Abstract

Coq 8.5 has a number of new features. It has more powerful universe polymorphism support. It allows tactics to be run at interpretation to construct other terms. The ability to switch from Gallina to Ltac in arbitrary locations nicely complements the `constr:` notation permitting the switch from Ltac to Gallina in tactics, and opens up many new possibilities. I propose to present three tricks involving these new features: tactics in terms allows the construction of tactics that recurse under binders; tactics in terms together with typeclasses allows overloading notations based on the type of their arguments; and there is a way to talk about universe levels explicitly, helped along by tactics in terms.

Tactics in Terms

Coq 8.5 features tactics in terms using the syntax `$(...)$` to denote a tactic script, which evaluates to the proof tree it builds in proving a goal. For example, we can simplify terms generated by tactics before saving them.

```
Definition I_am_one : nat := $(let x := constr:(2 - 1) in exact x)$.
Print I_am_one.
(* I_am_one = 2 - 1 : nat *)
Definition I_am_exactly_one : nat := $(let x := constr:(2 - 1) in
                                       let y := (eval compute in x) in
                                       exact y)$.
Print I_am_exactly_one.
(* I_am_exactly_one = 1 : nat *)
```

Recursive Tactics

The ability to go back and forth between Gallina and Ltac at will allows tactics that recurse under binders, by using Gallina to add binders and Ltac to recurse. Here is an example that returns the left projection of an `and` under binders:

```
Ltac ret_and_left f :=
  let tac := ret_and_left in
  let T := type of f in
  lazymatch eval hnf in T with
    | ?a ∧ ?b ⇒ exact (proj1 f)
    | ?T' → _
      ⇒ let ret := constr:(λ x' : T' ⇒ let fx := f x' in $(tac fx)$) in
         let ret' := (eval cbv zeta in ret)  in
         exact ret'
    end.
Goal ∀ A B : Prop, (A → A → A ∧ B) → True.
  intros A B H.
  pose $(ret_and_left H)$.
  (* λ x' x'0 : A ⇒ proj1 (H x' x'0) : A → A → A *)
```

This code works around some bugs in the current implementation, such as the fact that the tactic currently being defined is not available inside of `$(...)$`, and binders added within `constr:` are not renamed when going under binders.

Overloading Notations on Argument Types

Coq provides a way of overloading notations via scopes; together with binding scopes to types, this allows reusing the same notation to mean similar but different things when you are expecting terms of different resulting types. Using typeclasses, we can extend this overloading to a dependence on the type of the arguments, though this breaks rewriting. Using tactics in terms (in notations), we can fix rewriting. Due to limitations in the current trunk version of Coq, this breaks using evars in such notations.[1] This is useful, for example, if you want to define both horizontal and vertical composition using the same symbol, which both return the same type of morphism, but have different argument types. Here is a very simple example:

```
Parameters T1 T2 T3 : Type.
Parameter F : T1 → T3.
Parameter G : T2 → T3.
Parameters (t1 : T1) (t2 : T2).

Class MyNotation {A R} (a : A) (r : R) := {}.
Definition mynotation A R a r `{@MyNotation A R a r} := r.
Instance MyF x : MyNotation x (F x) | 10.
Instance MyG x : MyNotation x (G x) | 100.

Notation "% x"
    := ($(let ret := constr:(@mynotation _ _ x y _) in
          let ret' := (eval cbv beta delta [mynotation] in ret) in
          exact ret')$)
  (at level 40, only parsing).
Check (% t1). (* F t1 : T3 *)
Check (% t2). (* G t2 : T3 *)
```

If we wanted to display these notations as well, we could add extra notations for each constructor, for example,

```
Notation "% x" := (F x).
Notation "% x" := (G x).
```

above the general notation above.

Explicit Universe Levels

The final trick involves the new, more powerful universe polymorphism algorithm that Matthieu Sozeau has recently implemented. Although Coq still uses typical ambiguity and gives no explicit way to talk about universes, it is possible to define a `Lift` function which takes a type and returns the same type in a higher universe. It is possible to use this to prove, for example, than if one universe satisfies functional extensionality, then so do all lower universes; that theorem is useful for theorems that end up needing multiple instances of functional extensionality at various universe levels to avoid universe inconsistencies. Here is the definition of `Lift`, and an associated lift for equality. (The lift for equality only works correctly if `eq` is polymorphic.)

```
Set Universe Polymorphism.
Inductive eq {A} (x : A) : A → Set := eq_refl : eq x x.
Notation "x = y :> A" := (@eq A x y) : type_scope.
Notation "x = y" := (x = y :> _) : type_scope.
Definition Lift
 : $(let U1 := constr:(Type) in
     let U0 := constr:(Type : U1) in
     let U0' := (eval simpl in U0) in
     exact (U0' → U1))$
    := λ A ⇒ A.
Definition lift_eq {T x y} (p : x = y :> T) : x = y :> Lift T
    := match p with eq_refl _ ⇒ eq_refl _ end.
```

[1]`https://coq.inria.fr/bugs/show_bug.cgi?id=3278`