Reflexive Tactics

From: Introduction to the COQ Proof-Assistant for Practical Software Verification (by Christine Paulin-Mohring)

Timothy Fräser

Ltac

- by David Delahaye

- language for creating complex tactics without ML code

  COQ has a functional CAML kernel

- combining tactics in Ltac can be inefficient and create large proof terms

- another idea is to program a tactic inside COQ

Why use Reflexive Tactics?

- to proof a property `P:Prop`
- given a mechanical way of proving *P* based on it's structure
- problem: cannot reason directly about the structure of *P* inside COQ

How to use Reflexive Tactics?

- can reason about the structure of inductive types (data)
- so we represent *P* as a term having an inductive type *D*
  - called reification
  - ex: an Abstract Syntax Tree

Algorithms

- Every property *P* needs a data representation `d:D`

- `d2P : D → Prop`
  - interpretation of the data-type (converts `d:D` to *P*)

- `d2b : D → bool`
  - given mechanical way of proving P based on it's structure
  - `d2b` needs to be `correct`
  - `d2b` needs to be efficient.

- `correct : ∀d:D, d2b d = true → d2P d`

Convertibility Rule

- important rule of COQ theory

$$\frac{\Gamma \vdash U : s \qquad \Gamma \vdash t : T \qquad T \equiv U}{\Gamma \vdash t : U}$$

- computation (for $T \equiv U$) becomes part of type checking
- termination: important to keep decidability of type checking
- compatible with all languages having possible computations in their terms

Reflexifity “Rule”

- convertibility rule:

$$\frac{\Gamma \vdash \mathtt{U:s} \qquad \Gamma \vdash \mathtt{t:T} \qquad \mathtt{T} \equiv \mathtt{U}}{\Gamma \vdash \mathtt{t:U}}$$

- reflexifity "rule":

$$\frac{\mathtt{refl\_eq : true = true} \qquad \mathtt{d2b\ d} \equiv \mathtt{true}}{\mathtt{refl\_eq : d2b\ d = true}}$$

- provability completely depends on convertibility (*d2b d ≡ true*)

Example 1: from data d to Property P

```
(* data-type *)
Inductive form : Set :=
    | T
    | F
    | Var  : nat  -> form
    | Conj : form -> form -> form.


(* environment for un-interpretable sub-propositions *)
Definition env := list Prop.
Fixpoint find_env (e:env) (n:nat) :=
    match e with
      nil       => True
    | cons x xs => match n with
                    0   => x
                  | S p => find_env xs p
                  end
    end.
```

Example 1: from data d to Property P

```
(* data-type -> P *)
Fixpoint d2P e (f:form) {struct f} : Prop :=
    match f with
      T          => True
    | F          => False
    | Conj p q => d2P e p /\ d2P e q
    | Var n      => find_env e n
    end.

Notation "x :: xs" := (cons x xs).

(* compute data-type -> P *)
Definition e := (True :: False :: (0=0) :: nil).
Eval compute in
    (d2P e (Conj (Var 0) (Conj (Var 2) (Var 1)))).

(* outputs: "= True /\ 0 = 0 /\ False : Prop" *)
```

Example 2: Reification

```
(* compute environment from formula *)
Ltac env_form l f :=
    match f with
      True      => constr:(l,T)
    | False     => constr:(l,F)
    | ?A /\ ?B => match  env_form l  A with (?l1,?A1) =>
                      match env_form l1 B with (?l2,?A2) =>
                        constr:(l2, Conj A1 A2)
                     end
                    end
    | ?A        => let n := eval compute in (length l)
                      in constr:(cons A l, Var n)
    end.
```

Example 2: Reification

```
(* P -> data-type (reify) *)
Ltac reify :=
    match goal with |- ?P =>
      match (env_form (nil (A:=Prop)) P) with
        (?l,?f) => let e := eval compute in (rev l)
                     in change (d2P e f)
      end
    end.

(* compute P -> data-type (reify) *)
Lemma test1 : 0=0 /\ False -> False /\ 1=1 /\ (0=0).
reify.

(* outputs:
1 subgoal
  ============================
   d2P ((0 = 0)::(False -> False)::(1 = 1)::(0 = 0)::nil)
       (Conj (Var 0) (Conj (Var 1) (Conj (Var 2) (Var 3))))
*)
```