Proof Assistants
Functional Programming 2

Bruno Barras, Christine Paulin-Mohring

INRIA, Université Paris-Sud 11

11/01/10
Overview

Program Certification
   Structural scheme
   General scheme

Monads

Modules

Tactics
   Automated tactics
Correspondance between:

- Recursive definition schemes
- Induction schemes

Introduce ase the right principle for each definition.

Basic schemes are introduced for an inductive definition $I$:

- Minimality principle if $I$ is in sort $\text{Prop}$.
- Induction principle if $I$ is in sort $\text{Set}$ or $\text{Type}$.
- No principle for mutual or nested inductive definitions.
Complex inductive definitions

Inductive term : Type :=
  Var: nat->term | App: nat->lterm->term
with lterm : Type := Emp | Add: term->lterm->lterm.

term_ind : forall P : term -> Prop,
  (forall n, P (Var n))->(forall n l, P (App n l))
-> forall t:term, P t

Mutual recursion:

Scheme term_lterm := Induction for term Sort Prop
with lterm_term := Induction for lterm Sort Prop.
term_lterm : forall (P:term->Prop)(Q:lterm->Prop),
  (forall n, P (Var n)) -> (forall n l, Q l->P (App n l))
-> Q Emp ->(forall t, P t -> forall l, Q l->Q (Add t l))
-> forall t, P t
### Nested inductive definitions

Inductive term : Type :=

Definition of the full induction principle:

Fixpoint term_lterm (t:term) : P t :=
  match t return P t with
  Var n => Pv n
  | App n l => Pa n l
  ((fix lterm_term (l:list term): Q l :=
    match l return Q l with
    nil => Qn
    | t::lt => Qc t lt (term_lterm t) (lterm_term lt) end) l)
end.
Ad-hoc schemes

Example of non-recursive definition by cases:

Definition \( f(x: I) : A \) := match \( x \) with
\[ p_1 \Rightarrow t_1 \ldots \mid p_n \Rightarrow t_n \]
end.

Associated principle:

- **Case-analysis:**
  \[ \forall P : I \rightarrow \text{Prop}, (\forall x_1^\vec P p_1) \rightarrow (\forall x_2^\vec P p_2) \rightarrow \cdots \rightarrow (\forall x_n^\vec P p_n) \rightarrow \forall x : I, P x \]

- **Analysis of the function:**
  \[ \forall P : I \rightarrow A \rightarrow \text{Prop}, (\forall x_1^\vec P p_1 t_1) \rightarrow (\forall x_2^\vec P p_2 t_2) \rightarrow \cdots \rightarrow (\forall x_n^\vec P p_n t_n) \rightarrow \forall x : I, P x (f x) \]
Building the induction principles

Experimental tools dealing with a certain class of structural recursive functions.

- Principle produced by:
  
  **Functional Scheme** \( \text{name} \) := **Induction for** \( \text{function Sort s} \).

- Principle used with:
  
  **elim term using** \( \text{name} \).

- Tactic: **functional induction** \( \text{function arg} \)
  
  applies the functional principle associated to **function** at argument \( \text{arg} \).
The **Function** tool

Function insert (a:A) (l:list A) {struct l} : list A :=
mismatch l with nil => nil
| cons b m => if leb a b then a::b::m
else b::insert a m
end.

- The same function viewed both as a relation \(R_{insert}\) and as an algorithm \(insert\).
- correspondance:

  \[
  R_{insert_complete}: \forall (a:A) \ (l \ res:list A), \quad R_{insert} \ a \ l \ res \rightarrow \ res = insert \ a \ l. 
  \]

  \[
  R_{insert_correct}: \forall (a:A) \ (l \ res:list A), \quad res = insert \ a \ l \rightarrow R_{insert} \ a \ l \ res. 
  \]

- \(insert\_equation\): proof of the fixpoint equation
- \(insert\_ind, (_rec, _rect)\): induction principles associated to the function.
General recursion

Using measures:

Require Export Recdef.

Definition mes (l: list A*list A) : nat :=
    let (l1,l2):= l in length l1 + length l2.

Function merge (l:list A*list A) {measure mes} : list A :=
    match l with
    (nil,l2) => l2
    | (l1,nil) => l1
    | (a::m1,b::m2) => if le a b then a::merge (m1,b::m2)
    else b::merge (a::m1,m2)
    end.

... tactics ...

Defined.

Generation of proof obligations to show that measure decreases.
General Recursion

Using well-founded orders.

Definition \( Rm \) (l m : list A*list A) :=
let (l1,l2):= l in let (m1,m2):= m in
  length l1 < length m1 \/
  (length l1 = length m1 \x\ length l2 < length m2).

Function merge (l:list A*list A) {wf Rm} : list A :=
match l with
  (nil,l2) => l2
| (l1,nil) => l1
| (a::m1,b::m2) => if le a b then a::merge (m1,b::m2)
  else b::merge (a::m1,m2)
end.

Construction by iteration and a termination proof.
Overview

Program Certification
  Structural scheme
  General scheme

Monads

Modules

Tactics
  Automated tactics
Monads

Principles

- A systematic way to represent imperative features in a purely functional language.
- Heavily used in Haskell.
- Also used to encode imperative programs in Why.
Monad basics

- Imperative features: mutable references, exceptions, continuations, IOs, random numbers...
- An imperative program of type $A$ is transformed (systematically) into a purely functional program in an associated type $M(A)$.
- $M(A)$ represents a type of computations; the evaluation of a program of type $M(A)$ produces an effect and (under some modality) a value of type $A$.
- Two key operators:
  - $\text{unit} : A \rightarrow M(A)$ injects pure (i.e. effect-free) values into computations
  - $\text{bind} : M(A) \rightarrow (A \rightarrow M(B)) \rightarrow M(B)$ composes two effects, the latter parameterized by the pure value of the former ($\text{bind} \ a \ \lambda x. b \approx \text{let} \ x = a \ \text{in} \ b$).
- Algebraic properties: $\text{bind} (\text{unit} \ a) \ f = f \ a$, …
Mutable references (state monad):
- **State** $S$ (with access and update functions)
- $M(A) := S \rightarrow A \times S$
- **unit** $a := \text{fun } s \Rightarrow (a, s)$
- **bind** $c f := \text{fun } s \Rightarrow \text{let } (a, s_1) := c \text{ in } f a s_1$
- $!x := \text{fun } s \Rightarrow \text{access } s \ x$
- $(x := c) := \text{fun } s \Rightarrow \text{let } (a, s_1) := c \text{ in } (a, \text{update } s_1 \ x \ a)$
Monad examples

Exceptions:

- $M(A) := \text{option } A$
- unit $a := \text{Some } a$
- bind $c f := \text{match } c \text{ with } \text{Some } a \Rightarrow f a \mid \text{None } \Rightarrow \text{None}$
- fail := None
- try $c \text{ with } \text{fail } \Rightarrow d := \text{match } c \text{ with } \text{Some } a \Rightarrow \text{Some } a \mid \text{None } \Rightarrow d$
Monad examples

Random generators:
Special case of the state monad where the state is an infinite stream of independent coin flips.

- $M(A) := B^\infty \to A \times B^\infty$
- $\text{flip} := \text{fun } s \Rightarrow \text{let } (b, s_1) := s \text{ in } (b, s_1)$

Continuations:

- $M_C(A) = (A \to C) \to C$
- $\text{unit } a := \text{fun } h \Rightarrow h a$
- $\text{bind } c \ f := \text{fun } h \Rightarrow c (\text{fun } a \Rightarrow f a h)$
Overview

Program Certification
   Structural scheme
   General scheme

Monads

Modules

Tactics
   Automated tactics
Modules

- Extension of OCaml’s module system to type theory.
  - Module expressions are not first-order (i.e. not mere records).
  - Interfaces contain abstract or manifest definitions.
  - Modules can be parameterized (functors).
- General metatheory done by J. Courant (98)
- First implementation by J. Chrząszcz (02)
- Second implementation by E. Soubiran (10)
- Also deals with non-logical aspects: notations, Hint databases, Extraction are compatible with the module system.
General principles

Interface

Module Type Modulename.
Variable var : type.
Definition def : type := term.
Declare Module mod : Moduletype.
End Modulename.

Implementation

Module Modulename.
Variable var : type.
Definition def : type := term.
Module mod := modterm.
End Modulename.
Using modules

Namespaces:
- Access to elements of a module: `Modulename.varname`
- Opening modules: `Export Modulename.` to allow direct access to `varname`.

2 ways to constrain the type of a module:
- Checking compatibility:
  
  ```
  Module SetoidBool <: Setoid.
  ```

- Hiding implementation details:
  ```
  Module SetoidBool : Setoid.
  ```
Example

An interface:

```plaintext
Module Type Order.
Variable A:Type.
Variable leb:A→A→bool.
End Setoid.
```

An implementation of this interface:

```plaintext
Module OBool <: Order.
Definition A:=bool.
Definition leb (x y:A) := if x then y else true.
End OBool.
```
Mixing objects and proofs

Module Type Order.
Variable A:Type.
Hypothesis lebrefl : forall x, leb x x = true.
Hypothesis lebtrans : forall x y z,
    leb x y = true -> leb y z = true -> leb x z = true.
End Order.

Module Type Setoid.
Variable A:Type.
Hypothesis eqbrefl : forall x, eqb x x = true.
Hypothesis eqbsym : forall x y, eqb x y = eqb y x.
Hypothesis eqbtrans : forall x y z,
    eqb x y = true -> eqb y z = true -> eqb x z = true.
End Setoid.

Properties declared in the interface proved in the implementation.
Functors

Definition of a functor:

Module SetofOrder (O:Order) : Setoid.
Export O.
Definition A:=A.
Definition eqb (a b:A) := if leb a b then leb b a else false.
Lemma eqb_refl : forall x, eqb x x = true.
unfold eqb; intros; rewrite (lebrefl x); simpl; auto.
Save.
Lemma eqbsym : forall x y, eqb x y = eqb y x.
... Save.
Lemma eqbtrans : forall x y z,
  eqb x y = true -> eqb y z = true -> eqb x z = true.
... Save.
End SetofOrder.
Example: cartesian product

Module ProdOrder (O1 O2:Order) : Order.
Definition leb (x y:A) :=
  let (x1,x2):=x in let (y1,y2):=y in
  if O1.leb x1 y1 then O2.leb x2 y2 else false.
Lemma lebrefl : forall x, leb x x = true.
...
Lemma lebtrans : forall x y z,
  leb x y = true -> leb y z = true -> leb x z = true.

Possibility to export information

Module ProdOrder (O1 O2:Order)
Overview

Program Certification
  Structural scheme
  General scheme

Monads

Modules

Tactics
  Automated tactics
Automated tactics

- Any tactic is a strategy that ultimately builds a proof term that is re-checked by the kernel of Coq.

- Several automated tactics of Coq:
  - discriminate: proves inequalities between constructors;
  - auto: resolution based depth first proof search using a lemma database;
  - intuition: decomposition in the propositional fragment;
  - omega: Pressburger arithmetic;
  - ring: simplification of expressions in a ring structure

- Tactics are composed by:
  - sequence: \texttt{tac1; tac2}
  - iteration: \texttt{repeat tac}
  - trial: \texttt{try tac}

- Two notions of efficiency: during proof search and during re-checking.
Tactic language

Ltac designed by D. Delahaye

- A way to write complex tactic without linking code to the implementation of Coq (in OCaml).

- An Ad-hoc language:
  - A higher-level pattern-matching operator applied to the goal (non-linear patterns)

```latex
match goal with
  id:?A \ / \ ?B |- ?A => case id; trivial
  | _ => idtac
end.
match goal with |- context[?a+0] => rewrite ...
```

- Specific notion of backtracking
  - Patterns are tried in order as long as the right hand-side fails

- Other specific constructions: fresh name, type of term ...