements a) \u (elements b) =
    match a with
    | Empty -> b
    | Node (ah, al, ax, ar) ->
      match b with
      | Empty -> a
      | Node (bh, bl, bx, br) ->
        let ha = height(a) in
        let hb = height(b) in
        if ha >= hb then
          if hb = 1 then add (bx, a)
          else
            let (bl, br, pres) = split (ax, b) in
            assert sup (ax, elements bl) and inf (ax, elements br) in
            join (union (al, bl), ax, union (ar, br))
        else if ha = 1 then add (ax, b)
          else
            let (al, ar, pres) = split (bx, a) in
            assert sup (bx, elements al) and inf (bx, elements ar) in
            join (union (al, bl), bx, union (ar, br))
```

AVL trees from "Pangolin" (Régis-Gianas & Pottier)

A generated proof obligation

Quicksort from "Why" (Filliâtre)

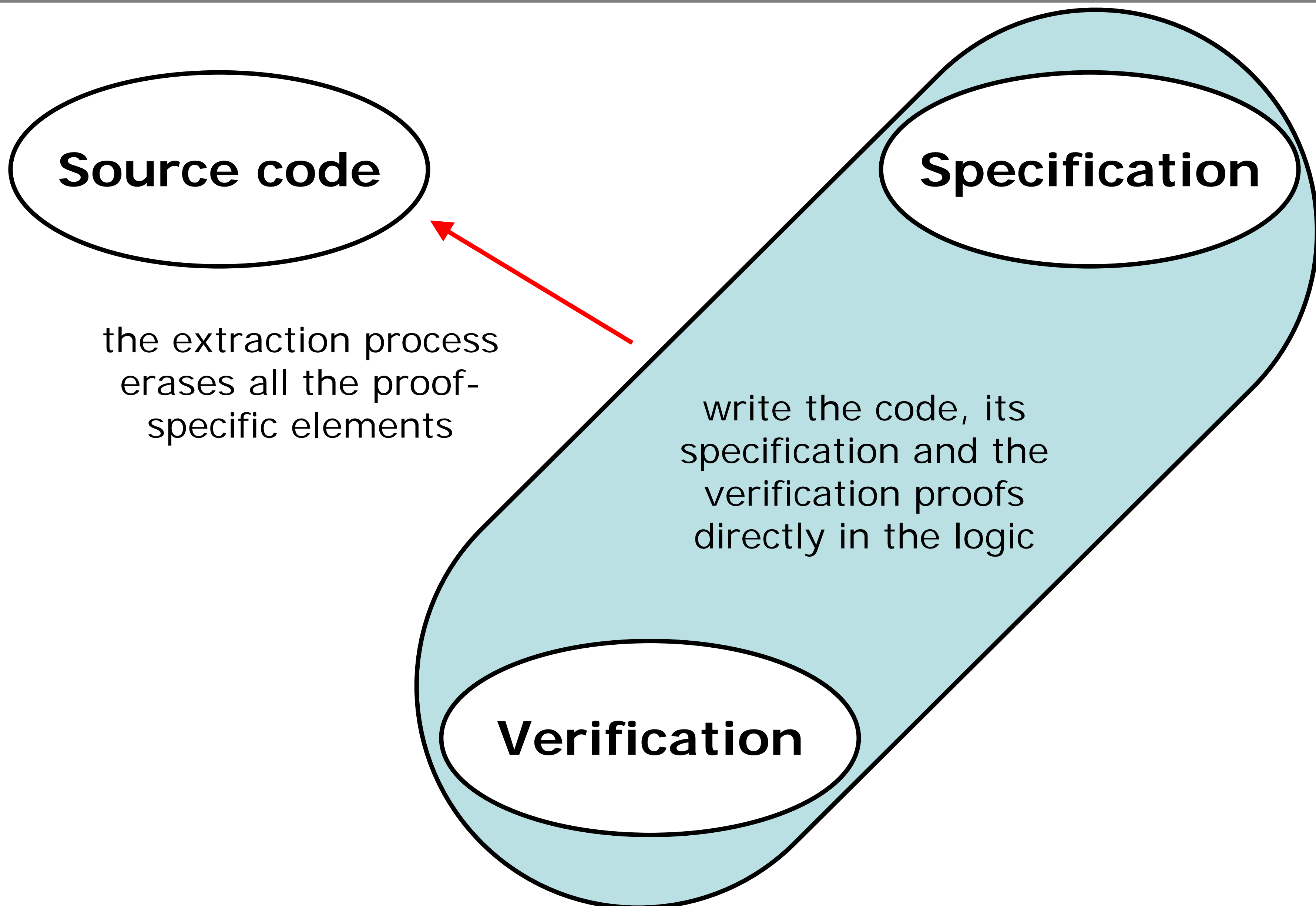
```
Lemma quicksort_rec_po_4 : forall
  (l: Z)
  (r: Z)
  (t: (array Z))
  (Pre15: `0 <= l` /\ `r < (array_length t)`)
  (Variant1: Z)
  (l0: Z)
  (r0: Z)
  (t0: (array Z))
  (Pre14: Variant1 = `1 + r0 - l0`)
  (Pre13: `0 <= l0` /\ `r0 < (array_length t0)`)
  (Test2: `l0 < r0`)
  (Pre12: (`0 <= l0` /\ `l0 < r0`)
       /\ `r0 < (array_length t0)`) (t1: (array Z))
  (p: Z)
  (Post5: (`l0 <= p` /\ `p <= r0`) /\ (partition_p t1 l0 r0 p)
       /\ (sub_permut l0 r0 t1 t0))
  (Pre11: `0 <= l0` /\ `p - 1 < (array_length t1)`)
  (t2: (array Z)),
       (sorted_array t2 l0 `p - 1`)
    /\ (sub_permut l0 `p - 1` t2 t1)) `0 <= p + 1`
    /\ `r0 < (array_length t2)`.
Proof. Intuition; SameLength t2 t1; SameLength t1 t0; Omega. Save.
```

Integrated proof language

```
private static FuncTree add(int k, Object v, FuncTree t)
/*: requires "v ~= null & (ALL y. (k,y) ~: t..content) & theinvs"
    ensures "result..content = t..content Un {(k,v)} & theinvs" */ {
   if (t==null) {
      FuncTree r = new FuncTree();
      r.data = v; r.key = k;
      r.left = null; r.right = null;
      //: "r..content" := "{(k,v)}";
      return r;
   } else {
      FuncTree new_left, new_right;
      if (k < t.key) {
         new_left = add1(k, v, t.left);
         new_right = t.right;
      } else {
         //: assert "t..key < k";
         new_left = t.left;
         new_right = add1(k, v, t.right);
      }
      FuncTree r = new FuncTree();
      r.data = t.data; r.key = t.key;
      r.left = new_left; r.right = new_right;
      //: "r..content" := "t..content Un {(k,v)}";
      return r; } }
```

Binary search trees from "Jahob" (Kuncak *et al*)

2) Extraction-based approaches



Programming in Coq

```
Definition transf_instr (f:function) (pc:node) (instr:instruct) :=
  match instr with
  | Icall sig ros args res s =>
      if is_return niter f s res
        && Conventions.tailcall_is_possible sig
        && opt_typ_eq sig.sig_res f.fn_sig.sig_res
      then Itailcall sig ros args
      else instr
  | _ => instr
  end.
```

Tail-call optimization
"Compcert" (Leroy)

```
Inductive transf_instr_spec (f:function) : instruct->instruc->Prop :=
  | transf_instr_tailcall: forall sig ros args res s,
    f.(fn_stacksize) = 0 -> is_return_spec f s res ->
    transf_instr_spec f (Icall sig ros args res s)
                        (Itailcall sig ros args)
  | transf_instr_default: forall i,
      transf_instr_spec f i i.

Lemma transf_instr_charact : forall f pc instr,
  f.(fn_stacksize) = 0 ->
  transf_instr_spec f instr (transf_instr f pc instr).
```

Dependently-typed code

```
Program Definition deep_L (A : Type) (measure : A -> v)
 (d : option (digit A)) (s : v)
 (mid : fingertree (node_measure measure) s)
 (sf : digit A)
 : fingertree measure (option_digit_measure
      measure d cdot s cdot digit_measure measure sf) :=
  match d with
    | Some pr => Deep pr mid sf
    | None =>
      match view_L mid with
        | nil_L => digit_to_tree sf
        | cons_L a sm' m' => Deep (node_to_digit a) m' sf
      end
  end.

Next Obligation.
intros.
unfold option_digit_measure; simpl.
monoid_tac; auto.
induction mid; simpl in *; monoid_tac; auto; try discriminate.
destruct l; simpl; try discriminate.
program_simpl; destruct (view_L mid); simpl; try discriminate.
Qed.
Next Obligation. ...
```

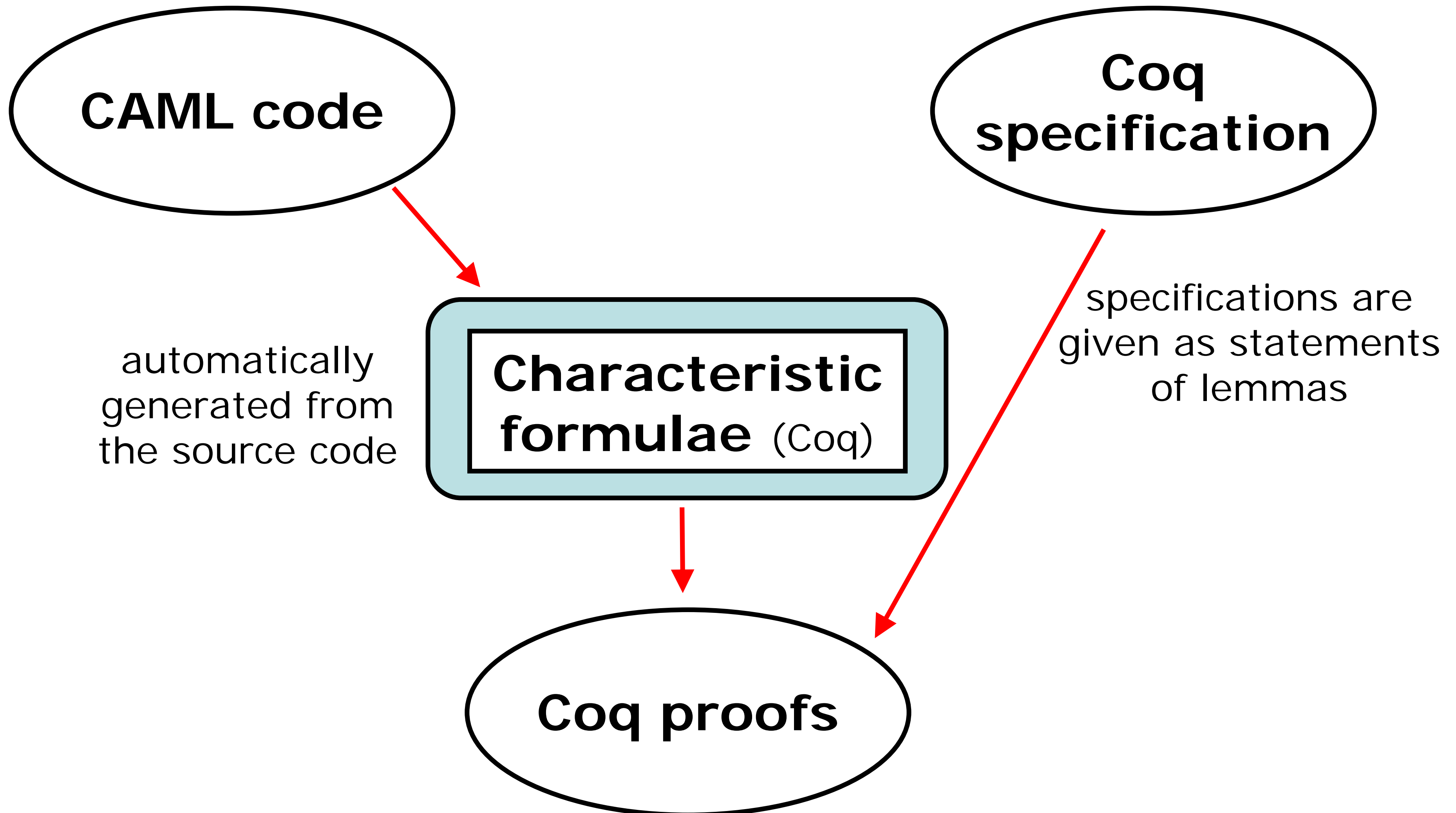
Finger trees from "Program" (Sozeau)

Coq with an IO monad

```
Program Definition enq (q : queue) (x : T) :
  STsep unit (fun i => exists xs, i \In shape q xs,
      fun y i h => forall xs, i \In shape q xs -> y = Val tt
                                /\ h \In shape q (rcons xs x)) :=
  Do (next <-- Allocb null 2;
      next ::= x;;
      ba <-- !back q;
      back q ::= next;;
      If ba == null
         then front q ::= next
         else ba .+ 1 ::= next).
Next Obligation.
apply: (ghost H) H0=>{H} xs [fr][bq][h']{H0}[D][<-] H; move: D.
hstep=>next; do !hstep; rewrite -(backfront H) un0h.
case: ifP H=>Ef; rewrite /shape' ?Ef.
- by case=>_->->; do !hstep=>D; hauto D;
apply: shape'_add_last; hauto D.
case=>[s2][x2][i2][->] D <- H2; do !hstep.
rewrite -!(push (back q)) -!(push (front q)) 4!(push (bq .+ 1))=>D1.
hauto D1; apply: shape'_add_last; hauto D1; apply: lseg_add_last.
by rewrite -!(unCA i2); hauto D1.
Qed.
Next Obligation. ...
```

Imperative queue from "HTT" (Nanevski *et al*), a new implem. of "Ynot"

3) Characteristic Formulae



Example of characteristic formula

```
let rec half x =
      if x = 0 then 0
      else if x = 1 then fail
      else let y = half (x − 2) in
            y + 1
```

The function *half* takes a nonnegative even integer and divides it by two. It is implemented recursively.

$$
\begin{array}{l}
\forall x.\, \forall P. \\
\left(\begin{array}{l}
\quad (x = 0 \;\Rightarrow\; P\, 0) \\
\wedge\; (x \neq 0 \;\Rightarrow \\
\quad\quad (x = 1 \;\Rightarrow\; \mathsf{False}) \\
\quad \wedge\; (x \neq 1 \;\Rightarrow \\
\quad\quad\quad \exists P'.\quad (\mathsf{AppReturns\,half}\,(x-2)\,P') \\
\quad\quad\quad\quad\quad \wedge\; (\forall y.\, (P'\,y) \;\Rightarrow\; P\,(y+1)) \quad))
\end{array}\right. \\
\Rightarrow\; \mathsf{AppReturns\,half}\; x\; P
\end{array}
$$

Its characteristic formula allows to specify the post-condition of applications of *half* to a value *x*.

Notation for characteristic formula

$$
\begin{array}{l}
\forall x.\, \forall P. \\
\left(\begin{array}{l}
\quad (x = 0 \;\Rightarrow\; P\,0) \\
\wedge\, (x \neq 0 \;\Rightarrow \\
\qquad (x = 1 \;\Rightarrow\; \text{False}) \\
\quad \wedge\, (x \neq 1 \;\Rightarrow \\
\qquad\quad \exists P'. \quad (\text{AppReturns half}\,(x-2)\,P') \\
\qquad\qquad\quad \wedge\; (\forall y.\, (P'\,y) \;\Rightarrow\; P\,(y+1)) \quad))
\end{array}\right. \\
\Rightarrow\; \text{AppReturns half}\; x\; P
\end{array}
$$

We set up Coq pretty-print this formula nicely, using its builtin notation system.

Characteristic formula:

$$
\begin{array}{l}
\text{LET half} := \text{Fun}\; x \mapsto \\
\qquad \text{If}\; x = 0\; \text{Then Return}\; 0 \\
\qquad \text{Else If}\; x = 1\; \text{Then Fail} \\
\qquad \text{Else Let}\; y := \text{App half}\,(x-2)\;\text{In} \\
\qquad\qquad \text{Return}\; (y+1)
\end{array}
$$

Source code:

$$
\begin{array}{l}
\text{let rec half}\; x = \\
\qquad \text{if}\; x = 0\; \text{then}\; 0 \\
\qquad \text{else if}\; x = 1\; \text{then fail} \\
\qquad \text{else let}\; y = \text{half}\,(x-2)\;\text{in} \\
\qquad\qquad y + 1
\end{array}
$$

Function specification

Informal specification: *"The function half takes a nonnegative even integer and divides it by two."*

Formal specification: (one possible formalization)

```
Lemma half_spec : forall x n,
  (x = 2 * n) -> (n >= 0) -> AppReturns half x (= n)
```

The post-condition states: *"returns a value equal to n"*.

We will use the following notation:

```
Lemma half_spec : Spec half (x:int) |R>>
  forall n, (x = 2 * n) -> (n >= 0) -> R (= n)
```

Read R as *"half applied to x returns a value such that"*.

Interactive proofs with basic tactics

Proving the program using its characteristic formula:

CoqIde (on vouvray)

File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help

half_proof.v | half.ml

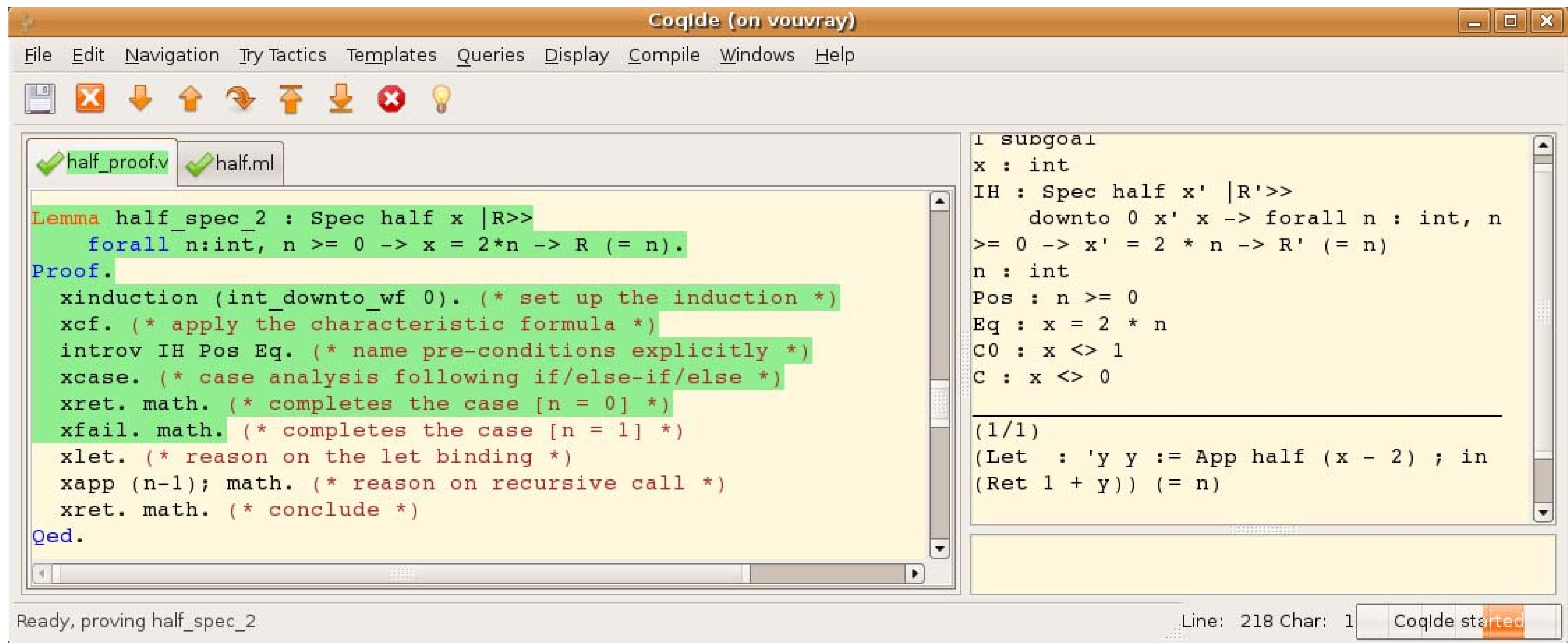
```
Lemma half_spec_0 : Spec_1 half (fun x R =>
   forall n:int, n >= 0 -> x = 2*n -> R (= n)).
Proof.
  apply (spec_induction_1 (int_downto_wf 0)).
  apply half_cf. repeat intro; auto. (* characteristic formula *)
  intros x IH n Pos Eq.
  split; intros C1; fold_bool; fold_prop. (* case analysis on n *
  hnf. math. (* prove [x = 2*n] and [x = 0] implies [n = 0] *)
  split; intros C2; fold_bool; fold_prop. (* case analysis on n *
  hnf. math. (* prove that [x = 1] is absurd since [x = 2*n] *)
  esplit. split. (* reason on the let binding *)
  (* --reasoning on the recursive call-- *)
  eapply (spec_elim_1_1' IH). (* apply the elimination lemma *)
  intros R HR. apply HR with (n:=n-1). (* prove premises *)
    unfolds. math. (* show that the measure decreases *)
    math. (* check [n-1 >= 0] *)
    math. (* check [x-2 = 2*(n-1) *)
    apply pred_le_refl. (* post condition is infered *)
  (* --reasong on the addition [_x21+1]-- *)
  intros y Hy. (* use the name [n'] in place of [_x21] *)
  hnf. hnf in Hy. math. (* concludes: [n' = n-1] implies [1+n' = :
Qed.
```

```
1 subgoal
x : int
IH : Spec half x' |R'>>
    downto 0 x' x -> forall n : int,
n >= 0 -> x' = 2 * n -> R' (= n)
n : int
Pos : n >= 0
Eq : x = 2 * n
C1 : x <> 0
C2 : x <> 1
______________________________________
(1/1)
(Let  : 'y y := App half (x - 2) ; in
(Ret 1 + y)) (= n)
```

Ready, proving half_spec_0 — Line: 160 Char: 1 — CoqIde started

With specialized tactics

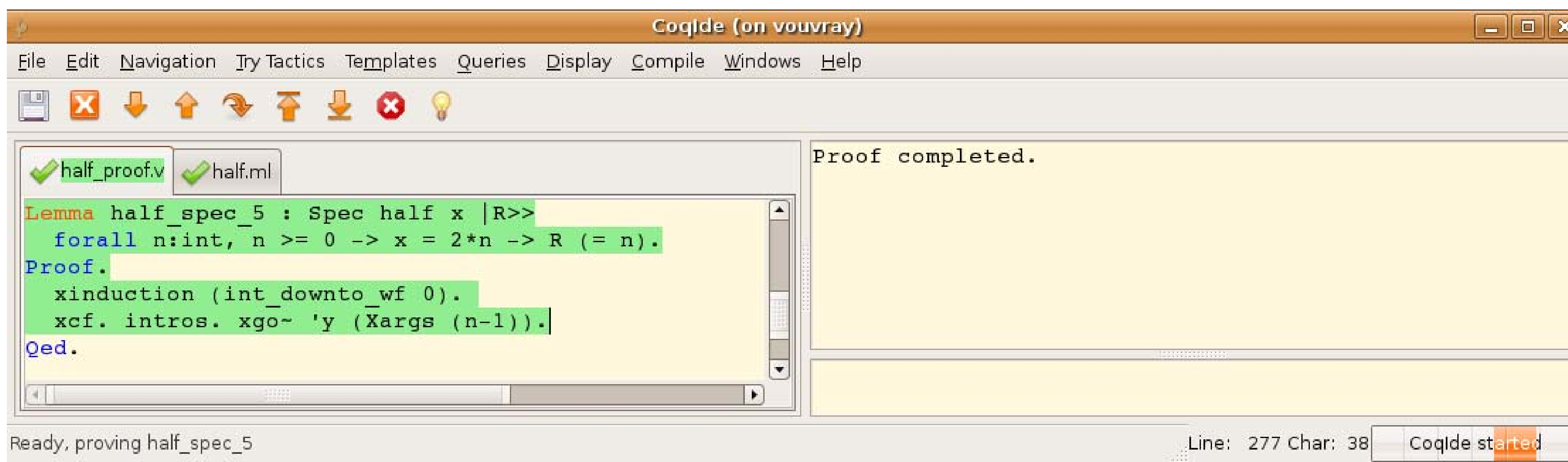
We introduce one tactic for each type of syntax node (their names always starts with the letter 'x').



Proof scripts become now much more tidy.

Automatic application of x-tactics

We define "xgo" to be a tactic that keeps applying the appropriate x-tactic on each subgoal.



xgo accepts hints to guide the instantiation of ghost variables, for reasoning on function applications.

Automation, denoted by the "~" symbol, takes care of solving all the remaining goals.

Benefits of interactive proofs

– **On simple code:** run xgo and prove obligations, exactly like you would do with a VCG.

– **On complex code:** mix Coq reasoning with manual applications of x-tactics and bounded calls to xgo.

→ A strength of my approach is the possibility to interleave VCG-style and interactive-style proofs.

Origins of Characteristic Formulae

In concurrency theory:

– called "characteristic" or "distinguishing" formulae

– expressed in some temporal logic

→ prove behavioural (dis)-equivalence of processes

Honda, Berger & Yoshida's program logic:

– called "total characteristic assertion pairs"

– expressed in an ad-hoc logic

→ verify programs by showing the most-general Hoare-triple to imply the targeted specification

Two major improvements brought by my work:

→ Build human-readable characteristic formulae

→ Implement them in a standard higher-order logic

Implementation of the generator

Parses, normalizes and type checks OCaml code, then generates Coq definitions

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

```
Inductive tree (A:Type) :=
    | Empty : tree A
    | Node : tree A -> A -> tree A -> tree A.
```

```
let x  =  let a = 3 * 12 in  Some (a, a / 2)
```

```
Axiom x : option (int * int).
Axiom x_cf : forall P, [...] -> P x.
```

```
let rec fact n = if n <= 0 then 1 else n * fact (n-1)
```

```
Axiom fact : Func.
Axiom fact_cf : forall P n, [...] -> AppReturns fact n P
```

The benchmark

Purely functional programs:

– that programmers are relatively familiar with,

– that are known to be nontrivial to verify.

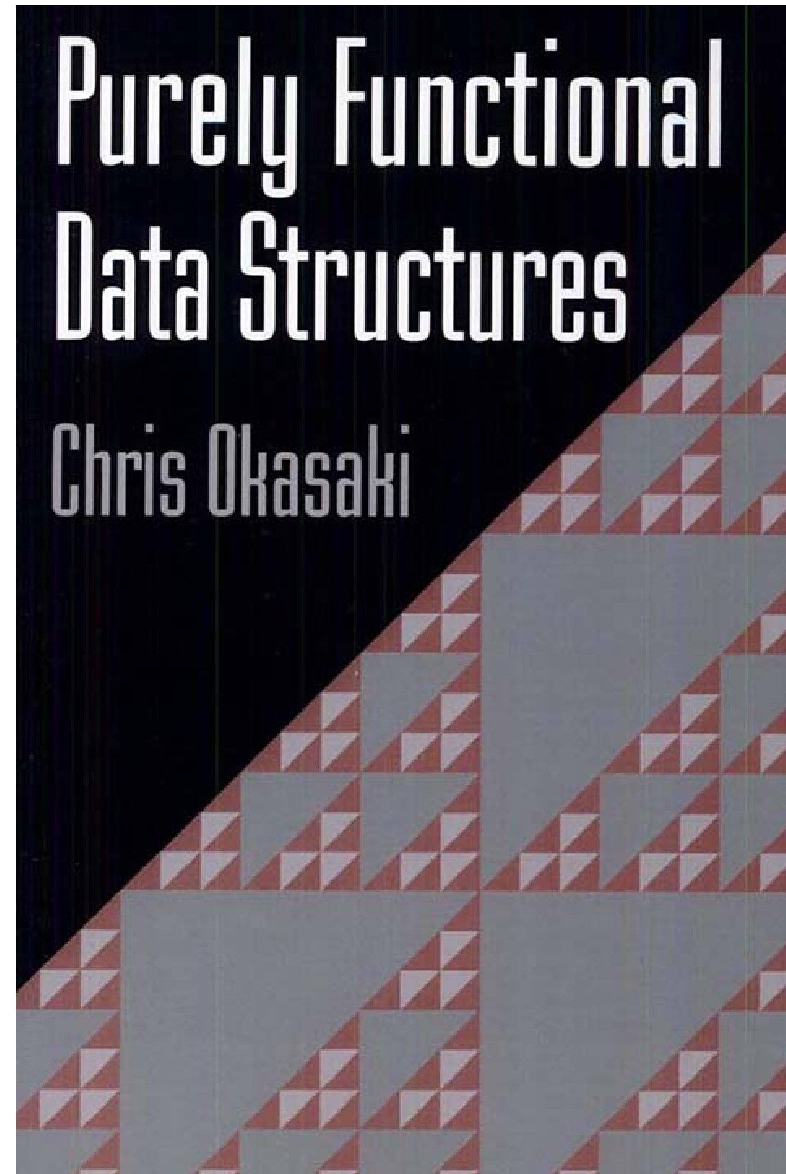
I have formalized various implementations of:

– queues and deques

– binary search trees

– random access lists

– priority queues (heaps)



Representations

Relates an implementation with its mathematical model
– a binary search tree is a representation for a set
– a heap is a representation for a multiset
– a queue is a representation for a sequence

→ If `a` and `A` are two types, `Rep a A` holds if objects of type `a` are modelled by objects of type `A`.

```
Class Rep a A :=
  { rep : a -> A -> Prop }.
```

Examples: assuming `Rep t T` holds, we have
– `Rep (tree t) (set T)`
– `Rep (heap t) (multiset T)`
– `Rep (queue t) (list T)`

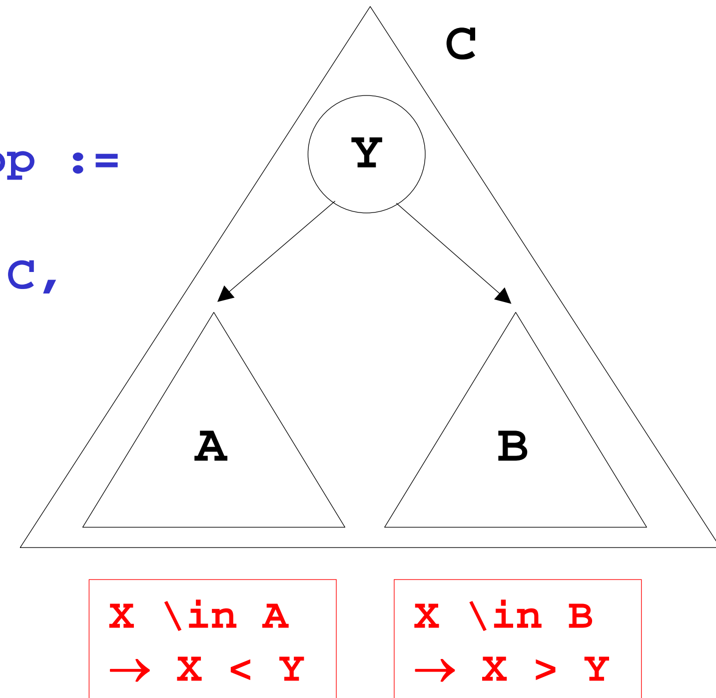
Unbalanced binary search trees

```
type 'a tree = Empty | Node of color * 'a tree * 'a * 'a tree

Instance rep : Rep t T.

Inductive inv : tree t -> set T -> Prop :=
  | inv_empty : inv Empty \{}
  | inv_node : forall col a y b A Y B C,
      inv a A ->
      inv b B ->
      rep y Y ->
      foreach (is_lt Y) A ->
      foreach (is_gt Y) B ->
      C = \{Y} \u A \u B ->
      inv (Node col a y b) C.

Instance tree_rep : Rep (tree t) (set T) := { rep := inv }.
```



Note: the whole development is parameterized by an OrderedType module containing comparison functions implementing a total ordering on the set of elements.

Specification of insert

Specification of the insert function:

```
Lemma insert_spec :
  Spec insert (x:t) (e:tree t) |R>>
    forall X E, rep x X -> rep e E ->
    R (fun e' => exists E', rep e' E' /\ E' = \{X} \u E).
```

Specification with implicit representation (RepSpec):

```
Lemma insert_spec :
  RepSpec insert (X;t) (E;tree t) |R>>
    R (fun E' => E' = \{X} \u E ;; tree t).
```

```
Lemma insert_spec :
  RepSpec insert (X;t) (E;tree t) |R>>
    R (= \{X} \u E ;; tree t).
```

```
Lemma insert_spec :
  RepTotal insert (X;t) (E;tree t) >> = \{X} \u E ;; tree t.
```

→ The specification could hardly be simpler.

Insertion for unbalanced trees

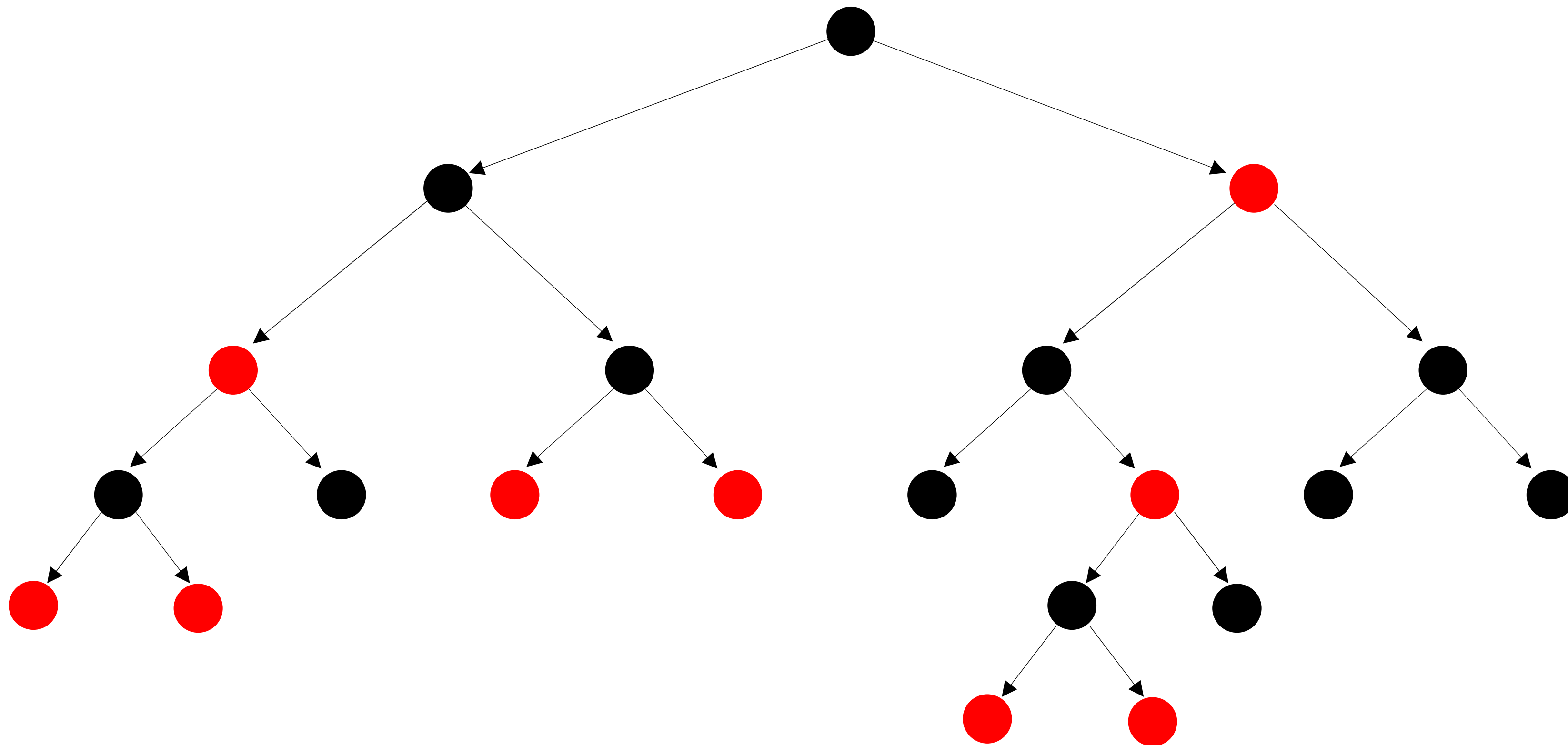
```
let rec insert x = function
    | Empty -> Node (Empty,x,Empty)
    | Node (a,y,b) as s ->
        if Element.lt x y then Node (insert x a, y, b)
        else if Element.lt y x then Node (a, y, insert x b)
        else s


Lemma insert_spec :
  RepTotal insert (X;t) (E;tree t) >> = \{X} \u E ;; tree t.
Proof.
  xinduction (fun (x:t) (e:tree t) => size e).
  xcf. intros x e IH X E RepX RepE.
  inverts RepE; [| subst E]; xgo~.
  applys~ inv_node.
  applys~ inv_node.
  applys~ inv_node.
  asserts_rewrite (X = Y). applys~ nlt_nslt_to_eq.
   subst. applys~ inv_node.
Qed.
```

Red-black trees: invariant

Red-black trees are binary search trees with nodes colored red or black.

1) Same number of black nodes on any path from a leaf to the root.
2) No red node has a red child.
3) The root must be black.



The two first invariants ensure that the tree has a logarithmic depth.

Red-black trees: invariant

Invariant "`inv n t E`" holds if the tree `t` represents the set `E` models `t` and if each path contains `n` black nodes

```
Inductive inv : nat -> tree t -> set T -> Prop :=
  | inv_empty : forall rok,
      inv 0 Empty \{}
  | inv_node : forall rok n m col a y b A Y B E,
      inv m a A -> inv m b B -> rep y Y ->
      foreach (is_lt Y) A -> foreach (is_gt Y) B ->
      (n = match col with Black => m+1 | Red => m end) ->
      (match col with | Black => True
                      | Red => node_color a = Black
                            /\ node_color b = Black end) ->
      E = (\{Y} \u A \u B) ->
      inv n (Node col a y b) E.

Instance tree_rep : Rep (tree t) (set T) := { rep :=
  fun e E => exists n, inv n e E /\ node_color e = Black }.
```

Okasaki's red-black trees

```
module RedBlackSet (Element : Ordered) : Fset = struct

 type color = Red | Black
 type tree = Empty | Node of color * tree * elem * tree

 let balance = function
 | (Black, Node (Red, Node (Red, a, x, b), y, c), z, d)
 | (Black, Node (Red, a, x, Node (Red, b, y, c)), z, d)
 | (Black, a, x, Node (Red, Node (Red, b, y, c), z, d))
 | (Black, a, x, Node (Red, b, y, Node (Red, c, z, d))) ->
    Node (Red, Node (Black, a, x, b), y, Node (Black, c, z, d))
 | (c,a,x,y) -> Node (c,a,x,y)

 let rec insert x s =
   let rec ins = function
      | Empty -> Node (Red, Empty, x, Empty)
      | Node (col, a, y, b) as s ->
        if Element.lt x y then balance (col, ins a, y, b)
        else if Element.lt y x then balance (col, a, y, ins b)
        else s in
   let Node (_, a, y, b) = ins s in Node (Black, a, y, b)
```

Red-black trees: actual invariant

Invariant "`inv i n t E`" is the same as "`inv n t E`" if `i` is true, otherwise the color constraint is relaxed.

```
Inductive inv : bool -> nat -> tree t -> set T -> Prop :=
  | inv_empty : forall i,
      inv i 0 Empty \{}
  | inv_node : forall i n m col a y b A Y B E,
      inv true m a A -> inv true m b B -> rep y Y ->
      foreach (is_lt Y) A -> foreach (is_gt Y) B ->
      (n = match col with Black => m+1 | Red => m end) ->
      (if i then match col with
                 | Black => True
                 | Red => node_color a = Black
                       /\ node_color b = Black end
            else True) ->
      E = (\{Y} \u A \u B) ->
      inv i n (Node col a y b) E.

Instance rep : Rep (tree t) (set T) := { rep := fun e E =>
   exists n, inv true n e E /\ node_color e = Black }.
```

Red-black trees: insertion

```
Lemma insert_spec : RepTotal insert (X;elem) (E;set) >>
  = \{X} \u E ;; set.
Proof.
xcf. introv RepX (n&InvE&HeB).
xfun_induction_nointro (ins_spec X) size.
  clears s n E. intros e IH n E InvE. inverts InvE as.
  xgo*. simpl. constructors~.
  introv InvA InvB RepY GtY LtY Col Num. xgo~.
  (* case insert right *)
  destruct~ col; destruct (node_color a); tryifalse; auto.
  ximpl as e. simpl. applys_eq~ Hx 1 3.
  (* case insert left *)
  destruct~ col; destruct (node_color b); tryifalse; auto.
  ximpl as e. simpl. applys_eq~ Hx 1 3.
  (* case no insertion *)
  asserts_rewrite~ (X = Y). apply~ nlt_nslt_to_eq.
  subst s. simpl. destruct col; constructors~.
xlet. xapp~. inverts P_x5; xgo. fset_inv. exists~ __.
Qed.
```

Red-black trees: insertion goal

```
(x : elem) (X : OS.T) (RepX : rep x X) (ins : val) (n : nat)
(col : color) (a : set) (y : elem) (b : set) (A : LibSet.set T) (Y : T)
(B : LibSet.set T) (m : nat) (_x1 : bool) (_x4 : tree) (s : tree)
InvA : inv true m a A
InvB : inv true m b B
RepY : rep y Y
GtY : foreach (is_lt Y) A
LtY : foreach (is_gt Y) B
Col : match col with
      | Red => node_color a = Black /\ node_color b = Black
      | Black => True end
Num : n = match col with Red => m | Black => S m end
Es : s = Node col a y b
P_x1 : X < Y
P_x4 : inv (match node_color a with Red => false Black => true end)
           m _x4 ('{X} \u A)
__________________________________________________________________
(App balance (col, _x4, y, b) ;)
  (fun e' : set => inv
      (match node_color (Node col a y b) with
       | Red => false
       | Black => true
      end) n e' ('{X} \u '{Y} \u A \u B))
```

Red-black trees: balance

```
Lemma balance_spec :
  Spec balance (p : color * tree t * t * tree t) |R>>
  let '(col,e1,x,e2) := p in
  forall i1 i2 m E1 E2 X,
  rep x X ->
  inv i1 m e1 E1 ->
  inv i2 m e2 E2 ->
  foreach (is_lt X) E1 ->
  foreach (is_gt X) E2 ->
  match col with Black => i1 \/ i2 | Red => i1 /\ i2 end ->
  R (fun e =>
    inv (match col with Black => true | Red => false end)
        (match col with Black => m+1 | Red => m end)
        e (\{X} \u E1 \u E2)).
```

Red-black trees: balance

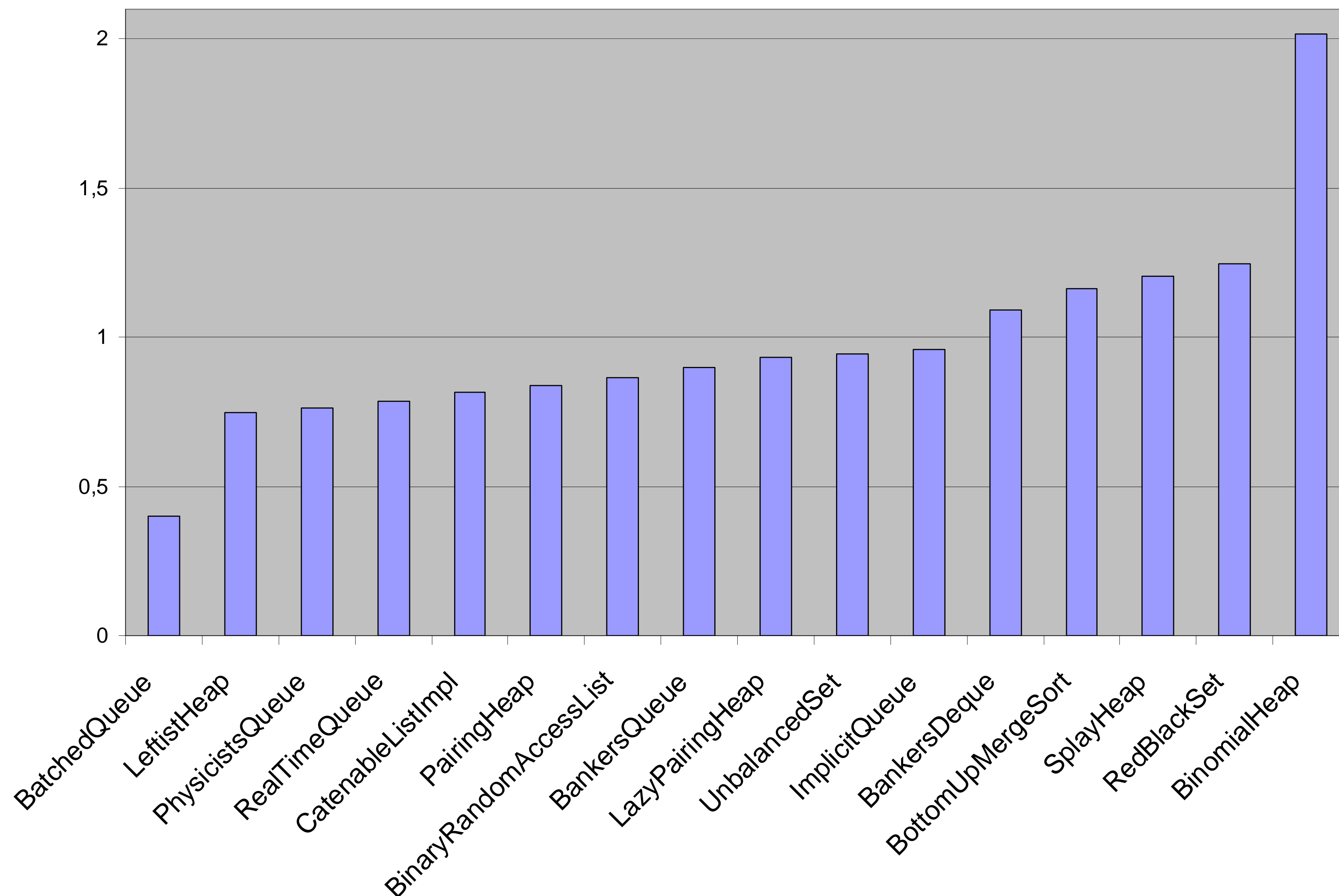
```
Proof.
xcf; intros [[[c e1] x] e2]. xisspec.
introv RepX I1 I2 GtX LtX VI. xgo.
(* rebalance 1 *)
xcleanpat. inverts I1 as IA IB. inverts IA. subst.
destruct i1; destruct VI; tryifalse.
substb i2. my_intuit. apply~ inv_node.
(* rebalance 2 *) ....
(* rebalance 3 *) ....
(* rebalance 4 *) ....
(* no rebalance *)
destruct c.
  destruct VI. substb i1. substb~ i2. destruct i1; destruct i2.
    constructors~.
    inverts keep I2; auto~. constructors~. apply~ inv_strengthen.
      destruct~ col. split.
        destruct~ a. destruct~ c. false~ C1.
        destruct~ b. destruct~ c. false~ C2.
    inverts keep I1; auto~. constructors~. apply~ inv_strengthen.
      destruct~ col. split.
        destruct~ a. destruct~ c. false~ C.
        destruct~ b. destruct~ c. false~ C0.
    destruct VI; false.
Qed.
```

Statistics

dvpt	ml	real_ml	coq	real_coq	inv	facts	spec	verif	time
BatchedQueue	30	20	104	73	4	0	16	16	3
BankersQueue	30	19	131	95	6	20	15	16	7
PhysicistsQueue	39	28	145	109	8	10	19	32	11
RealTimeQueue	37	26	146	104	4	12	21	28	6
ImplicitQueue	44	35	191	149	25	21	14	50	21
BankersDeque	61	46	223	172	7	26	24	58	27
LeftistHeap	48	36	182	132	16	28	15	22	12
PairingHeap	45	33	182	137	13	17	16	35	6
LazyPairingHeap	47	34	182	132	12	24	14	32	11
BinomialHeap	65	48	447	367	24	118	41	110	25
SplayHeap	66	53	241	176	10	41	20	59	26
UnbalancedSet	27	21	119	85	9	11	5	22	7
RedBlackSet	42	35	237	183	20	43	22	53	34
BottomUpMergeSort	41	29	194	151	23	31	9	40	10
BinaryRandomAccessList	80	63	337	271	29	37	47	83	24
CatenableListImpl	51	38	207	153	9	20	23	37	9
Total	**753**	**564**	**3268**	**2489**	**219**	**459**	**321**	**693**	**4 min**

– "Real" means blanks and comments are not counted
– "spec" includes specification of local functions
– "time" measures compilation time in seconds (only one core used)

(facts+verif) / (code+inv+spec)



What you need to do anyway: code, invariant and specification
Extra effort needed for a formal proof: lemmas and verification

Implementation

Implementation of my verification tool:

– Charact. formula generator: 3.000 lines of OCaml
  (not including OCaml's parser and typechecker)

– Coq lemmas, tactics and notation: 4.000 lines of Coq

Verification of Okasaki's book:

– OCaml code for all of the book: 1.700 lines of OCaml
  (among which 825 lines have been verified)

– Verification of half of the book: 5.000 lines of Coq
  (including specification of module types)

Relies on my own version of Coq standard library:

– Coq developed in several years: 30.000 lines of Coq

Note: figures include blank lines and comments

Summary

Interactive proofs in higher-order logic

– Higher-order logic can handle all your invariants
– Instantaneous feedback saves a lot of time
– Automated proof search is also available from Coq

Benefits over proof-obligation generation:

– xgo can almost simulate the behaviour of a VCG
– xgo can be parameterized with hints in Jahob's style
– xgo can be told to run until a given point in the code

Benefits over dependent types:

– Avoids technicalities of coding with dependent types
– The code you write is the code you compile and run
– Supports the verification of existing code

Thanks!

Further information in the draft paper:

Program Verification Through Characteristic Formulae

Arthur Charguéraud

http://arthur.chargueraud.org/research/2010/cfml/