Lecture 4 : Advanced functional programming with Coq

- Example minimal.v, Monads.v
- Challenges section 5.2 ListChallenges.v
Outline

- Introduction
- Basics of CoQ system
- Using simple inductive definitions
- Functional programming with CoQ
  - Recursive functions
  - Partiality and dependent types
  - Simple programs on lists
  - Well-founded induction and recursion
  - Imperative programming
Representing a partial function

- \( f : A \rightarrow B \) defined only on a subset \( \text{dom} \) of \( A \).
- extend \( f \) in an arbitrary way (need a default value in \( B \))
- use an option type:

  Coq? Print option.
  Inductive option (A : Type) : Type
  := Some : A \rightarrow option A | None : option A

  - need to consider the \( \text{None} \) case explicitely
  - can be hidden with monadic notations

- Consider the function as a relation of type \( A \rightarrow B \rightarrow \text{Prop} \)
  - Prove functionality
  - \( F x y \) instead of \( f x \)
  - no computation
Introducing logic in types

Types and properties can be mixed freely.

- add an explicit precondition to the function:

  \[ f : \forall x : A, \text{dom} x \rightarrow B \]

- each call to \( f a \) requires a proof \( p \) of \( \text{dom} a \) internally: \( f a p \).

- partially hide the proof in a subset type:

  \[ f : \{ x : A | \text{dom} x \} \rightarrow B \quad \text{internally} \quad f (a, p) \]

  Coq? Print sig.

  Inductive sig (A : Type) (P : A \rightarrow Prop) : Type :=
  exist : forall x : A, P x \rightarrow sig P

- use high-level tools like Program or type classes to separate programming from solving proof obligations.
Using subset types for specifications

▶ Proposition can appear also to restrict images:

\[ S : \text{nat} \rightarrow \{ n : \text{nat} | 0 < n \} \]

▶ restrictions can depend on the input:

\[ \text{next} : \forall n : \text{nat}, \{ m : \text{nat} | n < m \} \]

Dependent types
Other useful dependent types

Coq? `Print` `sumbool`.
`Inductive sumbool (A B : Prop) : Set :=
  left : A -> {A} + {B} | right : B -> {A} + {B}`

Coq? `Print` `sumor`.
`Inductive sumor (A : Type) (B : Prop) : Type :=
inleft : A -> A + {B} | inright : B -> A + {B}`

Coq? `Check` `forall` `n m, {n <= m}+{m < n}`.

Coq? `Check` `forall` `n, { m | m < n }+{n=0}`.

**Use objects in `sumbool` as booleans value:**

Coq? `Check` `zerop`.
`zerop
  : forall n : nat, {n = 0} + {0 < n}`

Coq? `Definition` `choice` `n x y:nat` `:=
  if zerop n then x else y`.
`choice` `is defined`
Annotated programs

Advantages:

▶ Develop program and proof simultaneously.
▶ The specification is available each time the program is used.
▶ Possibly discovery of a program from a proof.

Coq? Goal forall n m, {n <= m} + {m <= n}.

Drawbacks:

▶ Inside Coq, the program contains proof-terms: printing, reduction can become awful.
▶ Some proof-irrelevance mechanism needed (primitive in PVS)
Outline

- Introduction
- Basics of CoQ system
- Using simple inductive definitions
- Functional programming with CoQ
  - Recursive functions
  - Partiality and dependent types
  - Simple programs on lists
  - Well-founded induction and recursion
  - Imperative programming
Coq? Require Import List.

Coq? Print list.
Inductive list (A : Type) : Type :=
   nil : list A | cons : A -> list A -> list A
For nil: Argument A is implicit and maximally inserted

First challenge on computing sum and max proving:

\[ \text{sum} \leq \text{length} \times \text{max} \]
Outline

- Introduction
- Basics of CoQ system
- Using simple inductive definitions
- Functional programming with CoQ
  - Recursive functions
  - Partiality and dependent types
  - Simple programs on lists
  - Well-founded induction and recursion
  - Imperative programming
Well-founded induction and recursion

- Only structural recursion is accepted
- But it can be structural on complex inductive definitions
- Loop:
  
  \[
  \text{let } f \ n = \text{if } p \ n \ \text{then } n \ \text{else } f \ (n + 1) \n  \]

- Fixpoint for well-founded relations:

  \[
  \text{let } f \ x = t(x, f) \n  \]

  Any call to \( f \ y \) in \( t \) is such that \( y < x \) for a well-founded relation (no infinite decreasing sequence).
Analysing the loop construction

\[
\text{let } f \ n = \ \text{if } p \ n \ \text{then } n \ \text{else } f(n + 1)
\]

Terminates only if there exists \( m \geq n \) such that \( p \ m = \text{true} \).

See \texttt{minimal.v}
Typing well-founded fixpoint

\textbf{let } f \ x = t(x, f)

- To get \( f : A \rightarrow B \) we need:
  \( t : A \rightarrow (A \rightarrow B) \rightarrow B \)

- To ensure calls are done on smaller instances:
  \( t : \forall x : A, (\forall y : A, y < x \rightarrow B) \rightarrow B \)

- Generalisation to \( f : \forall x : A, P(x) \)
  \( t : \forall x : A, (\forall y : A, y < x \rightarrow P(y)) \rightarrow P(x) \)
Well-founded fixpoint in Coq

Coq? Check Fix.

Fix

: forall (A : Type) (R : A -> A -> Prop),
well_founded R ->
forall P : A -> Type,
(forall x : A, (forall y : A, R y x -> P y) -> P x) ->
forall x : A, P x

Property of Fix

Coq? Check Fix_eq.

Fix_eq

: forall (A : Type) (R : A -> A -> Prop) (Rwf : well_founded R)
(P : A -> Type)
(F : forall x : A, (forall y : A, R y x -> P y) -> P x),
(forall (x : A) (f g : forall y : A, R y x -> P y),
(forall (y : A) (p : R y x), f y p = g y p) -> F x f = F x g)
forall x : A,
Fix Rwf P F x = F x (fun (y : A) (_ : R y x) => Fix Rwf P F y)
Accessibility

How is it possible?

- The fixpoint is on the proof of well-foundness of the relation.

Coq? Print Acc.

\[
\text{Inductive Acc (x : A) : Prop :=}
\begin{align*}
\text{Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x}
\end{align*}
\]

Coq? Print Acc_inv.

\[
\begin{align*}
\text{Acc_inv} =
\text{fun (x : A) (H : Acc R x) =>}
\text{match H with | Acc_intro H0 => H0 end}
\text{: forall (x : A), Acc R x -> forall y : A, R y x -> Acc R y}
\end{align*}
\]

When \( H : \text{Acc } R \ x \), \( \text{Acc}\_\text{inv} \ x \ H \) is structurally smaller than \( H \).

Coq? Print Fix_F.

\[
\begin{align*}
\text{Fix_F (P : A -> Type)}
\text{(F : forall x : A, (forall y : A, R y x -> P y) -> P x) =}
\text{fix Fix_F (x : A) (a : Acc R x) {struct a} : P x :=}
\text{F x (fun (y : A) (h : R y x) => Fix_F y (Acc_inv a h))}
\end{align*}
\]
Summary

- Fixpoint in CoQ are syntactically restricted but expressive.
- High-level tools help define recursive functions and reason on them: Program, Function...
- Types may contain logical parts.
- Functions in CoQ are computable, may help doing proofs.
- Extraction of Ocaml or Haskell.
- Formally proved versions of tools developed this way:
  - Compilers (C, lustre)
  - Verifiers (prover traces, register allocation, LR-automata...)
  - Kernel of an SMT solver (alt-ergo)
Outline

- Introduction
- Basics of Coq system
- Using simple inductive definitions
- Functional programming with Coq
  - Recursive functions
  - Partiality and dependent types
  - Simple programs on lists
  - Well-founded induction and recursion
  - Imperative programming
Monads to handle non functional behavior

Used in Haskell to simulate imperative features.

A monad is given by

- \( \text{comp} : \text{Type} \rightarrow \text{Type} \),
  \( \text{comp} \, A \) represents computations of type \( A \);
- \( \text{return} : A \rightarrow \text{comp} \, A \) (aka \text{unit})
  \( \text{return} \, v \) represents the value \( v \) seen as the result of a computation;
- \( \text{bind} : \text{comp} \, A \rightarrow (A \rightarrow \text{comp} \, B) \rightarrow \text{comp} \, B \),
  \( \text{bind} \) passes the result of the first computation to the second one.

Syntax: “do p <- e1; e2” means \( \text{bind} \, e_1 \, (\text{fun} \, p \Rightarrow \, e_2) \).

See Monads.v
Memory representation

- The previous monadic approach for states does not consider aliases in programs.
- Functional representation of memory and addresses:

  See LinkedLists.v
Representation of memory

Coq? Definition adr := option Z.

Coq? Definition null : adr := None.

Coq? Record node : Type
   := mknode { value : nat ; next : adr}.

Coq? Definition heap := Z -> option node.

Coq? Definition val (h : heap) (a : adr) : option node
   := match a with None => None | Some z => h z end.

The state of the program will be the heap.
Alternative representation of memory

More static separation (model à la Burstall-Bornat)

Coq? Definition alloc := Z → bool.

Coq? Definition value_m := Z → nat.

Coq? Definition next_m := Z → adr.

Coq? Definition val (h:alloc) (vm:value_m) (nm:next_m) (a:adr) : option node
Coq? := match a with None => None
Coq? | Some z =>
Coq? if h z then Some (mknode (vm z) (nm z))
Coq? else None
Coq? end.
Frame properties

- Need to specify logically validity of addresses, separation
- Separation logic can be encoded
- See Ynot (Harvard, Morrisett) : a COQ library to reason on imperative programs with separation logic.