Program Verification Through Characteristic Formulae

Arthur Charguéraud

INRIA

ICFP'10 Baltimore, 2010/10/30

The big picture

Source code (un-annotated)

Automatic generation

Characteristic formulae

Interactive proofs

Specification & verification

In this talk:

only pure programs

CF generalized to imperative programs since submission.

CF for pure programs

Total correctness judgment for pure programs:
the term t terminates and returns a value satisfying P

$$t \Downarrow P$$

Characteristic formula, written $[\![t]\!]$, such that:

$$[\![t]\!]\, P \iff t \Downarrow P$$

higher-order logic predicate

($\wedge$, $\Rightarrow$, $\forall$, $\exists$, ...)

program syntax

Note: characteristic formula of type (T $\rightarrow$ Prop) $\rightarrow$ Prop

CF: a new practical approach

- **Not a deep embedding**

  → Formulae do not refer to program syntax

- **Not a shallow embedding**

  → Caml lists are described as Coq lists

  → Caml functions are not described as Coq functions

  → Functions are described by an abstract type "Func" and specified using a predicate called "AppReturns"

- **Related to Berger, Honda and Yoshida's TCAPs**

  → Sound and complete description of programs

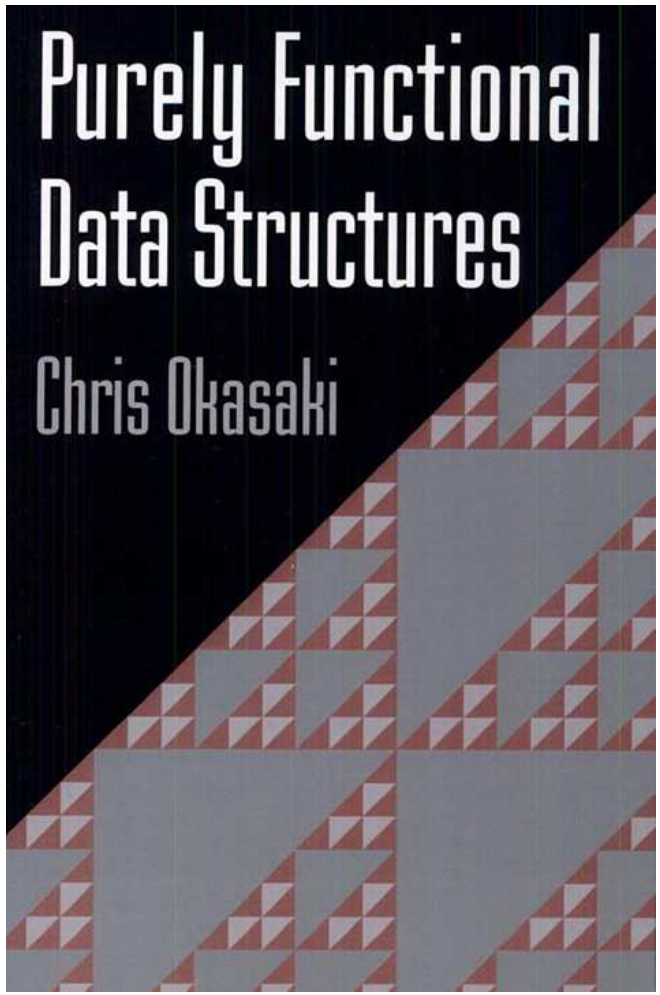
  → Difference: CF are expressed in a standard logic, making it possible to reuse existing proof tools

In this talk

1) How to build characteristic formulae

2) How to use characteristic formulae

$$
\begin{aligned}
&[\![v]\!] \;\equiv\\
&\quad \lambda P.\; P\,v\\
&[\![f\;v]\!] \;\equiv\\
&\quad \lambda P.\; \mathsf{AppReturns}\; f\; v\; P\\
&[\![\mathsf{crash}]\!] \;\equiv\\
&\quad \lambda P.\; \mathsf{False}\\
&[\![\mathsf{if}\; x\; \mathsf{then}\; t_1\; \mathsf{else}\; t_2]\!] \;\equiv\\
&\quad \lambda P.\; (x = \mathsf{true} \;\Rightarrow\; [\![t_1]\!]\, P) \;\wedge\; (x = \mathsf{false} \;\Rightarrow\; [\![t_2]\!]\, P)\\
&[\![\mathsf{let}\; x = t_1\; \mathsf{in}\; t_2]\!] \;\equiv\\
&\quad \lambda P.\; \exists P'.\; [\![t_1]\!]\, P' \;\wedge\; (\forall x.\; P'\, x \;\Rightarrow\; [\![t_2]\!]\, P)\\
&[\![\mathsf{let\ rec}\; f\; x = t_1\; \mathsf{in}\; t_2]\!] \;\equiv\\
&\quad \lambda P.\; \forall f.\; (\forall x P'.\; [\![t_1]\!]\, P' \;\Rightarrow\; \mathsf{AppReturns}\; f\; x\; P') \;\Rightarrow\; [\![t_2]\!]\, P
\end{aligned}
$$



CF for a let-binding

Reasoning rule:

$$\frac{t_1 \Downarrow P' \qquad (\forall x.\ P'\,x \ \Rightarrow\ t_2 \Downarrow P)}{(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2) \Downarrow P}$$

Characteristic formula:

$$[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!] \ \equiv$$
$$\lambda P.\ \exists P'.\ [\![t_1]\!]\,P' \ \wedge\ (\forall x.\ P'\,x \ \Rightarrow\ [\![t_2]\!]\,P)$$

Introduction of notation:

$$(\mathbf{let}\,x \ =\ \mathcal{F}_1\ \mathbf{in}\ \mathcal{F}_2) \ \equiv$$
$$\lambda P.\ \exists P'.\ \mathcal{F}_1\,P' \ \wedge\ (\forall x.\ P'\,x \ \Rightarrow\ \mathcal{F}_2\,P)$$

Characteristic formula, with notation:

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

Characteristic formula generation

A similar trick applies for other constructions:

$$
\begin{array}{lcl}
[\![v]\!] & \equiv & \mathbf{return}\ v \\
[\![f\ v]\!] & \equiv & \mathbf{app}\ f\ v \\
[\![\mathsf{crash}]\!] & \equiv & \mathbf{crash} \\
[\![\mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2]\!] & \equiv & \mathbf{if}\ v\ \mathbf{then}\ [\![t_1]\!]\ \mathbf{else}\ [\![t_2]\!] \\
[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2]\!] & \equiv & \mathbf{let}\ x = [\![t_1]\!]\ \mathbf{in}\ [\![t_2]\!] \\
[\![\mathsf{let\ rec}\ f\ x = t_1\ \mathsf{in}\ t_2]\!] & \equiv & \mathbf{let\ rec}\ f\ x = [\![t_1]\!]\ \mathbf{in}\ [\![t_2]\!]
\end{array}
$$

CF: easy to generate, compositional, easy to read

$$t \Downarrow\!| P \quad \Longleftrightarrow \quad [\![t]\!]\ P \quad \Longleftrightarrow \quad \mathbf{t}\ P$$

Note: "`f t`" is first normalized to "`let x = t in f v`"

CF for function applications

AppReturns f v P asserts that the application of the function f to v returns a value satisfying P.

Example:

$$\mathsf{AppReturns}\ \mathsf{succ}\ n\ (\lambda m.\ m = n + 1)$$

Type:

$$\mathsf{AppReturns} : \forall A\, B.\ \mathsf{Func} \to A \to (B \to \mathsf{Prop}) \to \mathsf{Prop}$$

Characteristic formulae for applications:

$$[\![f\ v]\!]\ P \ \Leftrightarrow\ \mathsf{AppReturns}\ f\ v\ P$$

$$[\![f\ v]\!] \ \equiv\ \mathsf{AppReturns}\ f\ v$$

CF for function definitions

Instances of the predicate AppReturns:

– have to be provided for reasoning on applications,

– are given by the formula of a function definition.

Instances generated for $\mathsf{let\,rec}\ f\ x = t_1$

$$\forall x P'.\ [\![t_1]\!]\ P' \Rightarrow \mathsf{AppReturns}\ f\ x\ P'$$

Characteristic formulae for functions:

$$[\![\mathsf{let\,rec}\ f\ x = t_1\ \mathsf{in}\ t_2]\!] \equiv$$
$$\lambda P.\ \forall f.\ (\forall x P'.\ [\![t_1]\!]\ P' \Rightarrow \mathsf{AppReturns}\ f\ x\ P') \Rightarrow [\![t_2]\!]\ P$$

Recursive functions are proved correct by induction.

Characteristic formula generation

$$[\![v]\!] \equiv$$
$$\lambda P.\ P\, v$$

$$[\![f\ v]\!] \equiv$$
$$\lambda P.\ \mathsf{AppReturns}\ f\ v\ P$$

$$[\![\mathsf{crash}]\!] \equiv$$
$$\lambda P.\ \mathsf{False}$$

$$[\![\mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2]\!] \equiv$$
$$\lambda P.\ (v = \mathsf{true} \Rightarrow [\![t_1]\!]\, P) \wedge (v = \mathsf{false} \Rightarrow [\![t_2]\!]\, P)$$

$$[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2]\!] \equiv$$
$$\lambda P.\ \exists P'.\ [\![t_1]\!]\, P' \wedge (\forall x.\ P'\, x \Rightarrow [\![t_2]\!]\, P)$$

$$[\![\mathsf{let\ rec}\ f\ x = t_1\ \mathsf{in}\ t_2]\!] \equiv$$
$$\lambda P.\ \forall f.\ (\forall x P'.\ [\![t_1]\!]\, P' \Rightarrow \mathsf{AppReturns}\ f\ x\ P') \Rightarrow [\![t_2]\!]\, P$$

Language supported by CFML

Supported:

– core ML language (including polymorphic recursion)
– algebraic datatypes, pattern matching
– lazy evaluation (simply ignores "lazy" keywords)
– modules, functors (partial support, Coq is limited)
– side effects, local reasoning (not in this talk)

Not supported:

– modulo arithmetics, real numbers
– catchable exceptions
– other fancy features of OCaml

Program verification through CF

Verification of purely-functional data structures, red-black trees as running example.

1) Feed the code to CFML

2) State invariant

3) State specifications

4) Conduct verification

5) Read proof obligations

6) Compute "proof/code" ratio

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)
```

Generated Coq file

```
Require Import OrderedSig_ml FsetSig_ml.

Module MLRedBlackSet (MLElement : OrderedSig_ml.MLOrdered)
 <: FsetSig_ml.MLFset.

  Inductive color : Type :=
    | Red : color
    | Black : color.

  Inductive tree : Type :=
    | Empty : tree
    | Node : color -> tree -> elem -> tree -> tree.

  ...

  Axiom insert : Func.

  Axiom insert_cf : (* shown on the next slide *).

End MLRedBlackSet.
```

Characteristic formula of "insert"

```
Axiom insert_cf : (FuncPrint.tag tag_topfun None (FuncPrint.tag tag_body None (forall K : (elem -> (tree -> ((hprop -> ((_ ->
hprop) -> Prop)) -> Prop))), ((is_spec_2 K) -> ((forall x : elem, (forall s : tree, (((K x) s) (FuncPrint.tag tag_letfun
(Some 'ins) (fun H : hprop => (fun Q : (_ -> hprop) => (forall ins : val, (Logic.ex (fun Pins : (val -> Prop) => ((Logic.and
((FuncPrint.tag tag_body None (forall K : (tree -> ((hprop -> ((_ -> hprop) -> Prop)) -> Prop)), ((is_spec_1 K) -> ((forall
_x0 : tree, ((K _x0) (FuncPrint.tag (tag_match 2%nat) (Some '_m1) (FuncPrint.tag tag_case None (fun H : hprop => (fun Q : (_
-> hprop) => ((Logic.and (((Logic.eq _x0) Empty) -> (((FuncPrint.tag tag_ret None (fun H : hprop => (fun Q : (_ -> hprop) =>
((Logic.and (H heap_empty)) ((Q ((((Node Red) Empty) x) Empty)) heap_empty))))) H) Q))) ((Logic.not ((Logic.eq _x0) Empty)) -
> (((FuncPrint.tag tag_case None (fun H : hprop => (fun Q : (_ -> hprop) => ((Logic.and (forall col : color, (forall a :
tree, (forall y : elem, (forall b : tree, (((Logic.eq _x0) ((((Node col) a) y) b)) -> (((FuncPrint.tag tag_alias None (fun H
: hprop => (fun Q : (_ -> hprop) => (forall s : tree, (((Logic.eq s) ((((Node col) a) y) b)) -> (((FuncPrint.tag tag_let
(Some '_x1) (fun H : hprop => (fun Q : (_ -> hprop) => (Logic.ex (fun Q1 : (bool -> hprop) => ((Logic.and (((FuncPrint.tag
tag_apply None ((((((@app_2 elem) elem) bool) MLElement.lt) x) y)) H) Q1)) (forall _x1 : bool, (((FuncPrint.tag tag_if None
(fun H : hprop => (fun Q : (_ -> hprop) => ((Logic.and (((Logic.eq _x1) true) -> (((FuncPrint.tag tag_let (Some '_x4) (fun H
: hprop => (fun Q : (_ -> hprop) => (Logic.ex (fun Q1 : (tree -> hprop) => ((Logic.and (((FuncPrint.tag tag_apply None
((((@app_1 tree) tree) ins) a)) H) Q1)) (forall _x4 : tree, (((FuncPrint.tag tag_apply None ((((@app_1 ((prod ((prod ((prod
color) tree)) elem)) tree)) tree) balance) (col,_x4,y,b))) (Q1 _x4)) Q)))))))) H) Q))) (((Logic.eq _x1) false) ->
(((FuncPrint.tag tag_let (Some '_x2) (fun H : hprop => (fun Q : (_ -> hprop) => (Logic.ex (fun Q1 : (bool -> hprop) =>
((Logic.and (((FuncPrint.tag tag_apply None ((((((@app_2 elem) elem) bool) MLElement.lt) y) x)) H) Q1)) (forall _x2 : bool,
(((FuncPrint.tag tag_if None (fun H : hprop => (fun Q : (_ -> hprop) => ((Logic.and (((Logic.eq _x2) true) ->
(((FuncPrint.tag tag_let (Some '_x3) (fun H : hprop => (fun Q : (_ -> hprop) => (Logic.ex (fun Q1 : (tree -> hprop) =>
((Logic.and (((FuncPrint.tag tag_apply None ((((@app_1 tree) tree) ins) b)) H) Q1)) (forall _x3 : tree, (((FuncPrint.tag
tag_apply None ((((@app_1 ((prod ((prod ((prod color) tree)) elem)) tree)) tree) balance) (col,a,y,_x3))) (Q1 _x3)) Q)))))))
H) Q))) (((Logic.eq _x2) false) -> (((FuncPrint.tag tag_ret None (fun H : hprop => (fun Q : (_ -> hprop) =>
heap_empty)) ((Q s) heap_empty))))) H) Q)))))) (Q1 _x2)) Q))))))))) H) Q))))) (((forall col : color, (forall a : tree, (forall y : elem, (forall
b))))))) -> (((FuncPrint.tag tag_done None (fun H : hprop => (fun Q : (_ -> hprop) => ((Logic.and ((is_spec_1 K)
((spec_1 K) ins))))) -> (Pins ins))) ((Pins ins) -> (((FuncPrint.tag tag_ret None (fun H : hprop => (fun Q : (_ ->
hprop) => (Logic.ex (fun Q1 : (tree -> hprop) => ((Logic.and (((FuncPrint.tag tag_apply None ((((@app_1 tree) tree) ins) b))
H) Q1)) (forall _x5 : tree, (((FuncPrint.tag (tag_match 2%nat) (Some '_m2) (FuncPrint.tag tag_case None (fun H : hprop =>
(fun Q : (_ -> hprop) => ((Logic.and (((Logic.eq _x5) Empty) -> (((FuncPrint.tag tag_ret None (fun H : hprop => (fun Q : (_
-> hprop) => False))) H) Q))) ((Logic.not ((Logic.eq _x5) Empty)) -> (((FuncPrint.tag tag_ret None (fun H : hprop => (fun Q
: (_ -> hprop) => ((Logic.and (forall _p0 : color, (forall a : tree, (forall y : elem, (forall b : tree,
((((Node _p0) a) y) b)) -> (((FuncPrint.tag tag_ret None (fun H : hprop => (fun Q : (_ -> hprop) => ((Logic.and (H
heap_empty)) ((Q ((((Node Black) a) y) b)) heap_empty))))) H) Q))) ((Logic.not ((Logic.eq _x5) ((((Node _p0) a) y) b)))
elem, (forall b : tree, (Logic.not ((Logic.eq _x5) ((((Node _p0) a) y) b)))) -> (((FuncPrint.tag tag_ret None (fun H :
hprop => (fun Q : (_ -> hprop) => True))) H) Q)))))) H) Q))))) (((FuncPrint.tag tag_ret None (fun H : hprop => (fun Q :
insert))))))).
```

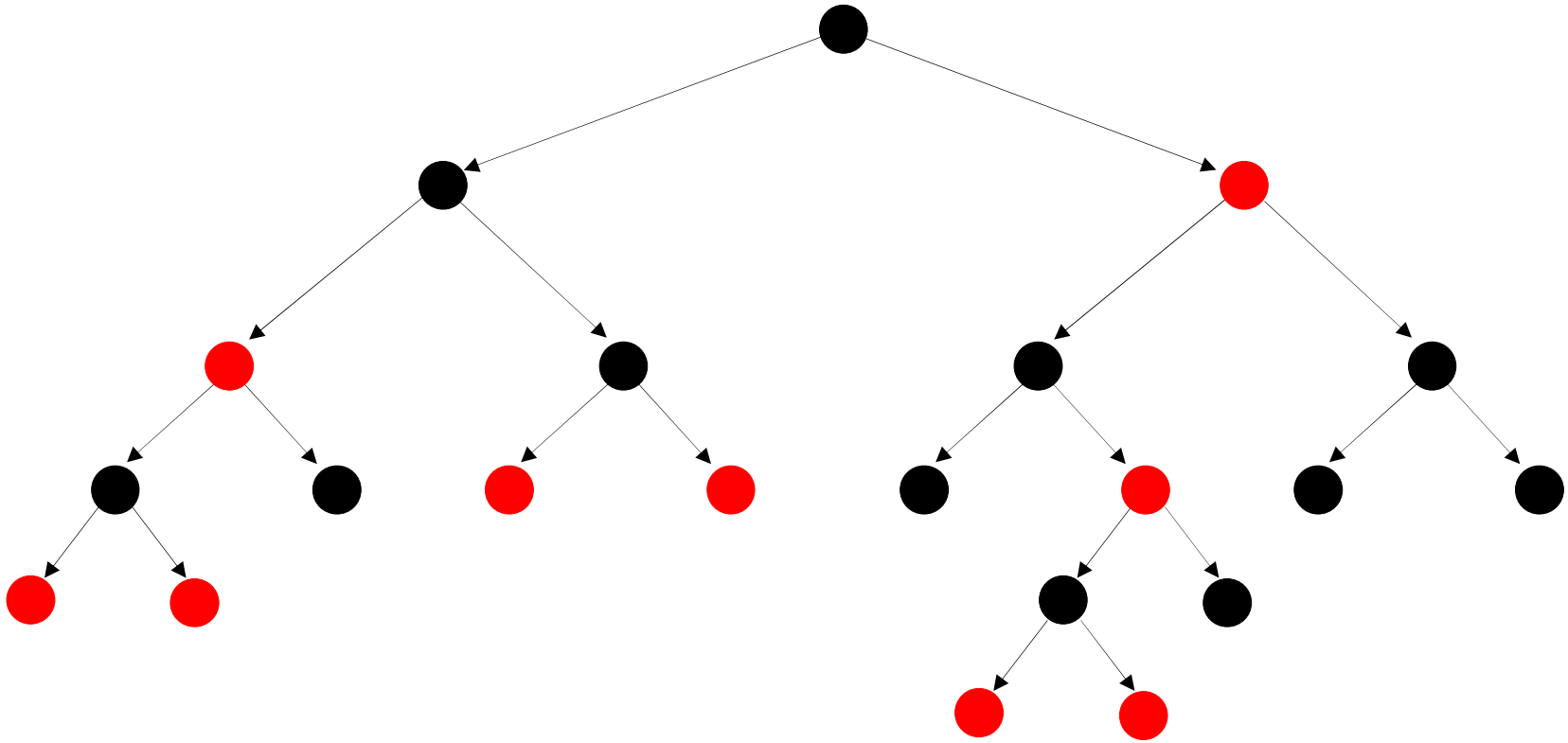
Pretty-printed version

Check insert_cf.

```
TopLetFun insert x s =>
(LetFun ins _x0 => (Match 2
   (Case (_x0 = Empty) Then (Ret Node Red Empty x Empty) Else
   (Case _x0 = Node col a y b [col a y b] Then
     Alias s0 := Node col a y b in
     (Let _x1 := App MLElement.lt x y; in
     (_If _x1
        Then Let _x4 := App ins a; in App balance (col, _x4, y, b);
      Else (Let _x2 := App MLElement.lt y x; in (_If _x2
        Then Let _x3 := App ins b; in App balance (col, a, y, _x3);
      Else (Ret s0))))))
   Else Done))) in
(Let _x5 := App ins s; in
(Match 2
  (Case _x5 = Empty Then Crash Else
  (Case _x5 = Node _p0 a y b [_p0 a y b] Then Ret Node Black a y b
  Else Done))))).
```

Future work: a Coq plugin for even nicer pretty-printing

Invariants of red-black trees



Binary search tree where:

1) Same number of black nodes in every path
2) No red node has a red child
3) Root is always black

Representation predicates

Representation predicate: used to relate a value with the corresponding mathematical structure.

```
rep (Node Black Empty 3 Empty) \{3}
```

Type-class definition: to associate a datastructure of type `a` with a mathematical structure of type `A`.

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

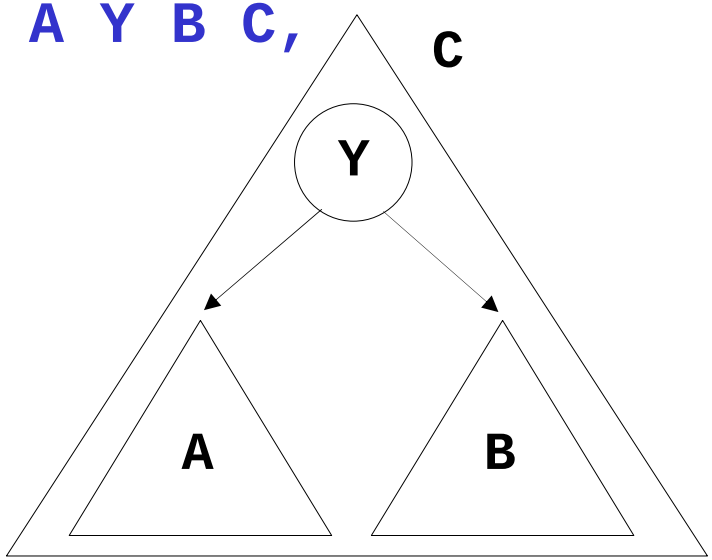
For red-black trees:

```
Instance t_rep : Rep t T.
Instance t_le : Le T.
Instance t_le_order : Order t_le.
Instance tree_rep : Rep (tree t) (set T)
  := (* defined on the next slide *).
```

Invariant for unbalanced trees

Specification of binary search trees:

```
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 }.
```

"rep e E" enforces the invariant on the tree and also gives the set of elements that it contains.

Invariant for red-black trees

The invariant "**inv n t E**" holds if the tree **t** represents the set **E** and every path in **t** 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 }.
```

Specification of insertion

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).
```

Making representation predicates implicit:

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

Streamlining of the statement:

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

Verification of insertion

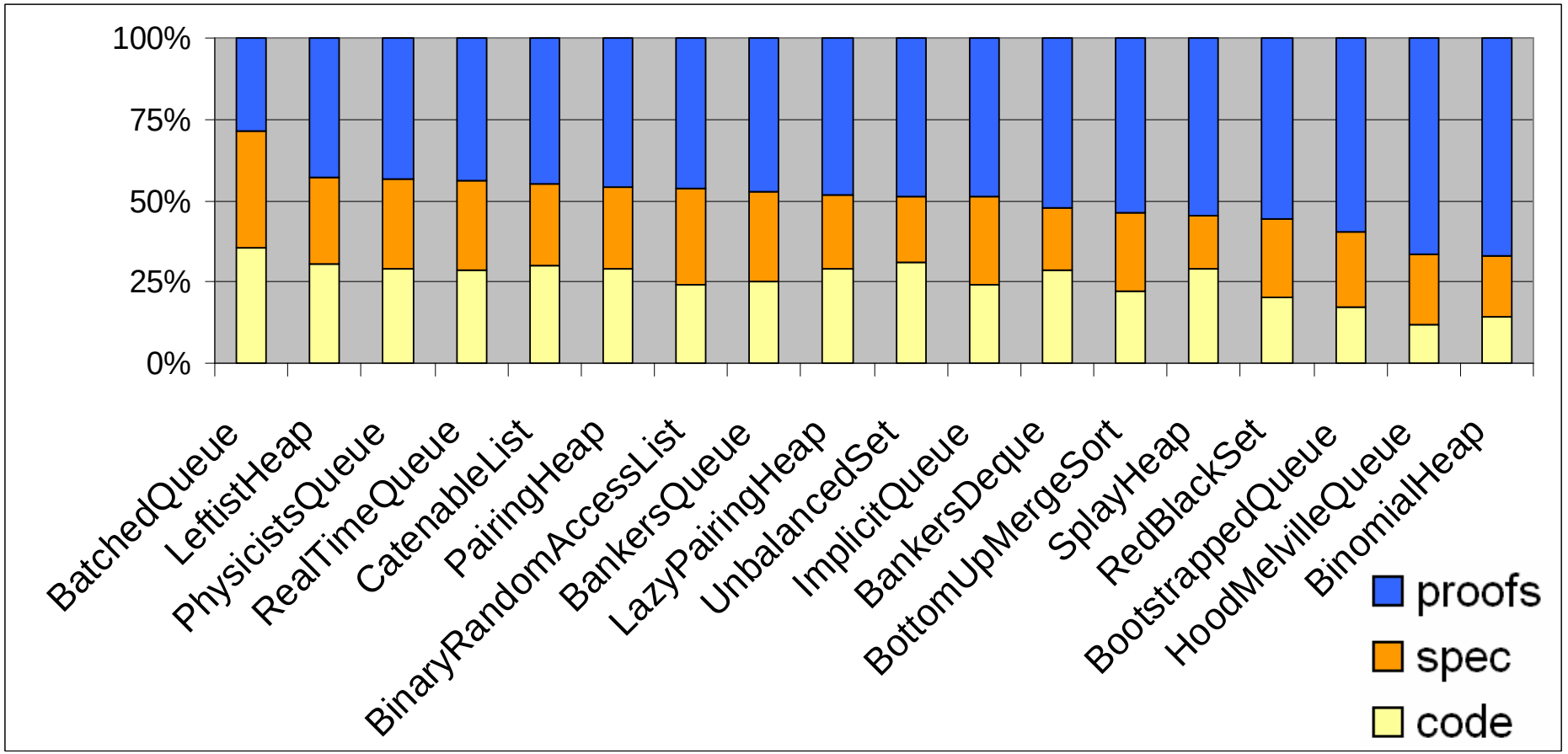
```
Lemma insert_spec : RepTotal insert (X;elem) (E;set) >>
  = \{X} \u E ;; set.
Proof.
  xcf. introv RepX (n&InvE&HeB).
  xfun_induction_nointro (ins_spec X) size.
    clears s n E. intros e IH n E InvE. inverts InvE as.
    xgo*. simpl. constructors~.
    introv InvA InvB RepY GtY LtY Col Num. xgo~.
    (* -- case insert left -- *)
    destruct~ col; destruct (node_color a); tryifalse; auto.
    ximpl as e. simpl. applys_eq~ Hx 1 3.
    (* -- case insert right -- *)
    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.
```

Proof obligation on call to "balance"

```
(x : elem) (X : OS.T) (ins : val) (n : nat) (col : color)
(a : tree) (y : elem) (b : tree) (A : set T) (Y : T)
(B : set T) (m : nat) (_a : tree) (s : tree)
InvA : inv m a A
InvB : inv m b B
RepX : rep x X
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
C1 : X < Y
P_a : inv m _a ('{X} \u A)
```

```
(App balance (col, _a, y, b);)
  (fun e' => inv n e' ('{X} \u '{Y} \u A \u B))
```

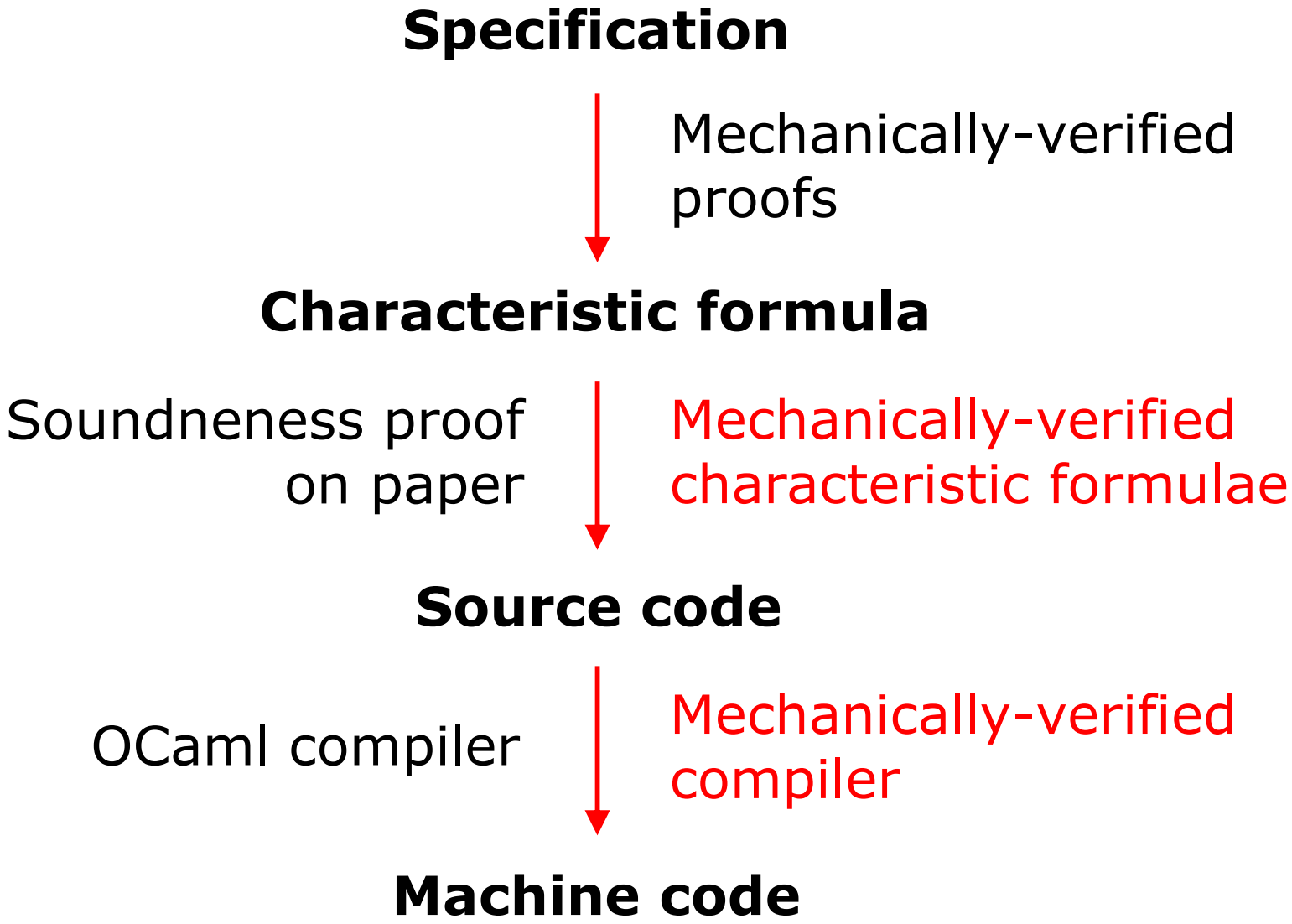
Statistics on half of Okasaki's book



Total: **564** non-blank lines of **Caml** (very concise code)

2489 non-blank lines of **Coq** (2min. to compile)

Future work: fully-verified chain



Not discussed in this talk

- **Curried n-ary functions**
- **Pattern matching**
- **Polymorphism**
- **Higher-order functions**
- **Mutable references** (Separation Logic style)
- **Null pointers and strong updates**
- **Loops** (without loop invariants, like in Tuerk's work)
- **Proofs of soundness and completeness**

Advertisement: it's all in my dissertation

Proof scripts and further information:

http://arthur.chargueraud.org/

Thanks!

Many thanks to my advisor François Pottier for giving a lot of useful feedback on this work.