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 correspondence
Curry-Howard Correspondance

Natural Deduction

\[
\begin{align*}
(v) & \quad A \in \Gamma \\
\frac{}{\Gamma \vdash A} \\
(i) & \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \\
(a) & \quad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}
\end{align*}
\]

Simply Typed \(\lambda\)-Calculus

\[
\begin{align*}
(V) & \quad x : A \in \Gamma \\
\frac{}{\Gamma \vdash x : A} \\
(L) & \quad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x : A. e) : A \rightarrow B} \\
(A) & \quad \frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash (e \ e') : B}
\end{align*}
\]

Curry-Howard Correspondance

Natural Deduction – Example 1

\[
\begin{align*}
(v) & \quad A \rightarrow C \in \Gamma \\
\frac{}{\Gamma \vdash A \rightarrow C} \\
(v) & \quad A \in \Gamma \\
\frac{}{\Gamma \vdash A} \\
(i) & \quad \frac{\Gamma \equiv A, B, A \rightarrow C, B \rightarrow C \vdash C}{A, B, A \rightarrow C \vdash (B \rightarrow C) \rightarrow C} \\
(i) & \quad \frac{A, B \vdash (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C}{A \vdash B \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C} \\
(i) & \quad \frac{A \vdash B \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C}{\vdash A \rightarrow B \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C}
\end{align*}
\]
Curry-Howard Correspondance

Natural Decduction – Example 2

\[
\begin{align*}
&\frac{(\forall) \quad B \to C \in \Gamma}{\Gamma \vdash B \to C} \quad (\forall) \quad B \in \Gamma \\
&\frac{(i) \quad \Gamma \equiv A, B, A \to C, B \to C \vdash C}{A, B, A \to C \vdash (B \to C) \to C} \quad (i) \\
&\frac{(i) \quad A, B \vdash (A \to C) \to (B \to C) \to C}{A \vdash B \to (A \to C) \to (B \to C) \to C} \quad (i) \\
&\vdash A \to B \to (A \to C) \to (B \to C) \to C
\end{align*}
\]

Curry-Howard Correspondance

\[\lambda\text{-calculus: find a term with the given type}\]

\[
\begin{align*}
&\frac{(V) \quad f:A \to C \in \Gamma}{\Gamma \vdash f:A \to C} \quad (V) \quad x:A \in \Gamma \\
&\frac{(L) \quad \Gamma \equiv x:A, y:B, f:A \to C, g:B \to C \vdash ? \quad : \quad C}{x:A, y:B, f:A \to C \vdash ?} \quad (L) \\
&\frac{(L) \quad x:A, y:B \vdash ? \quad : \quad (A \to c) \to (B \to C) \to C}{x:A \vdash ?} \quad (L) \\
&\vdash ? \quad : \quad A \to B \to (A \to C) \to (B \to C) \to C
\end{align*}
\]

\[\lambda x:A. \lambda y:B. \lambda f:A \to C. \lambda g:B \to C. (f \ x)\]

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 \( \lambda \)-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

... or the CoqIDE
We open the file `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).
```

1available at http://frederic.loulergue.eu/nii2013
Previous examples in Coq

We state a lemma and enter the interactive proof mode:

```
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.
```

```
Definition proof3: forall (A B C Prop), A -> B -> (A -> C) -> (B -> C) -> C :=
```

Previous examples in Coq

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

```
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.
```

```
Definition proof3: forall (A B C Prop), A -> B -> (A -> C) -> (B -> C) -> C :=
```

Previous examples in Coq

The context is now similar to \( \Gamma \):

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

and so now we have only to deal with \( A \) ...
Previous examples in Coq

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

```
No more subgoals

叩 proof done

叩 -term built
```

F. Loulergue
SyDPaCC – Lecture 2
October-November 2013

19 / 68

Previous examples in Coq

Qed typechecks the term against the lemma statement:

```
No more subgoals ≡ proof done ≡ λ-term built
```

F. Loulergue
SyDPaCC – Lecture 2
October-November 2013

20 / 68
Previous examples in Coq

Second version, we do multiple intro:

\begin{verbatim}
Section Examples.

Parameters A B C : Prop.

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

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

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

and apply HBC instead of apply HAC:
Previous examples in Coq

Print t. prints the term $t$:

It is the $\lambda$-term we constructed “by hand”

F. Loulergue
SyDPaCC – Lecture 2
October-November 2013
23 / 68

Previous examples in Coq

The $\lambda$-term for the second proof is:
We could give directly the proof as a $\lambda$-term:

\begin{verbatim}
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.
\end{verbatim}

...or use Coq more powerful tactics:
Inductive definitions

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

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

Print bool.
Check bool.
Print and.
Check and.

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

Function definition by pattern-matching

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 : \forall 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] _ _.
Fixpoint app {A:Type} (xs ys :list A) : list A :=
  match xs with
  | nil => ys
  | cons x xs => cons x (app xs ys)
end.
Notation "[]" := nil.
Notation "x :: xs" := (cons x xs).
Notation "[ x1 ; .. ; x2 ]" :=
  (cons x1 .. (cons x2 []) ..).
Notation "l1 ++ l2" := (app l1 l2).
```

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

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

Usual notations for lists
Outline

1. Introduction
2. Functional programming in Coq
3. Stating and proving properties
   - More tactics
     - Homomorphism theorems on lists
     - Partial functions
4. Program extraction
5. Bibliography
Proofs by induction

Require Import list_part1.

Lemma app_nil_l:
\(\forall (A:\Type)(xs:list\ A),\)
\([\mathbf{} ++ xs = xs\).  
Proof.
  intros A xs.
  simpl.
  reflexivity.
Qed.

Lemma app_nil_r:
\(\forall (A:\Type)(xs:list\ A),\)
\(xs ++ [\mathbf{} = xs\).  
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

1. Introduction
2. Functional programming in Coq
3. Stating and proving properties
   - More tactics
   - Homomorphism theorems on lists
   - Partial functions
4. Program extraction
5. Bibliography
Monoids: A First Definition

Definition **associative** \( \{ A : \text{Type} \} (f : A \to A \to A) : \text{Prop} := \) 
\( \forall \ a \ b \ c : A, \ f \left( f \ a \ b \right) \ c = f \ a \left( f \ b \ c \right). \)

Definition **left_neutral** \( \{ A : \text{Type} \} (f : A \to A \to A)(e : A) : \text{Prop} := \) 
\( \forall \ a, \ f \ e \ a = a. \)

Definition **right_neutral** \( \{ A : \text{Type} \} (f : A \to A \to A)(e : A) : \text{Prop} := \) 
\( \forall \ a, \ f \ a \ e = a. \)

Definition **monoid** \( \{ A : \text{Type} \} (f : A \to A \to A)(e : A) : \text{Prop} := \) 
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
\([n_1 \mid n_2]:\) disjunctive pattern

trivial: ends the proof either by
Folds: Definitions

Require Import list.

Fixpoint foldr \{ A B:Type \}(op:A\to B\to B)(e:B)(xs:list A) : B :=
match xs with
| [] \Rightarrow e
| x::xs \Rightarrow op x (foldr op e xs)
end.

Fixpoint foldl \{ A B:Type \}(op:A\to B\to A)(e:A)(xs:list B) : A :=
match xs with
| [] \Rightarrow e
| x::xs \Rightarrow foldl op (op e x) xs
end.

Folds: a Lemma

Require Import monoid_defs fold_defs.

Lemma folds:
\forall (A:Type)(op:A\to A\to A)(e:A),
\textit{monoid op e} \to
\forall 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 \langle \forall x, \textit{foldr op e xs} = \textit{foldl op e xs} \rangle
    with (x:=op x a).
    rewrite \langle \forall x, \textit{foldr op e xs} = \textit{foldl op e xs} \rangle with (x:=op x a).
    rewrite \forall x, \textit{foldr op e xs} = \textit{foldl op e xs}.
    trivial.

Qed.
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:
\[ \forall \{ A B : \text{Type} \} \text{(op:B→B→B)}(e:B) \]
\[ (m:\text{monoid op e})(f:A→B), \]
\[ \text{hom op e m f} \equiv (\text{hom op e m (id B)}) \cdot \text{map f}. \]

Proof.
intros A B op e m f xs.
induction xs as [ | x xs IH].
- trivial.
- simpl. now f_equal.
Qed.

Second Homomorphism Theorem I

Require Import fold_defs hom_defs.

Theorem Second_Homomorphism_Theorem:
\[ \forall \{ A B : \text{Type} \} \text{(op:B→B→B)}(e:B) \]
\[ (m:\text{monoid op e})(f:A→B), \]
\[ (\text{let oplus := fun a s ⇒ op (f a) s in} \]
\[ \text{hom op e m f} \equiv \text{foldr oplus e} ) \land \]
\[ (\text{let otimes := fun r a⇒ op r (f a) in} \]
\[ \text{hom op e m f} \equiv \text{foldl otimes e}). \]

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

1 Introduction

2 Functional programming in Coq

3 Stating and proving properties
   More tactics
   Homomorphism theorems on lists
   Partial functions

4 Program extraction

5 Bibliography
Operator of a homomorphic function

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

$$h(x \mathbin{\oplus} y) = (hx) \odot (hy)$$

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 \( \text{nth} \{ A \text{: Type} \}(n:nat)(xs:list A)(default:A) : A := \)

match xs with
| [] \Rightarrow \text{default}
| x::xs \Rightarrow 
  
  match n with
  | 0 \Rightarrow x
  | S n \Rightarrow \text{nth} n xs default

end
end.

F. Loulergue
SyDPaCC – Lecture 2
October-November 2013
47 / 68

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 \( \text{nth\_pre} \{ A \text{: Type} \}(n:nat)(xs:list A) \)
  \( (H: n < \text{length} xs) : A := \)

match xs with
| [] \Rightarrow _
| x::xs \Rightarrow match n with
  | 0 \Rightarrow x
  | S n \Rightarrow \text{nth\_pre} n xs _

end

end.

F. Loulergue
SyDPaCC – Lecture 2
October-November 2013
48 / 68
Partial functions

Ways to deal with partiality using only total functions:

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

Program Fixpoint \( \text{nth}_\text{sig} \ {\{A: \text{Type}\}\{xs: \text{list} \ A\}} \ (n: \{n: \text{nat}\} | n < \text{length} \ xs\}): A :=

\[
\begin{align*}
\text{match} \ xs \ \text{with} \\
\mid \emptyset & \Rightarrow \_ \\
\mid x::xs & \Rightarrow \text{match} \ n \ \text{with} \\
\mid 0 & \Rightarrow x \\
\mid S \ n & \Rightarrow \text{nth}_\text{sig} \ xs \ n \\
\end{align*}
\]

end.

\( I \)

where \( \{x : A \ | \ P \ x\} \) is a notation for \( \text{@sig} \ A \ P \)

\( I \) a value of this type is a dependent pair containing:

\( I \) a value \( x \) of type \( A \)

\( I \) a proof of \( P \ x \)

Operator of a homomorphific function \( I \)

The subset of \( B \) that is in the range of \( h \):

**Definition** \( \text{range} \ {\{A B: \text{Set}\}\{h: \text{list} \ A \rightarrow B\}} := \{b: B \ | \ \exists \ xs, h \ xs = b\} \).

A value of type \( \text{range} \ h \) 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 \(\text{range } h\):

\[
\text{Definition } \text{to\_range} \{ A \, B : \text{Set} \} \, (h: \text{list } A \to B)(xs: \text{list } A) : \text{range } h :=
\]
\[
\text{let } P := \text{fun } b \mapsto \exists \; xs, \; h \, xs = b \text{ in }
\]
\[
\text{let } \text{prf} := \text{ex\_intro} \left( \text{fun } xs0 \mapsto h \, xs0 = h \, xs \right) \text{ xs eq\_refl } \text{in }
\]
\[
\text{exist } P \left( h \, xs \right) \, \text{prf}.
\]

Operator of a homomorphic function III

To get the value of type \(B\) from a \(\text{range } h\):

\[
\text{Definition } \text{of\_range1} \{ A \, B : \text{Set} \} \, \{ h: \text{list } A \to B \} \{ b: \text{range } h \} : B :=
\]
\[
\text{match } b \text{ with }
\]
\[
| \text{exist } b \_ \Rightarrow b
\]
\[
\text{end}.
\]

A more generic function is defined in Coq library: \(\text{proj1\_sig}\).

To get the proof of type \(\exists \; xs, \; h \, xs = b\) from a \(\text{range } h\):

\[
\text{Definition } \text{of\_range2} \{ A \, B : \text{Set} \} \, \{ h: \text{list } A \to B \} \{ b: \text{range } h \} :
\]
\[
\exists \; xs, \; h \, xs = \text{of\_range1} \, b :=
\]
\[
\text{match } b \text{ with }
\]
\[
| \text{exist } \_ \, \text{prf} \Rightarrow \text{prf}
\]
\[
\text{end}.
\]

A more generic function is defined in Coq library: \(\text{proj2\_sig}\).
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:
\[ \forall \{ A B: Set \}(h: list A → B)(op: B → B → B)
   (hom: homomorphic h op)(b1 b2:B), \]
\[ (\exists xs1, h xs1 = b1) → \]
\[ (\exists xs2, h xs2 = b2) → \]
\[ (\exists xs, h xs = op b1 b2). \]

**Proof.**
intros A B h op hom b1 b2
[xs1 Hb1] [xs2 Hb2].
rewrite ← Hb1, ← Hb2, ← hom.
exists (xs1 ++ xs2).
reflexivity.

**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 existential variable and gives it as a witness. At the end of the proof there should be no remaining existential variable.

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

Operator of a homomorphic function VII

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

Lemma `t_to_range_inj`:
\[ ∀ \{A B:Set\} \{h:list A→B\}(xs ys:list A), \]
\[ xs = ys → \]
\[ t_to_range h xs = t_to_range h ys. \]

Proof.
- intros \( A B h xs ys Heq \).
- rewrite `Heq`.
- trivial.
- Qed.
Any value of type \( \text{range} \ h \) could be obtained using the function \( \text{to\_range} \):

**Lemma norm** :
\[
\forall \{A \ B:\text{Set}\}\{h:\text{list}\ A\to B\}\{b:\text{range} \ h\}, \\
\exists \ xs, \ b = \text{to\_range} \ h \ xs.
\]

**Proof**.
\[
\text{intros} \ A \ B \ h \ b. \\
\text{destruct} \ b \ \text{as} \ [b \ [xs \ Hb]]. \\
\text{exists} \ xs. \\
\text{rewrite} \leftarrow \ Hb. \\
\text{now apply} \ \text{to\_range\_inj}. \\
\text{Qed}.
\]

---

**Operator of a homomorphic function IX**

\( \text{restrict} \) and \( \text{to\_range} \) composition:

**Lemma restrict\_to\_range** :
\[
\forall \{A \ B:\text{Set}\}\{h:\text{list}\ A\to B\}\{\text{op}:B\to B\to B\} \\
(\text{hom}:\text{homomorphic} \ h \ \text{op}) \ (xs \ ys:\text{list} \ A), \\
\text{restrict} \ \text{op} \ \text{hom} \ (\text{to\_range} \ h \ xs)(\text{to\_range} \ h \ ys) = \\
\text{to\_range} \ h \ (xs++ys).
\]

**Proof**.
\[
\text{intros} \ A \ B \ h \ \text{op} \ \text{hom} \ xs \ ys. \\
\text{unfold} \ \text{restrict}, \ \text{restrict\_obligation\_1}, \ \text{to\_range}. \\
\text{simpl}. \\
\text{rewrite} \leftarrow \ \text{hom}. \\
\text{reflexivity}. \\
\text{Qed}.
\]

This lemma could be proven because \( \text{restrict} \) and its associated obligation \( \text{restrict\_obligation\_1} \) have been carefully designed and made \( \text{transparent} \) using \( \text{Defined} \) instead of \( \text{Qed} \).
Operator of a homomorphic function $X$

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

**Lemma** homomorphic\_op\_left\_neutral:
\[
\forall \{A \rightarrow B\}(\text{list} \ A \rightarrow \text{B} \rightarrow \text{B}) \ (\text{hom:homomorphic} \ h \ \text{op}),
\]
\[
\text{left\_neutral} \ (\text{restrict \ op \ hom}) \ (\text{to\_range \ h \ []}).
\]

**Proof.**
- intros $A \ B \ h \ \text{op} \ h \ b$.
- destruct $(\text{norm \ b})$ as $[xs \ \text{Hb}]$.
- rewrite $\text{Hb}$.
- rewrite $\text{restrict\_to\_range}$.
- now apply $\text{to\_range\_inj}$.

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:
\[
\forall \{A \rightarrow B\}(\text{list} \ A \rightarrow \text{B} \rightarrow \text{B}) \ (\text{hom:homomorphic} \ h \ \text{op}),
\]
\[
\text{right\_neutral} \ (\text{restrict \ op \ hom}) \ (\text{to\_range \ h \ []}.
\]

**Proof.**
- intros $A \ B \ h \ \text{op} \ h \ b$.
- destruct $(\text{norm \ b})$ as $[xs \ \text{Hb}]$.
- rewrite $\text{Hb}$.
- rewrite $\text{restrict\_to\_range}$.
- apply $\text{to\_range\_inj}$.
- apply $\text{app\_nil\_r}$.

Qed.
Operator of a homomorphic function XII

\( op \) restricted to the range of \( h \) is associative:

**Lemma homomorphic_op_assoc:**

\[ \forall \{A \, B : \text{Set}\}(h : \text{list } A \rightarrow B)(op : B \rightarrow B \rightarrow B) \]

\[ (\text{hom:homomorphic } h \, op), \]

\[ \text{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.

Dealing with subset/sigma types

**Subset/sigma types**

\[
\text{Inductive } \text{sig}\{A : \text{Type}\}\{P : A \rightarrow \text{Prop}\} : \text{Type} := \\
\text{exist} : \forall x : A, P x \rightarrow @\text{sig } A \, P.
\]

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 you are interested in,
- Use of the proof irrevelance axiom, in \text{Coq.Logic.ProofIrrelevance}:

\[ \text{Axiom proof_irrelevance} : \forall (P : \text{Prop}) (p1 \, p2 : P), p1 = p2. \]

\[ \text{and its consequences in } \text{ProofIrrelevanceTheory} \]

\(^2\)see \text{Coq.Logic.Eqdep_dec}
Program extraction

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 \Rightarrow "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

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

Bibliography I

http://hal.inria.fr/inria-00001173.


doi:10.1017/S0956796800001908.