

program extraction

logical verification

week 7

2004 10 20

why intuitionism?

foundational crisis

Russell, start 20th century:

$$\{x \mid x \notin x\} \in \{x \mid x \notin x\} ?$$

shows that **naive** set theory / type theory is inconsistent

Brouwer

three schools:

- **formalism**

Hilbert ... leads eventually to ZFC set theory

- **logicism**

Russell ... early version of type theory

- **intuitionism**

Brouwer rejects excluded middle, proves all functions continuous



Heyting the logic of intuitionism



Bishop variant that is strictly weaker than classical mathematics

constructivism

Brouwer-Heyting-Kolmogorov interpretation

proof of \perp ... doesn't exist

proof of $A \rightarrow B$ \leftrightarrow function that maps proofs of A to proofs of B

proof of $A \wedge B$ \leftrightarrow pair of a proof of A and a proof of B

proof of $A \vee B$ \leftrightarrow either a proof of A or a proof of B

proof of $\forall x. P(x)$ \leftrightarrow function that maps object x to proof of $P(x)$

proof of $\exists x. P(x)$ \leftrightarrow **object a** together with proof of $P(a)$

proof of existence corresponds to **constructing** an example

proofs are programs

program extraction

intuitionistic proof



executable algorithm

intuitionism the natural logic for computer science?

'code-carrying proofs'

verified programs

two approaches

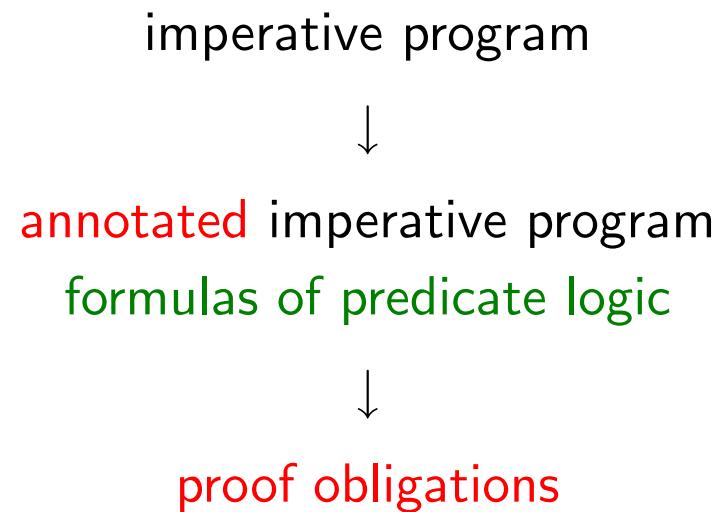
- correctness proofs

program → ... + proof

- program extraction

program ← proof

Hoare logic



why & caduceus

Jean-Christophe Filliâtre

- **why**

Hoare logic for small programming language

union of imperative and functional programming language

programming language independent

proof assistant independent

designed to be used with Coq

- **caduceus**

Hoare logic for almost full ANSI C

built on top of why

example

```
/*@ requires \valid_range(t,0,n-1)
 @ ensures
 @   (0 <= \result < n => t[\result] == v) &&
 @   (\result == n => \forall int i; 0 <= i < n => t[i] != v)
 @*/
int index(int t[], int n, int v) {
    int i = 0;
    /*@ invariant 0 <= i && \forall int k; 0 <= k < i => t[k] != v
     @ variant n - i */
    while (i < n) {
        if (t[i] == v) break;
        i++;
    }
    return i;
}
```

program extraction

specification

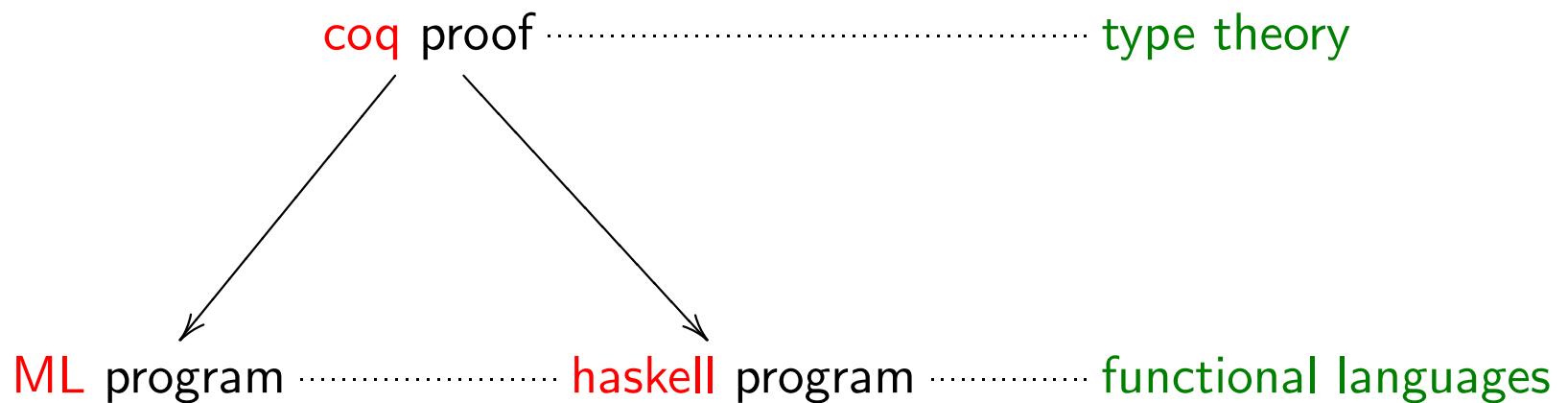


constructive **proof** of existence of solution to the specification



automatically generated **functional program**
guaranteed correct with respect to the specification

extraction to functional programs



example: mirroring trees

bintree

inductive type

```
Inductive bintree : Set :=
  leaf : nat -> bintree
| node : bintree -> bintree -> bintree.
```

mirror

recursive function

```
Fixpoint mirror (t : bintree) : bintree :=
  match t with
    leaf n => leaf n
  | node t1 t2 => node (mirror t2) (mirror t1)
  end.
```

Mirrored

inductive predicate

```
Inductive Mirrored : bintree -> bintree -> Prop :=
  Mirrored_leaf :
    forall n : nat, Mirrored (leaf n) (leaf n)
  | Mirrored_node :
    forall t1 t2 t1' t2' : bintree,
      Mirrored t1 t1' -> Mirrored t2 t2' ->
      Mirrored (node t1 t2) (node t2' t1').
```

correctness of mirror

```
Lemma Mirrored_mirror :  
  forall t : bintree, Mirrored t (mirror t).  
induction t.  
simpl.  
apply Mirrored_leaf.  
simpl.  
apply Mirrored_node.  
exact IHt1.  
exact IHt2.  
Qed.
```

two kinds of existential statements

$$\exists x : A. P(x)$$

- existential in Prop

exists x : A, P x

- existential in Set

{x : A | P x}

definition of ex

inductive type

```
Inductive ex (A : Set) (P : A -> Prop) : Prop :=
  ex_intro : forall x : A, P x -> ex P
```

in practice

exists x : A. P x

is syntax for

ex A (fun x : A => P x)

definition of sig

inductive type

```
Inductive sig (A : Set) (P : A -> Prop) : Set :=  
  exist : forall x : A, P x -> sig P
```

in practice

{x : A | P x}

is syntax for

```
sig A (fun x : A => P x)
```

existence proof for specification

Lemma **Mirror** :

forall t : bintree, {t' : bintree | Mirrored t t'}.

induction t.

exists (leaf n).

apply Mirrored_leaf.

elim IHt1.

intros t1' H1.

elim IHt2.

intros t2' H2.

exists (node t2' t1').

apply Mirrored_node.

exact H1.

exact H2.

Qed.

extracting the program

```
Coq < Extraction Mirror.  
(*> val mirror : bintree -> bintree sig0 **)  
  
let rec mirror = function  
| Leaf n -> Leaf n  
| Node (b0, b1) -> Node ((mirror b1), (mirror b0))  
  
Coq <  
  
type 'a sig0 = 'a
```

summarizing

- **specification**

```
Inductive Mirrored : bintree -> bintree -> Prop := ...
```

- **implementation**

```
Fixpoint mirror (t : bintree) : bintree := ...
```

- **correctness**

```
forall t : bintree, Mirrored t (mirror t)
```

- **program extracted from existence proof for specification**

```
forall t : bintree, {t' : bintree | Mirrored t t'}
```

the general pattern

Π_2 sentences

program specification

$$\forall x : A. P(x) \rightarrow \exists y : B. Q(x, y)$$

A input type

B output type

$P(x)$ precondition

$Q(x, y)$ input/output behavior

the proof term versus the extracted program

coq type theory = functional programming language

coq proof term = functional program

ML language = functional programming language

ML program = functional program

program extraction is **almost** the identity function

- differences in type system
- not all parts of coq terms are computationally relevant

Prop versus Set

not all coq terms are computationally relevant

'Curry-Howard-de Bruijn' terms don't need to be calculated

terms of type in Prop 'non-informative' discarded

terms of type in Set 'informative' kept

‘elimination of Prop over Set’

```
Inductive or (A : Prop) (B : Prop) : Prop :=
  or_introL : A -> A ∨ B
| or_introR : B -> A ∨ B.
```

```
Definition foo (A : Prop) (H : A ∨ ~A) : bool :=
  match H with
    or_introL _ => true
  | or_introR _ => false
  end.
```

Elimination of an inductive object of sort : ‘Prop’
is not allowed on a predicate in sort : ‘Set’
because non-informative objects may not construct informative ones.

example: negation in the booleans

statement

```
forall b : bool, {b' : bool | ~(b = b')}
```

extracted program

```
(** val negation : bool -> bool sig0 **)

let negation = function
| True -> False
| False -> True
```

proof term

```
fun b : bool =>
bool_rec (fun b0 : bool => b' : bool | b0 <> b')
  (exist (fun b' : bool => true <> b') false
    (fun H : true = false =>
      let H0 :=
        eq_ind true (fun ee : bool => if ee return Prop then True else False)
          I false H in
        False_ind False H0))
  (exist (fun b' : bool => false <> b') true
    (fun H : false = true =>
      let H0 :=
        eq_ind false
          (fun ee : bool => if ee return Prop then False else True) I true H in
        False_ind False H0)) b

bool_rec :
forall P : bool -> Set, P true -> P false -> forall b : bool, P b
```

example: the predecessor function

statement

forall n : nat, $\sim(n = 0) \rightarrow \{m : \text{nat} \mid S m = n\}$

extracted program

```
(** val pred : nat -> nat sig0 **)

let rec pred = function
  | 0 -> assert false (* absurd case *)
  | S n0 -> n0
```

the assert corresponds in the proof term to ...

```
False_rec {m : nat | S m = 0} (H (refl_equal 0))
  : {m : nat | S m = 0}
```

... recursion on a proof of False

extraction in the large

FTA project

coq formalization of non-trivial mathematical theorem

Fundamental Theorem of Algebra

every non-constant complex polynomial has a root

finished in 2000

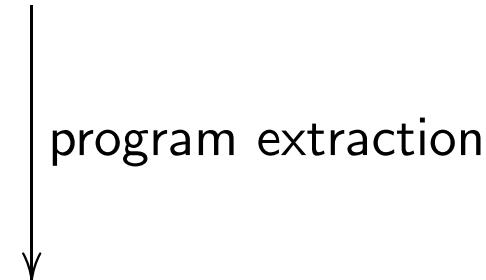
Herman Geuvers, Randy Pollack, Freek Wiedijk, Jan Zwanenburg

intuitionistic proof

extracting the Fundamental Theorem of Algebra

complex polynomials

$$\forall p. (p \text{ not constant}) \rightarrow \exists z. p(z) = 0$$



program for calculating roots of polynomials

input complex polynomial

output sequence converging to a root

extracting the Intermediate Value Theorem

real polynomials

$$\forall p. (p(0) < 0 \wedge p(1) > 0) \rightarrow \exists x. (0 < x \wedge x < 1 \wedge p(x) = 0)$$

take $p(x) = x^2 - 2$



program extraction

program for approximating $\sqrt{2}$

example: sorting lists

natlist

inductive type

```
Inductive natlist : Set :=
  nil : natlist
| cons : nat -> natlist -> natlist.
```

Sorted

inductive predicate

```
Inductive Sorted : natlist -> Prop :=
  Sorted_nil : Sorted nil
| Sorted_one : forall n : nat, Sorted (cons n nil)
| Sorted_cons :
  forall (n m : nat) (l : natlist),
  n <= m -> Sorted (cons m l) -> Sorted (cons n (cons m l)).
```

Inserted

inductive predicate

```
Inductive Inserted (n : nat) : natlist -> natlist -> Prop :=  
  Inserted_front :  
    forall l : natlist, Inserted n l (cons n l)  
  | Inserted_cons :  
    forall (m : nat) (l l' : natlist),  
    Inserted n l l' -> Inserted n (cons m l) (cons m l').
```

Inserted 4 [1,2,3] [4,1,2,3]

Inserted 4 [1,2,3] [1,4,2,3]

Inserted 4 [1,2,3] [1,2,4,3]

Inserted 4 [1,2,3] [1,2,3,4]

Permutation

inductive predicate

```
Inductive Permutation : natlist -> natlist -> Prop :=
  Permutation_nil : Permutation nil nil
| Permutation_cons :
  forall (n : nat) (l l' l'' : natlist),
  Permutation l l' -> Inserted n l' l'' ->
  Permutation (cons n l) l''.
```

statement

```
forall l : natlist,  
{l' : natlist | Permutation l l' /\ Sorted l'}
```

insert

recursive function

```
Fixpoint insert (n : nat) (l : natlist) struct l : natlist :=
  match l with
    nil => cons n nil
  | cons m k =>
    match le_lt_dec n m with
      left _ => cons n (cons m k)
      | right _ => cons m (insert n k)
    end
  end.
```

sort

recursive function

```
Fixpoint sort (l : natlist) : natlist :=
  match l with
    nil => nil
  | cons m k => insert m (sort k)
  end.
```