An Introduction to Program Verification with the Coq Proof Assistant

NII Lectures Series

Frédéric Loulergue

Université d'Orléans – LIFO – PaMDA Team

October-November 2013

Outline

1. Introduction
2. Functional programming in Coq
3. Stating and proving properties
4. Program extraction
5. Bibliography

The Coq Proof Assistant I

ACM SIGPLAN Software Award 2013

The Coq proof assistant provides a rich environment for interactive development of machine-checked formal reasoning. Coq is having a profound impact on research on programming languages and systems [. . .] It has been widely adopted as a research tool by the programming language research community [. . .] Last but not least, these successes have helped to spark a wave of widespread interest in dependent type theory, the richly expressive core logic on which Coq is based.

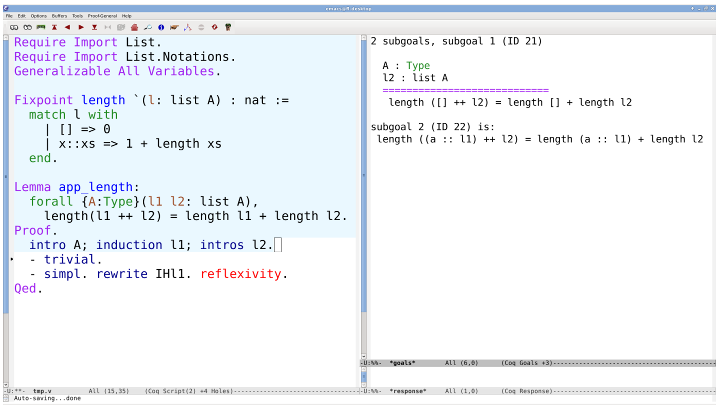
[. . .] The Coq team continues to develop the system, bringing significant improvements in expressiveness and usability with each new release.

In short, Coq is playing an essential role in our transition to a new era of formal assurance in mathematics, semantics, and program verification.

The Coq Proof Assistant II

Foundations

- Calculus of inductive constructions
- Curry-Howard correspondance



Curry-Howard Correspondance

Natural Deduction

$$(v)\frac{A \in \Gamma}{\Gamma \vdash A}$$

$$(i)\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

$$(a)\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

Simply Typed λ-Calculus

$$(V)\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$(L)\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x{:}A.e) : A \to B}$$

$$(A)\frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash (e\ e') : B}$$

Curry-Howard Correspondance

Natural Decduction – Example 1

$$(i)\frac{(i)\dfrac{(i)\dfrac{(i)\dfrac{(a)\dfrac{(v)\dfrac{A \to C \in \Gamma}{\Gamma \vdash A \to C} \qquad (v)\dfrac{A \in \Gamma}{\Gamma \vdash A}}{\Gamma \equiv A,\ B,\ A \to C,\ B \to C \vdash C}}{A,\ B,\ A \to C \vdash (B \to C) \to C}}{A,\ B \vdash (A \to C) \to (B \to C) \to C}}{A \vdash B \to (A \to C) \to (B \to C) \to C}}{\vdash A \to B \to (A \to C) \to (B \to C) \to C}$$

Curry-Howard Correspondance

Natural Decduction – Example 2

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(v)\dfrac{B \to C \in \Gamma}{\Gamma \vdash B \to C} \qquad (v)\dfrac{B \in \Gamma}{\Gamma \vdash B}}{\Gamma \equiv A,\ B,\ A \to C,\ B \to C \vdash C}(a)}{A,\ B,\ A \to C \vdash (B \to C) \to C}(i)}{A,\ B \vdash (A \to C) \to (B \to C) \to C}(i)}{A \vdash B \to (A \to C) \to (B \to C) \to C}(i)}{\vdash A \to B \to (A \to C) \to (B \to C) \to C}(i)
$$

Curry-Howard Correspondance

λ-calculus: find a term with the given type

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(V)\dfrac{f{:}A \to C \in \Gamma}{\Gamma \vdash f{:}A \to C} \qquad (V)\dfrac{x{:}A \in \Gamma}{\Gamma \vdash x{:}A}}{\Gamma \equiv x{:}A,\ y{:}B,\ f{:}A \to C,\ g{:}B \to C \vdash ? \quad : \ C}(A)}{x{:}A,\ y{:}B,\ f{:}A \to C \vdash ? \quad : \ (B \to C) \to C}(L)}{x{:}A,\ y{:}B \vdash ? \quad : \ (A \to c) \to (B \to C) \to C}(L)}{x{:}A \vdash ? \quad : \ B{\to}(A{\to}C){\to}(B{\to}C){\to}C}(L)}{\vdash ? \quad : \ A{\to}B{\to}(A{\to}C){\to}(B{\to}C){\to}C}(L)
$$

$\lambda x{:}A.\lambda y{:}B.\lambda f{:}A{\to}C.\lambda g{:}B{\to}C.(f\ x)$
is a way to encode the proof tree of
$A{\to}B{\to}(A{\to}C){\to}(B{\to}C){\to}C$

Curry-Howard Isomomorphism

For all formula there exists a proof of this formula in natural deduction if and only if there exists a λ-term that has this formula as type.

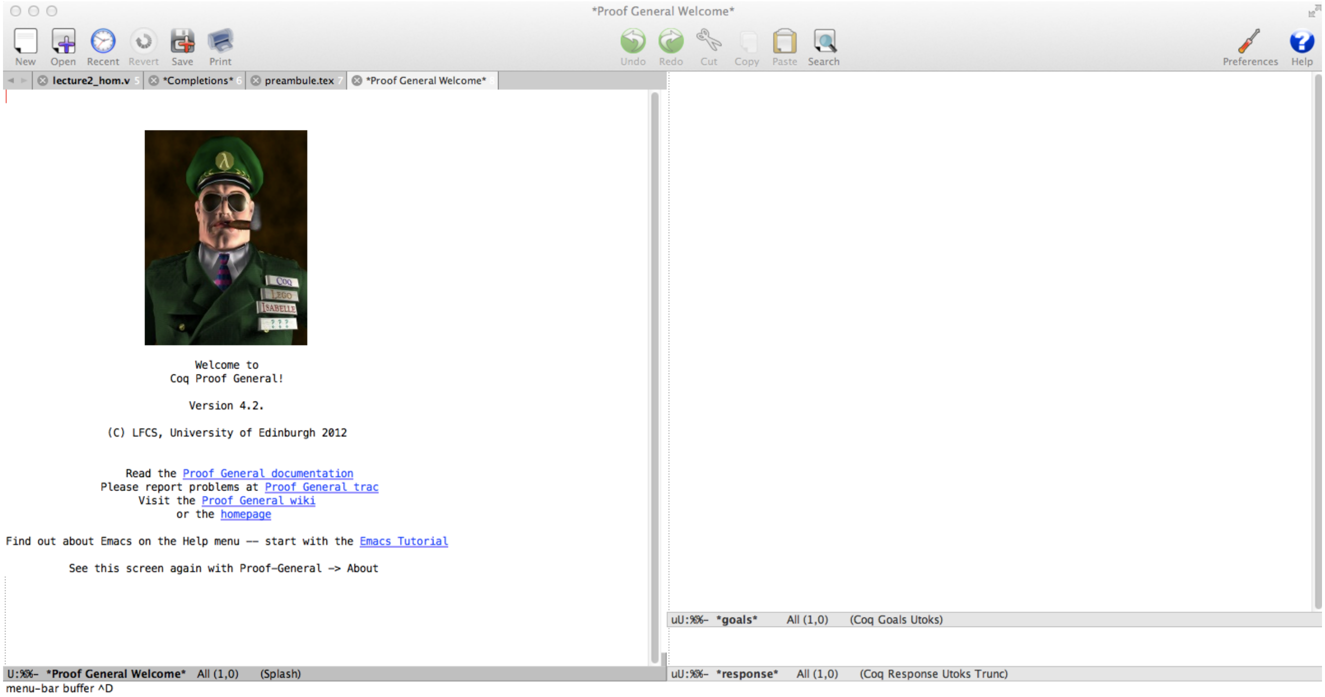
- Theorem statement $\Leftrightarrow$ Type
- Proof $\Leftrightarrow$ Program

Coq in practice

- Functional programming language
- Rich type system: allow to express logical properties
- Language for building proofs (ie proof terms)
- Program extraction

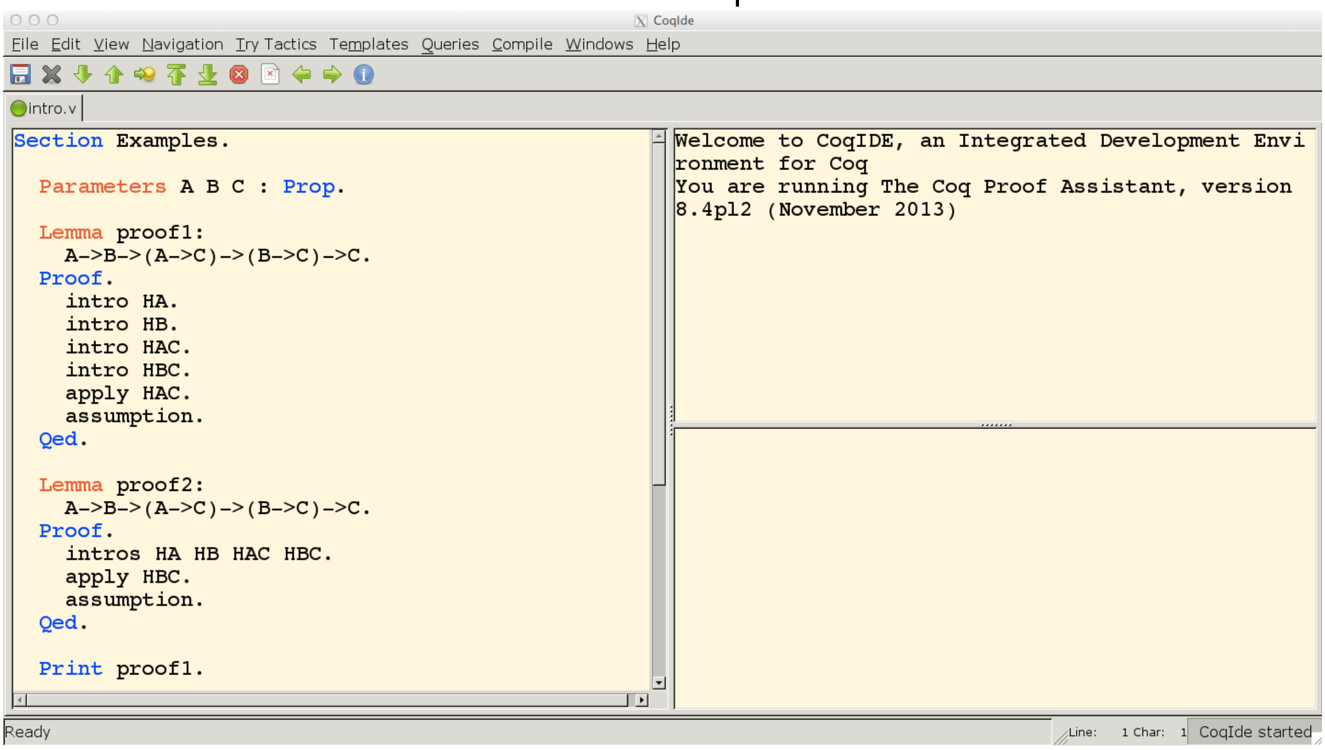
Previous examples in Coq

The Proof General mode for Emacs . . .



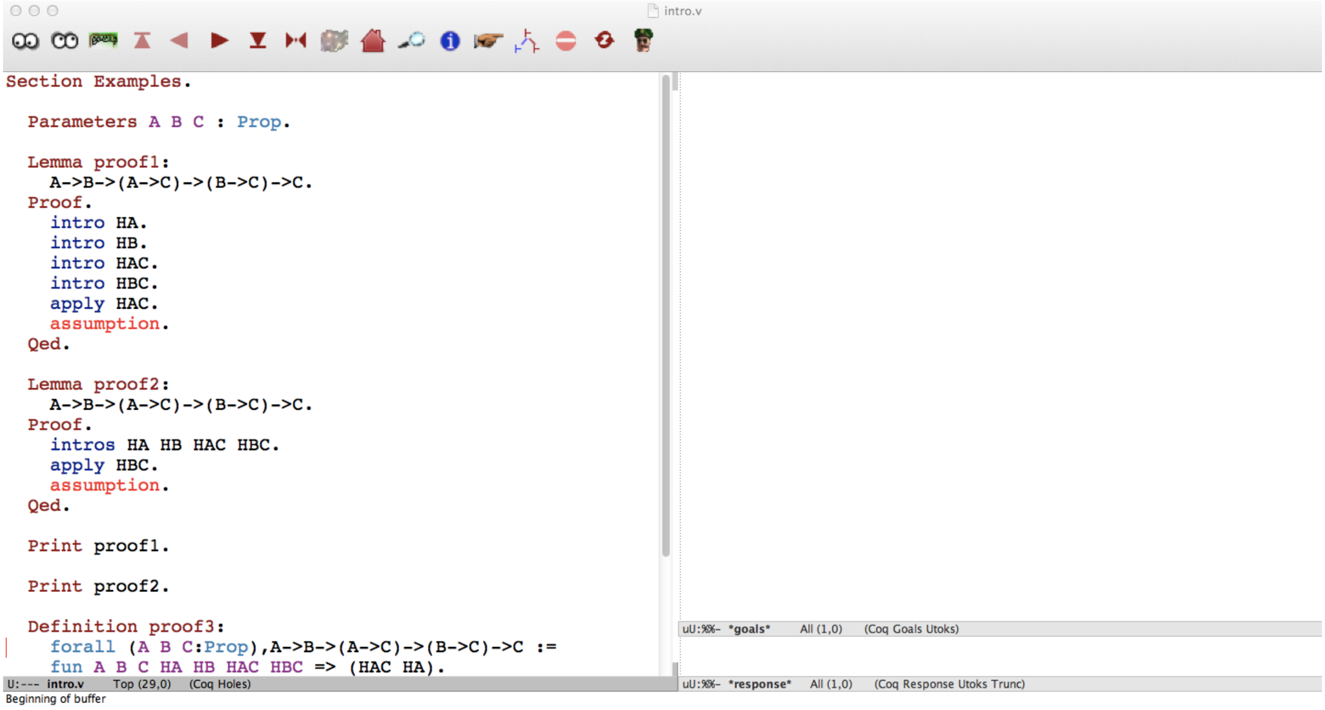
Previous examples in Coq

. . . or the CoqIDE



Previous examples in Coq

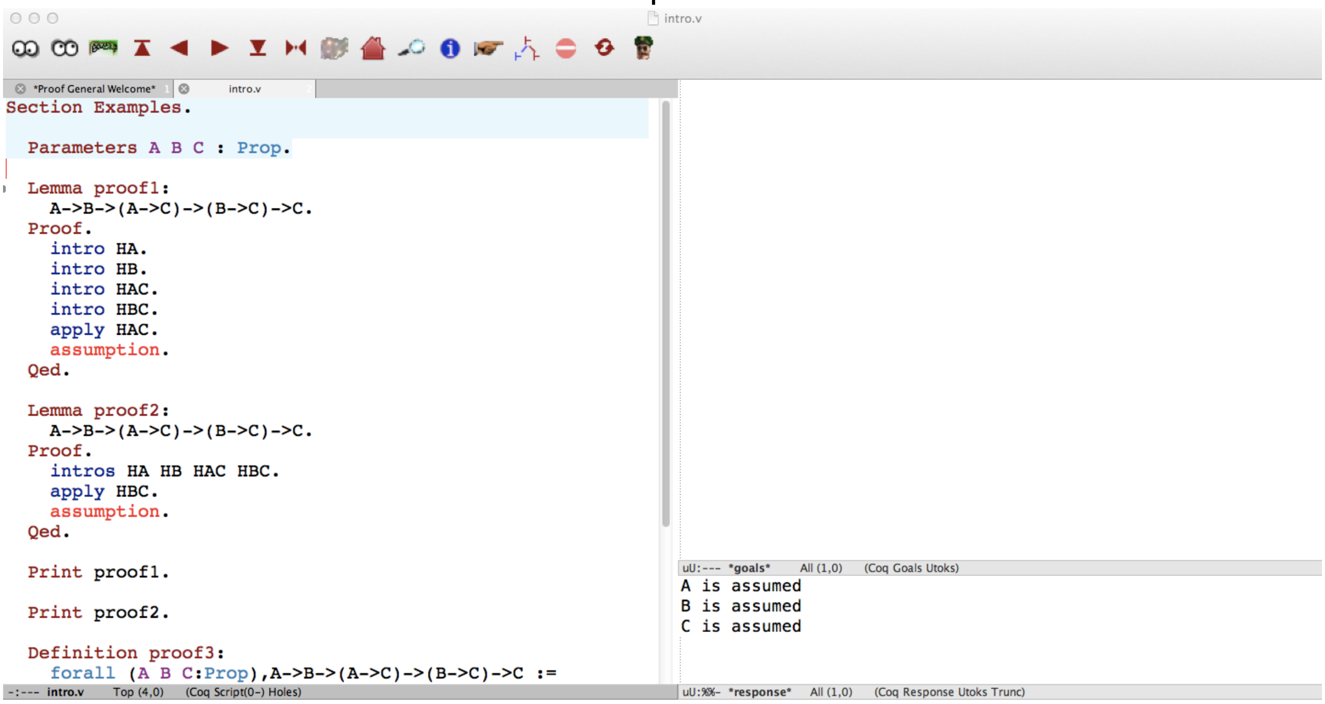
We open the file `intro.v`[1]:



[1]available at http://frederic.loulergue.eu/nii2013

Previous examples in Coq

We start to feed Coq with the commands:



Previous examples in Coq

We state a lemma and enter the interactive proof mode:

```
intro.v
*Proof General Welcome*   intro.v
Section Examples.

  Parameters A B C : Prop.

  Lemma proof1:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intro HA.
    intro HB.
    intro HAC.
    intro HBC.
    apply HAC.
    assumption.
  Qed.

  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
-:---  intro.v      Top (8,0)     (Coq Script(1-) Holes)
^C ^N
```

```
1 subgoals, subgoal 1 (ID 4)

  ______________
   A → B → (A → C) → (B → C) → C

uU:%%-  *goals*      All (4,0)     (Coq Goals Utoks)
uU:%%-  *response*   All (1,0)     (Coq Response Utoks Trunc)
```


Previous examples in Coq

The tactic `intro` "apply" the (i) rule:

```
intro.v
Section Examples.

  Parameters A B C : Prop.

  Lemma proof1:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intro HA.
    intro HB.
    intro HAC.
    intro HBC.
    apply HAC.
    assumption.
  Qed.

  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
    fun A B C HA HB HAC HBC => (HAC HA).
U:---  intro.v      Top (9,0)     (Coq Script(1-) Holes)
^C ^N
```

```
1 subgoals, subgoal 1 (ID 5)

  HA : A
  ______________
   B → (A → C) → (B → C) → C

uU:%%-  *goals*      All (5,0)     (Coq Goals Utoks)
uU:%%-  *response*   All (1,0)     (Coq Response Utoks Trunc)
```

Previous examples in Coq

The context is now similar to Γ:

```
intro.v
Section Examples.

  Parameters A B C : Prop.

  Lemma proof1:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intro HA.
    intro HB.
    intro HAC.
    intro HBC.
    apply HAC.
    assumption.
  Qed.

  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
    fun A B C HA HB HAC HBC => (HAC HA).
U:--- intro.v    Top (12,0)   (Coq Script(1-) Holes)
```

```
1 subgoals, subgoal 1 (ID 8)

  HA : A
  HB : B
  HAC : A → C
  HBC : B → C
  ______________
   C
uU:%%- *goals*   All (8,0)   (Coq Goals Utoks)
uU:%%- *response*   All (1,0)   (Coq Response Utoks Trunc)
```


Previous examples in Coq

We apply rule (a) by naming the implication part:

```
intro.v
Section Examples.

  Parameters A B C : Prop.

  Lemma proof1:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intro HA.
    intro HB.
    intro HAC.
    intro HBC.
    apply HAC.
    assumption.
  Qed.

  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
    fun A B C HA HB HAC HBC => (HAC HA).
U:--- intro.v    Top (13,0)   (Coq Script(1-) Holes)
^C ^N
```

```
1 subgoals, subgoal 1 (ID 9)

  HA : A
  HB : B
  HAC : A → C
  HBC : B → C
  ______________
   A
uU:%%- *goals*   All (8,0)   (Coq Goals Utoks)
uU:%%- *response*   All (1,0)   (Coq Response Utoks Trunc)
```

and so now we have only to deal with A . . .

Previous examples in Coq

...that is an assumption, we use rule (v):

```
Section Examples.

  Parameters A B C : Prop.

  Lemma proof1:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intro HA.
    intro HB.
    intro HAC.
    intro HBC.
    apply HAC.
    assumption.
  Qed.

  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
    fun A B C HA HB HAC HBC => (HAC HA).
```

```
No more subgoals.
```

"`No more subgoals`" $\equiv$ proof done $\equiv$ λ-term built

Previous examples in Coq

`Qed` typechecks the term against the lemma statement:

```
Section Examples.

  Parameters A B C : Prop.

  Lemma proof1:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intro HA.
    intro HB.
    intro HAC.
    intro HBC.
    apply HAC.
    assumption.
  Qed.

  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
    fun A B C HA HB HAC HBC => (HAC HA).
```

```
proof1 is defined
```

Previous examples in Coq

Second version, we do multiple `intro`:

intro.v

```
Section Examples.

  Parameters A B C : Prop.

  Lemma proof1:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intro HA.
    intro HB.
    intro HAC.
    intro HBC.
    apply HAC.
    assumption.
  Qed.

  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
    fun A B C HA HB HAC HBC => (HAC HA).
U:---  intro.v      Top (20,0)    (Coq Script(1-) Holes)
```

```
1 subgoals, subgoal 1 (ID 19)

  HA : A
  HB : B
  HAC : A → C
  HBC : B → C
  ______________
   C
uU:%%-  *goals*      All (8,0)     (Coq Goals Utoks)
uU:%%-  *response*   All (1,0)     (Coq Response Utoks Trunc)
```


Previous examples in Coq

and `apply HBC` instead of `apply HAC`:

intro.v

Proof General Welcome | intro.v

```
Section Examples.

  Parameters A B C : Prop.

  Lemma proof1:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intro HA.
    intro HB.
    intro HAC.
    intro HBC.
    apply HAC.
    assumption.
  Qed.

  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
-:---  intro.v      Top (22,6)    (Coq Script(0-) Holes)
^C ^P for goals; ^C ^L refreshes
```

```
uU:---  *goals*      All (1,0)     (Coq Goals Utoks)
proof2 is defined
uU:%%-  *response*   All (1,0)     (Coq Response Utoks Trunc)
```

Previous examples in Coq

Print t. prints the term t:

```
Section Examples.

  Parameters A B C : Prop.

  Lemma proof1:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intro HA.
    intro HB.
    intro HAC.
    intro HBC.
    apply HAC.
    assumption.
  Qed.

  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
    fun A B C HA HB HAC HBC => (HAC HA).
```

```
proof1 =
fun (HA : A) (_ : B) (HAC : A → C) (_ : B → C) ⇒ HAC HA
     : A → B → (A → C) → (B → C) → C
```

It is the λ-term we constructed "by hand"

Previous examples in Coq

The λ-term for the second proof is:

```
Section Examples.

  Parameters A B C : Prop.

  Lemma proof1:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intro HA.
    intro HB.
    intro HAC.
    intro HBC.
    apply HAC.
    assumption.
  Qed.

  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
    fun A B C HA HB HAC HBC => (HAC HA).
```

```
proof2 =
fun (_ : A) (HB : B) (_ : A → C) (HBC : B → C) ⇒ HBC HB
     : A → B → (A → C) → (B → C) → C
```

Previous examples in Coq

We could give directly the proof as a λ-term:

intro.v

```
  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
    fun A B C HA HB HAC HBC => (HAC HA).

  Definition proof4:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C.
  Proof.
    auto.
  Qed.

End Examples.
```

uU:--- *goals* All (1,0) (Coq Goals Utoks)

```
proof3 is defined
```

-:--- intro.v Bot (31,0) (Coq Script(0-) Holes) uU:%%- *response* All (1,0) (Coq Response Utoks Trunc)

^C ^N

Previous examples in Coq

...or use Coq more powerful tactics:

intro.v

```
  Lemma proof2:
    A->B->(A->C)->(B->C)->C.
  Proof.
    intros HA HB HAC HBC.
    apply HBC.
    assumption.
  Qed.

  Print proof1.

  Print proof2.

  Definition proof3:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C :=
    fun A B C HA HB HAC HBC => (HAC HA).

  Definition proof4:
    forall (A B C:Prop),A->B->(A->C)->(B->C)->C.
  Proof.
    auto.
  Qed.

End Examples.
```

uU:%%- *goals* All (1,0) (Coq Goals Utoks)

```
proof4 is defined
```

-:--- intro.v Bot (37,0) (Coq Script(0-) Holes) uU:%%- *response* All (1,0) (Coq Response Utoks Trunc)

^C ^N

Outline

Inductive definitions

```
Inductive bool :=
| true : bool
| false : bool.
```

For “data-structures”, inductive definitions are ML-like

```
Definition and (b1 b2: bool) : bool :=
  match b1 with
    | false ⇒ false
    | true ⇒ b2
  end.
```

Function definition by pattern-matching

```
Print bool.
Check bool.

Print and.
Check and.
```

`Check` returns the type of a term

Dependent types

An inductive definition could dependent on any kind of term:

- a type as in usual polymorphic definitions
- any other term

Lists

- OCaml:

  type 'a list =
  | nil | cons **of** 'a * 'a list

- Haskell:

  data List a =
  | Nil a | Cons a (**List** a)

- Coq:

```
Inductive list (A:Type) :=
| nil : list A
| cons: A → list A → list A.
```

Subsets and sigma-types

```
Inductive sig{A:Type}{P:A→Prop}:Type:=
  exist : ∀ x : A, P x → @sig A P.
```


Recursive functions and notations

```
Inductive list (A:Type) :=
| nil : list A
| cons: A → list A → list A.
```

```
Arguments nil [A].
Arguments cons [A] _ _.
```

To avoid to provide the type parameter of lists, for both `nil` and `cons`, the type argument is made *implicit*

```
Fixpoint app {A:Type}(xs ys :list A) : list A :=
  match xs with
    | nil ⇒ ys
    | cons x xs⇒ cons x (app xs ys)
  end.
```

Recursive functions must be terminating. Simple case: recursive call on a syntactic sub-term of an argument

```
Notation "[]" := nil.
Notation "x :: xs":=(cons x xs).
Notation "[ x1 ; .. ; x2 ]":=
  (cons x1 .. (cons x2 []) ..).
Notation "l1 ++ l2" :=(app l1 l2).
```

Usual notations for lists

Outline

Outline

Proofs by induction

```
Require Import list_part1.

Lemma app_nil_l:
  ∀(A:Type)(xs:list A),
    [] ++ xs = xs.
Proof.
  intros A xs.
  simpl.
  reflexivity.
Qed.

Lemma app_nil_r:
  ∀(A:Type)(xs:list A),
    xs ++ [] = xs.
Proof.
  intros A xs.
  induction xs.
  - trivial.
  - simpl. rewrite IHxs. trivial.
Qed.
```

Tactics

simpl: reduction of all the expressions in the goal

reflexivity: ends the proof if the goal has the form $e = e$

induction e: applies the induction principle associated to the type of e. Creates one sub-goal by induction case.

rewrite H: if H has the form $\forall \ldots, L = R$ finds the first sub-term that matches L in the goal, resulting in instances L' and R', then replaces all L' by R'. If H is conditional, creates new sub-goals.

Outline

Monoids: A First Definition

```
Definition associative {A:Type}(f:A→A→A) : Prop :=
  ∀ a b c : A, f (f a b) c = f a (f b c).
Definition left_neutral {A:Type}(f:A→A→A)(e:A) : Prop :=
  ∀ a, f e a = a.
Definition right_neutral {A:Type}(f:A→A→A)(e:A) : Prop :=
  ∀ a, f a e = a.
Definition monoid {A:Type}(f:A→A→A)(e:A) : Prop :=
  associative f ∧ left_neutral f e ∧ right_neutral f e.
```


Monoids: $(\mathbb{N}, +, 0)$ is a monoid

```
Require Import hom_defs.
Lemma monoid_plus_0 : monoid plus 0.
Proof.
  split.
  - intros a b c.
    induction a as [ |a Ha].
    + trivial.
    + simpl. rewrite Ha. trivial.
  - split.
    + intro a. trivial.
    + induction a as [ |a Ha].
      × trivial.
      × simpl. rewrite Ha. trivial.
Qed.
```

Tactics

split: splits a conjunctive goal into two sub-goals

induction *e* **as** *pattern*: applies the induction principle for *e* using *pattern* for naming the newly introduction terms.

$[n_1\ n_2]$: conjunctive pattern
$[n_1|n_2]$: disjunctive pattern

trivial: ends the proof either by

Folds: Definitions

```
Require Import list.
Fixpoint foldr {A B:Type}(op:A→B→B)(e:B)(xs:list A) : B :=
  match xs with
    | [] ⇒ e
    | x::xs ⇒ op x (foldr op e xs)
  end.
Fixpoint foldl {A B:Type}(op:A→B→A)(e:A)(xs:list B) : A :=
  match xs with
    | [] ⇒ e
    | x::xs ⇒ foldl op (op e x) xs
  end.
```


Folds: a Lemma

```
Require Import monoid_defs fold_defs.
Lemma folds:
  ∀ (A:Type)(op:A→A→A)(e:A),
    monoid op e →
    ∀ xs, foldr op e xs = foldl op e xs.
Proof.
  intros A op e Hmonoid xs.
  destruct Hmonoid as [Ha [Hl Hr]].
  induction xs as [ |x xs Hxs].
  - trivial.
  - simpl. rewrite Hxs. clear Hxs.
    rewrite Hl. generalize x. clear x.
    induction xs.
    + intro x. simpl. apply Hr.
    + intro x. simpl. rewrite Hl.
      rewrite ← IHxs with (x:=op x a).
      rewrite ← IHxs, Ha.
      trivial.
Qed.
```

destruct: splits a conjunctive (or disjunctive, or existential) *hypothesis* into two hypotheses. Could use the same renaming scheme than induction.

clear *H*: removes hypothesis *H* from the context.

generalize *x*: generalize the goal with respect to one of its sub-terms.

rewrite ← *H*: rewrites using the equality *H* from right to left.
rewrite *H*1**,** *H*2: rewrite using *H*1, then using *H*2.
rewrite H **with (**v:=t**)**: if *H* is a universaly quantified equality, binding variable *v*, specifies that *v* should be *t*.

Homomorphisms

```
Require Export list monoid_defs.

Definition homomorphic {A B:Type}
             (h:list A → B)(op:B→B→B) : Prop :=
  ∀ xs ys, h(xs ++ ys) = op (h xs) (h ys).

Fixpoint hom {A B:Type}(op:B→B→B)(e:B)
           (mon:monoid op e)(f:A→B)(xs:list A) : B :=
  match xs with
    | [] ⇒ e
    | x::xs ⇒op (f x) (hom op e mon f xs)
  end.

Definition ext_eq {A B:Type}(f g:A→B) : Prop :=
  ∀ a:A, f a = g a.

Notation "f == g":=(ext_eq f g)(at level 40).
```

From [4]

If f and g are functions, in Coq $f = g$ iff f and g are exactly the same. We want an equivalence relation that relates functions if their extensions are the same.

Homomorphisms: A Simple Property

```
Require Import hom_defs.

Lemma homomorphic_hom:
  ∀{A B:Type}(h:list A→B)(op:B→B→B)
         (Hom: homomorphic h op)
         (Mon: monoid op (h [])),
    h ≡ hom op (h[]) Mon (fun x⇒h[x]).
Proof.
  intros A B h op Hom Mon xs.
  induction xs as [ |x xs IH].
  - trivial.
  - simpl.
    change (x::xs) with ([x]++xs).
    rewrite Hom.
    rewrite IH.
    trivial.
Qed.
```

Tactics

change e **with** e': replaces e with e' in the goal if e and e' are convertible

First Homomorphism Theorem

```
Require Import hom_defs.
Theorem First_Homomorphism_Theorem:
  ∀{A B:Type}(op:B→B→B)(e:B)
        (m:monoid op e)(f:A→B),
    hom op e m f ≡ (hom op e m (@id B)) · map f.
Proof.
  intros A B op e m f xs.
  induction xs as [ |x xs IH].
  - trivial.
  - simpl. now f_equal.
Qed.
```

Tactics, notation, and tactical

@e: if e has implicit parameters, makes them explicit.

f_equal: if the goal is $f\ e_1\ \ldots\ e_n = g\ e_1'\ \ldots\ e_n'$ creates subgoals $f = g$, $e_1 = e_1'$, $\ldots e_n = e_n'$ and solves the simple ones.

now T: applies tactic T and if it generates sub-goals tries to solve them automatically. Fails if all subgoals are not proved automatically.

Second Homomorphism Theorem I

```
Require Import fold_defs hom_defs.
Theorem Second_Homomorphism_Theorem:
  ∀{A B:Type}(op:B→B→B)(e:B)
        (m:monoid op e)(f:A→B),
    (let oplus := fun a s ⇒ op (f a) s in
     hom op e m f ≡ foldr oplus e ) ∧
    (let otimes := fun r a⇒ op r (f a) in
     hom op e m f ≡ foldl otimes e).
Proof.
  intros A B op e m f.
  split.
  - intros oplus xs.
    induction xs as [ | x xs IH].
    + trivial.
    + simpl. unfold oplus. now f_equal.
```

Tactics

unfold e: replaces e by its definition.

Second Homomorphism Theorem II

```
- intros otimes xs.
  induction xs as [ | x xs IH].
  + trivial.
  + unfold otimes. simpl.
    destruct m as [Ha [Hnl Hnr]].
    rewrite Hnl, IH.
    clear IH. generalize (f x). clear x.
    induction xs as [ | x xs IH].
    × trivial.
    × intro b. simpl.
      rewrite ← IH with (b:=op b (f x)).
      rewrite ← IH.
      rewrite Ha.
      repeat f_equal.
      unfold otimes. rewrite Hnl.
      trivial.
Qed.
```


Outline

Operator of a homomorphic function

For a binary operator $\odot$, the list function h is $\odot$-homomorphic iff, for all lists x and y:

$$h(x +\!\!\!+\, y) = (hx) \odot (hy)$$

Note that $\odot$ is necessarily associative on the range of h [...]
Moreover, necessarily h [] is the unit of $\odot$ on the range of h

- we need to deal with partial functions
- but all functions are total in Coq

Partial functions

Ways to deal with partiality using only total functions:

Function returning an optional value

```
Inductive option (A : Type) : Type :=
  | Some : A → option A
  | None : option A.
Require Import list.

Fixpoint nth_option{A:Type}(n:nat)(xs:list A):option A:=
  match xs with
    | [] ⇒ None
    | x::xs ⇒
      match n with
        | 0 ⇒ Some x
        | S n ⇒ nth_option n xs
      end
  end.
```

Partial functions

Ways to deal with partiality using only total functions:

Function taking an **additional parameter**

that is returned if outside the range:

```
Require Import list.

Fixpoint nth {A:Type}(n:nat)(xs:list A)(default:A): A :=
  match xs with
    | [] => default
    | x::xs =>
      match n with
        | 0 => x
        | S n => nth n xs default
      end
  end.
```


Partial functions

Ways to deal with partiality using only total functions:

Function with **pre-conditions** on the parameters

```
Require Import list.

Require Import Omega Program.

Local Obligation Tactic :=
  (program_simpl; simpl in *; omega).

Program Fixpoint nth_pre {A:Type}(n:nat)(xs:list A)
        (H: n < length xs): A :=
  match xs with
    | [] => _
    | x::xs => match n with
                 | 0 => x
                 | S n => nth_pre n xs _
               end
  end.
```

Partial functions

Ways to deal with partiality using only total functions:

Function with **pre-conditions** on the parameters

```
Program Fixpoint nth_sig {A:Type}(xs:list A)
        (n:{n:nat|n < length xs}): A :=
  match xs with
    | [] ⇒ _
    | x::xs ⇒ match n with
                 | 0 ⇒ x
                 | S n ⇒ nth_sig xs n
               end
  end.
```

- where `{x : A | P x}` is a notation for `@sig A P`
- a value of this type is a dependent pair containing:
  - a value `x` of type `A`
  - a proof of `P x`

Operator of a homomorphic function I

The subset of B that is in the range of h:

```
Definition range {A B:Set}(h:list A→B) :=
  {b:B | ∃ xs, h xs = b}.
```

A value of type *range h* is a pair consisting of a value of type B and a proof that it is in the range of h.

Operator of a homomorphic function II

Seeing ($h\ xs$) as a value of type *range h*:

```
Definition to_range {A B:Set} (h:list A→B)(xs:list A) : range h :=
  let P := fun b⇒∃ xs, h xs=b in
  let prf := ex_intro (fun xs0⇒h xs0=h xs) xs eq_refl in
  exist P (h xs) prf.
```


Operator of a homomorphic function III

To get the value of type B from a *range h*:

```
Definition of_range1 {A B:Set} {h:list A→B}(b:range h) : B :=
  match b with
    | exist b _ ⇒ b
  end.
```

A more generic function is defined in Coq library: *proj1_sig*.
To get the proof of type $\exists\ xs,\ h\ xs = b$ from a *range h*:

```
Definition of_range2 {A B:Set} {h:list A→B}(b:range h) :
  ∃ xs, h xs = of_range1 b :=
  match b with
    | exist _ prf ⇒ prf
  end.
```

A more generic function is defined in Coq library: *proj2_sig*.

Operator of a homomorphic function IV

It is not possible to define such a function:

```
Definition list_of_range {A B:Set} {h:list A→B}(b:range h): list A.
Proof.
Abort.
```


Operator of a homomorphic function V

An auxiliary lemma:

```
Lemma range_op:
  ∀ {A B:Set}(h:list A→B)(op:B→B→B)
         (hom:homomorphic h op)(b1 b2:B),
    (∃ xs1, h xs1 = b1) →
    (∃ xs2, h xs2 = b2) →
    (∃ xs, h xs = op b1 b2).
Proof.
  intros A B h op hom b1 b2
         [xs1 Hb1] [xs2 Hb2].
  rewrite ← Hb1, ← Hb2, ← hom.
  exists (xs1++xs2).
  reflexivity.
Defined.
```

Tactics

exists *e*: if the goal has the form $\exists x.g$, provides a x and the goal becomes g

Operator of a homomorphic function VI

Using the Program feature of Coq, we define an operator on the range of h, from this operator and h:

```
Program Definition restrict {A B:Set}
        {h:list A→B}(op:B→B→B)
        (hom:homomorphic h op) :
  range h → range h → range h :=
  fun (x y:range h) ⇒ op x y.
Next Obligation.
  destruct x as [x [xs Hx]].
  destruct y as [y [ys Hy]].
  apply range_op.
  - trivial.
  - eexists. simpl. eassumption.
  - eexists. eassumption.
Defined.
```

Tactics

eexists: creates an existantial variable and gives it as a witness. At the end of the proof there shoud be no remaining existential variable.

eassumption: same as assumption but could eliminate existential variables in the goal.

Operator of a homomorphic function VII

to_range is injective: both the value and the proofs are equal when the values are equal:

```
Lemma to_range_inj:
  ∀ {A B:Set} {h:list A→B}(xs ys:list A),
    xs = ys →
    to_range h xs = to_range h ys.
Proof.
  intros A B h xs ys Heq.
  rewrite Heq.
  trivial.
Qed.
```

Operator of a homomorphic function VIII

Any value of type *range h* could be obtained using the function *to_range*:

```
Lemma norm :
  ∀ {A B:Set}{h:list A→B}(b:range h),
  ∃ xs, b = to_range h xs.
Proof.
  intros A B h b.
  destruct b as [b [xs Hb]].
  exists xs.
  rewrite ← Hb.
  now apply to_range_inj.
Qed.
```


Operator of a homomorphic function IX

restrict and *to_range* composition:

```
Lemma restrict_to_range:
  ∀ {A B:Set} {h:list A→B}{op:B→B→B}
     (hom:homomorphic h op) (xs ys:list A),
    restrict op hom (to_range h xs)(to_range h ys) =
    to_range h (xs++ys).
Proof.
  intros A B h op hom xs ys.
  unfold restrict, restrict_obligation_1, to_range.
  simpl.
  rewrite ← hom.
  reflexivity.
Qed.
```

This lemma could be proven because *restrict* and its associated obligation *restrict_obligation_1* have been carefully designed and made *transparent* using Defined instead of Qed.

Operator of a homomorphic function X

op restricted to the range of *h* has (*h* []) as a left neutral:

```
Lemma homomorphic_op_left_neutral:
  ∀ {A B:Set}(h:list A → B) (op:B→B→B) (hom:homomorphic h op),
    left_neutral (restrict op hom) (to_range h []).
Proof.
  intros A B h op hom b.
  destruct (norm b) as [xs Hb].
  rewrite Hb.
  rewrite restrict_to_range.
  now apply to_range_inj.
Qed.
```


Operator of a homomorphic function XI

op restricted to the range of *h* has (*h* []) as a right neutral:

```
Lemma homomorphic_op_right_neutral:
  ∀ {A B:Set}(h:list A → B) (op:B→B→B) (hom:homomorphic h op),
    right_neutral (restrict op hom) (to_range h []).
Proof.
  intros A B h op hom b.
  destruct (norm b) as [xs Hb].
  rewrite Hb.
  rewrite restrict_to_range.
  apply to_range_inj.
  apply app_nil_r.
Qed.
```

Operator of a homomorphic function XII

op restricted to the range of *h* is associative:

```
Lemma homomorphic_op_assoc:
  ∀ {A B:Set}(h:list A → B)(op:B→B→B)
     (hom:homomorphic h op),
    associative (restrict op hom).
Proof.
  intros A B h op hom b1 b2 b3.
  destruct (norm b1) as [xs1 Hb1].
  destruct (norm b2) as [xs2 Hb2].
  destruct (norm b3) as [xs3 Hb3].
  subst.
  repeat rewrite restrict_to_range.
  apply to_range_inj.
  rewrite app_assoc.
  trivial.
Qed.
```

Tactic & Tactical

subst: rewrites in the goal and the context using all the equalities of the context that have the form $v = e$ where v is a variable, then clears all these equalities.

repeat T: repeats the tactic T until its application fails.

Dealing with subset/sigma types

Subset/sigma types

```
Inductive sig{A:Type}{P:A→Prop}:Type:=
  exist : ∀ x : A, P x → @sig A P.
```

Alternative solutions (not possible in all cases):

- The proof part has for type an equality on a type with decidable equality: In this case the unicity of the equality proofs is proved[2]
- Prove that given a value v the proof of $P\ v$ is unique
- Carefull design of the functions and proofs so that the equality of proofs is true in the cases your are interested in,
- Use of the proof irrevelance axiom, in *Coq.Logic.ProofIrrelevance*:

  `Axiom proof_irrelevance : ∀ (P:Prop) (p1 p2:P), p1 = p2.`

  and its consequences in *ProofIrrelevanceTheory*

[2] see `Coq.Logic.Eqdep_dec`

Outline

1. Introduction
2. Functional programming in Coq
3. Stating and proving properties
4. Program extraction
5. Bibliography

Program extraction I

Coq

```
Require Import nth.
Extraction nth.nth.
```

OCaml

```
(** val nth_pre : nat → 'a1 list → 'a1 **)
let rec nth_pre n xs = match xs with
| Coq_nil → nth_pre_obligation_1 n xs
| Coq_cons (x, xs0) →
  (match n with
   | O → x
   | S n0 → nth_pre n0 xs0)
```

Program extraction II

Coq

```
Require Import nth.
Recursive Extraction nth.nth_pre.
```

OCaml

```
type nat = | O | S of nat
type 'a list = | Nil | Cons of 'a * 'a list
(** val nth_pre_obligation_1 : nat -> 'a1 list -> 'a1 **)
let nth_pre_obligation_1 n xs = assert false (* absurd case *)
(** val nth_pre : nat -> 'a1 list -> 'a1 **)
let rec nth_pre n xs = match xs with
| Nil -> nth_pre_obligation_1 n xs
| Cons (x, xs0) -> (match n with
                    | O -> x
                    | S n0 -> nth_pre n0 xs0)
```


Program extraction III

Coq

```
Require Import nth.
Extract Inductive list => "list" [ "[]" "(::)" ].
Extraction nth.nth_sig.
```

OCaml

```
(** val nth_sig : 'a1 list -> nat -> 'a1 **)
let rec nth_sig xs n =
  match xs with
  | [] -> nth_sig_obligation_1 xs n
  | x::xs0 ->
    (match n with
     | O -> x
     | S n0 -> nth_sig xs0 n0)
```

Outline

Bibliography I

[1] Y. Bertot. Coq in a hurry, 2006. http://hal.inria.fr/inria-00001173.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[3] A. Chlipala. An Introduction to Programming and Proving with Dependent Types in Coq. *Journal of Formalized Reasoning*, 3(2), 2010. doi:10.6092/issn.1972-5787/1978.

[4] J. Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657–665, 1996. doi:10.1017/S0956796800001908.

[5] The Coq Development Team. The Coq Proof Assistant. http://coq.inria.fr.