Seminar on Dependently Typed Programming

Wouter Swierstra

16-05-2012

Verification projects: tips & tricks

Use libraries

The Coq Standard Library

Here is a short description of the Coq standard library, which is distributed with the system. It provides a set of modules directly available through the `Require Import` command.

The standard library is composed of the following subdirectories:

Init: The core library (automatically loaded when starting Coq)
  Notations Datatypes Logic Logic_Type Peano Specif Tactics Wf (Prelude)

Logic: Classical logic and dependent equality
  SetIsType Classical_Pred_Set Classical_Pred_Type Classical_Prop Classical_Type (Classical) ClassicalFacts Decidable Eqdep_dec EqdepFacts Eqdep JMeq ChoiceFacts RelationalChoice ClassicalChoice ClassicalDescription ClassicalEpsilon ClassicalUniqueChoice Berardi Diaconescu Hurkens ProofIrrelevance ProofIrrelevanceFacts ConstructiveEpsilon Description Epsilon IndefiniteDescription FunctionalExtensionality

Structures: Algebraic structures (types with equality, with order, ...). DecidableType* and OrderedType* are there only for compatibility.
  Equalities EqualitiesFacts Orders OrdersTac OrdersAlt OrdersEx OrdersFacts OrdersLists GenericMinMax DecidableType DecidableTypeEx OrderedType OrderedTypeAlt OrderedTypeEx

Bool: Booleans (basic functions and results)
  Bool BoolEq DecBool IfProp Sumbool Zerob Bvector

Arith: Basic Peano arithmetic
  Arith_base Le Lt Plus Minus Mult Gt Between Peano_dec Compare_dec (Arith) Min Max MinMax Compare Div2 EqNat Euclid Even Bool_nat Factorial Wf_nat NatOrderedType

NArith: Binary positive integers
  BinPos BinNat (NArith) Pnat Nnat Ndigits Ndist Ndec NOrderedType Nminmax POrderedType Pminmax

ZArith: Binary integers
  BinInt Zorder Zcompare Znat Zmin Zmax Zminmax Zabs Zeven auxiliary ZArith_dec Zbool Zmisc Wf_Z Zhints (ZArith_base) Zcomplements Zsqrt Zpow_def Zpower Zdiv Zlogarithm (ZArith) Zgcd_alt Zwf Znumtheory Int ZOdiv_def ZOdiv Zpow_facts ZOrderedType Zdigits

QArith: Rational numbers
  QArith_base Qabs Qpower Qreduction Qring Qfield (QArith) Qreals Qcanon Qround QOrderedType Qminmax

Numbers: An experimental modular architecture for arithmetic

Prelude:

Use search automation

- SearchAbout
- SearchPattern

Sections

```
Section List.
  Variable a : Type.
  Fixpoint rev : list a -> list a
    := ...
End Section.
Check rev.
```

Tactic combinators

- idtac
- fail
- composition ;
- or-else ||
- try
- now

idtac and fail

- idtac – the identity tactic. It doesn't change the goal or context.
- fail – the failure tactic. It immediately causes the current proof attempt to fail.
- Both can be useful when writing larger, composite tactics.

Tactic composition

- You can chain together two tactics using a semi-colon:

```
induction n; simpl.
```

- Note that the simpl tactic is applied to *all* subgoals generated by the induction tactic.

Composition

- You can also use intro-pattern notation to apply certain tactics to subgoals:

```
induction n;
   [reflexivity | simpl].
```

- If you leave a 'branch' empty, Coq will apply idtac to that branch.

Or-else

- The tactic `a || b` tries to apply the tactic `a`, and if that fails, applies tactic `b`

```
induction n; reflexivity || idtac.
```

- This last expression may also be written as

```
induction n; try reflexivity.
```

Now

- Finally, the `now t` tactic is defined as

```
t; easy
```

- Where the `easy` tries to apply symmetry, reflexivity, and some assumption.
- If the goal is not solved, the tactic fails.

Good style

- Tactics that generate more than one subgoal should either:
  - immediately close all but one subgoal:
    `induction n; [ reflexivity | ].`
  - or use Case annotations/comments/bullets/indentation to distinguish what you are proving.
- Use `now` to close a subgoal or fail.

Good style

- If you can, try to keep one subgoal open at all times.
- If this doesn't work, try to make it blatantly obvious how many subgoals you expect to have open and which one you are proving.
- This will make your proof scripts *much easier to maintain.*

Exercise:

Given:

```
Fixpoint double (n : nat) :=
  match n with | O => O
               | S k => S(S(double k))
          end
```

Prove that, forall n and m:

```
double n = double m -> n = m
```

Example

```
Lemma double_inj (n m : nat) (H : double n = double m) : n = m.
  Proof.
    generalize H; generalize m; clear m H; induction n.
      (* Base case *)
      intros m H; destruct m as [ | k]; [reflexivity | discriminate].
      (* Inductive step *)
      intros m H; destruct m as [ | k]; [discriminate | ].
      f_equal; apply IHn; unfold double in *; now omega.
  Qed.
```

Ltac

- Coq also has a tactic-programming language called *Ltac*.
- It let's you write complex composite tactics, pattern match on the goal or context,

Simple Ltac

```
Ltac destruct_all t :=
 match goal with
  | x : t |- _ => destruct x; destruct_all t
  | _ => idtac
 end.
```

Example

```
Theorem all3_spec : forall b c : bool,
    orb
      (andb b c)
      (orb (negb b)
               (negb c))
  = true.
Proof.
  intros; destruct_all bool; reflexivity.
Qed.
```

Complex example

```
Ltac easy :=
  let rec use_hyp H :=
    match type of H with
    | _ /\ _ => exact H || destruct_hyp H
    | _ => try solve [inversion H]
    end
  with do_intro := let H := fresh in intro H; use_hyp H
  with destruct_hyp H := case H; clear H; do_intro; do_intro in
  let rec use_hyps :=
    match goal with
    | H : _ /\ _ |- _ => exact H || (destruct_hyp H; use_hyps)
    | H : _ |- _ => solve [inversion H]
    | _ => idtac
    end in
  let rec do_atom :=
    solve [reflexivity | symmetry; trivial] ||
    contradiction ||
    (split; do_atom)
  with do_ccl := trivial with eq_true; repeat do_intro; do_atom in
  (use_hyps; do_ccl) || fail "Cannot solve this goal".
```

Fancy tactics

- omega – solver for Presburger arithmetic.
- ring – proves equalities between rings (such as nat,+,*).
- intuition – proves tautologies
- ... and many more

Writing programs with tactics

- You can also write (simple) programs with tactics. For example:

```
Definition id (a : Type) : a -> a.
  intros x. apply x.
  Qed.
```

Writing proofs with programs

```
Definition id (a : Type) : a -> a.
  exact (fun x => x).
  Qed.
```

- The `exact` tactic lets us provide an exact proof for a (sub)goal.

Writing programs with tactics

```
Definition id (a : Type) : a -> a.
  refine (fun x => _).
  apply x.
  Qed.
```

- The `refine` tactic lets us provide parts of a proof for a (sub)goal.
- Any underscores result in subgoals that still need to be filled in.

Example

```
Definition wrong_pred (n : nat) : nat :=
  match n with
    | 0 => 0
    | S k => k
  end.
```

Example

```
Definition pred (n : nat) (H : n > 0) : nat :=
  match n with
    | 0 =>(* impossible! *)
    | S k => k
  end.
```

So how can we write this program?

Tactics or terms?

- Should I use tactics or provide precise proof terms?
- Tactics are usually easier and more flexible.
- ... but give you less control over the resulting term.
- It can be useful to mix both styles.

Verification projects

- No lectures next week.
- Instead, I want to meet everyone to discuss their project on Thursday morning.
- I'll put a schedule online.
- Prepare for this meeting.