An abstract type for constructing tactics in Coq

Arnaud Spiwack

An abstract type for constructing tactics in Coq

Arnaud Spiwack

LIX, École Polytechnique
arnaud@spiwack.net

Abstract. The Coq proof assistant is a large development, a lot of which happens to be more or less dependent on the type of tactics. To be able to perform tweaks in this type more easily in the future, we propose an API for building tactics which doesn't need to expose the type of tactics and yet has a fairly small amount of primitives. This API accompanies an entirely new implementation of the core tactic engine of Coq which aims at handling more gracefully existential variables (aka. metavariables) in proofs - like in more recent proof assistants like Matita and Agda2. We shall, then, leverage this newly acquired independence of the concrete type of tactics from the API to add backtracking abilities.

1 Introduction

In the Coq proof assistant, the type of tactics is something along the lines of:

```
type tactic = goal → (goal list * validation)
and validation = term list → term[1]
```

That is, from a "goal" (a sequent at a leave of a proof in progress), a tactic generates a list of subgoals that need to be proven, along with a "validation" which is a function building the actual proof for the considered goal from those of the generated subgoals.

This was supposed to operate under the assumption that all goals are independent from one another. But this assumption is too strong. Experience in automated proof search shows, for instance, that when one has to prove a proposition of the form $\exists x.\ P\ x$, one needs to pose some *existential variable* u and try to prove $P\ u$ and leave the question of what exactly is this u for a later unification. This introduces dependencies between goals; to cope with it, the type of tactics was enriched with a context dealing with these existential variables (and shared between goals).

Now this is still not enough. It can be useful, especially in presence of dependent types, to be able to pose as a goal not only this $P\ u$, but also a second goal whose *proof* is u. The added complexity of this principle is that solving the former goal can solve, as a side effect, the latter. This requires more profound modifications.

Modifying the type of tactics in a non trivial way impacts code throughout the sources of Coq. If most of the tactics are built out of combinators – and

[1] This is a simplification of the actual type which will be sufficient for the present discussion

are therefore rather robust with respect to details – almost all of them use explicitly the fact that tactics are functions whose first argument is a goal. This is not necessarily incompatible with a more fine grain treatment of existential variables, but it does not allow for other features like multiple goal tactics[2] or backtracking.

2 Refinement

A perk of having an unrestricted treatment, in proofs and tactics, of existential variables, is that a term with existential variables is isomorphic to a partial proof derivation. Therefore we need only one atomic tactic – called *refine*[3] – which reads a term as a partial proof derivation[4]. Whereas formerly, there was a need of a fairly large set of core tactics mimicking the derivation rules of the system. Additionally it is noteworthy that, as Agda2 [**?**] has demonstrated, refinement is a quite useful tool to write dependently typed programs. We also provide a few bureaucratic atomic tactics (such as clear and move)which implement explicit structural rules and are mostly useful for controlling automatic tactic and have little mathematical values, so we will forget about them here.

To sum up, at the most atomic level, we are given an intuitionistic sequent (a goal), and to advance in our proof, we provide a term with existential variables, which we read as a partial proof derivation, and we get back a list of new sequents (subgoals). The partial proof we provide depends on the sequent: typically to prove a sequent like $x : A \vdash A$, we would use the term x, which only makes sense in the context of this particular sequent. We shall call such a term *goal sensitive.* Goal sensitive values are represented as a type:

```
type 'a sensitive
```

This type will stay abstract, meaning that only a few primitives know of its concrete representation. Its concrete representation can be something like a function taking as argument a goal and a context for existential variables, and returning a value of type 'a. Among the needed primitives, we need access to the content of the goal. This is provided, mostly, by the following two primitives:

```
val concl : types sensitive
val hyps : named_context sensitive
```

where a named_context is, essentially, an association list from variable names to types. Those two primitives combined say that a goal sensitive value has access to an intuitionistic sequent, much what was expected. Because of specifics of Coq, we will also need a variant of hyps:

```
val env : env sensitive
```

which returns a full environment, containing, among other things, the hypotheses of the current goal.

[2] Multiple goal tactics have been implemented, independently, in Matita [**?**].

[3] Coq has already a refine tactic which works a bit like that, however it is not primitive, and doesn't have the same power than "real" refinement.

[4] This is, *mutatis mutandi*, equivalent to what Lengrand&*al.* propose in the setting of Pure Type Sequent Calculus [**?**].

```
(* evar is the type of existential variables *)
type goal = evar
(* evar_map is the type of context for existential variables.
We use a reference to an evar_map because we need to be
able to modify it in later primitives *)
type 'a sensitive = goal → env → evar_map ref → 'a

let return x _ _ _ = x
let (>-) s k goal env rdefs =
  k (s goal env rdefs) goal env rdefs

let concl goal _ rdefs =
  Evd.evar_concl (Evd.find goal !rdefs)
let hyps goal _ rdefs =
  Evd.evar_concl (Evd.find goal !rdefs)
let env goal env rdefs =
  let hyps = hyps goal env rdefs in
  Environ.reset_with_named_context hyps env
```

Fig. 1. Implementation of some primitives for goal sensitive values

Now, of course, we need a way to build on these primitives to make new goal sensitive values; for instance the number of hypotheses of the sequent is expected to be an `int sensitive`. For that purpose we introduce a monad on the goal sensitive values:

```
val return : 'a → 'a sensitive
val (>-) : 'a sensitive → ('a → 'b sensitive) → 'b sensitive
```

where `return x` is to be understood as the same as `x` but pretending to depend on a goal and `(>-)` "propagates" the goal under consideration. Figure 1 gives an implementation of these primitives. To illustrate, here is the code that computes the number of hypotheses:

```
let hcount =
  hyps >- fun hs →
  return (List.length hs)
```

We can, finally, move to disclose the type of the refinement tactic:

```
val refine : refinable → subgoals sensitive
```

Let us take it apart bit by bit. First it returns a goal sensitive value, as expected. This value has type `subgoals` which is simply a list of goals, except that it is made *private*, which is an OCaml keyword to say that anyone can use an element of that type as a list of goals, but only functions local to the current module can actually construct a value of type `subgoals`. This restriction exists to prevent accidentally writing "rogue" tactics, which are not defined in terms of `refine`, and may have an incorrect behaviour.

Let us make this a little more precise by stating explicitly that values of type `subgoals sensitive` are to be used as tactics. Indeed `'a sensitive` are not only elements of type `'a` defined in term of an unknown goal, they also depend on a sufficiently rich context to express the state of a proof (mainly definitions and typing information of existential variables around, plus the current partial proofs of goals which are also dealt with in terms of existential variables), which they carry around and modify when needed, much in the spirit of the State monad in Haskell [**?**].

The other part of this is the type refinable of the argument of refine. Elements of type refinable are almost terms. In fact they are terms with extra information recording which of their existential variables are "new". Indeed, some of the existential variables can have been created for a previous refine, in which case the goal for it has already been generated, and we do not want it to be duplicated. Definition of elements of type refinable are handled by a module which reads as:

```
module Refinable : sig
 type handle

 val make : (handle → term sensitive) → refinable sensitive
 val mkEvar : handle → env → types → term sensitive
end
```

It has actually a few more functions defined (half a dozen in total), but these two suffice for this discussion. First we notice the (abstract) type handle which is a sort of registration machine, whenever we create a new existential variable (through mkEvar) we get to tell it to the handle (which is, in fact the list of existential variables which have already been built with mkEvar). The function make says that if, in the context of a handle, we can build a term (actually a goal sensitive term), then we can get a goal sensitive refinable. Note that this refinable needs to be goal sensitive, because it modifies the existential variable context – the "state" in the 'a sensitive monad – as it introduces new existential variables. Finally mkEvar registers a new existential variable to a handle, given enough type information, and returns the corresponding term.

To put all this in practice and conclude this section here is the definition of a simple introduction tactic. That is a tactic which given the name x, takes a goal of the form $\Gamma \vdash A \to B$ and turns it into $\Gamma, x : A \vdash B$.

```
let intro x =
  concl >- fun c →
  let (_,a,b) = destProd c in
  env >- fun e →
  let new_env = push_named (x, None, a) e in
  Refinable.make (fun h →
    Refinable.mkEvar h new_env b >- fun e →
    return (mkNamedLambda (x,a,e))
  ) >- fun r →
  refine r
```

Where destProd decomposes a type of the shape $A{\to}B$, push_name (x, None, a) e adds $x : A$ on top of the environment, and mkNamedLambda (x,a,e) builds the term $\lambda x{:}A.\ e$.

3 Combining tactics

3.1 An abstract approach

So far, we haven't given a way to combine existing tactics into one; in other words our tactics can have a single refinement step. While this is not a limitation in expressiveness, this lacks some amount of flexibility. The most typical tactic combinators in coq are the composition – `t1;t2` applies `t1` and then applies `t2` to all generated subgoals – and the alternative – `t1||t2` tries to apply `t1`, if it fails, it applies `t2` instead. We will introduce, on the OCaml side, two combinators `(<*>)` and `(<+>)`, respectively, to represent them.

We could actually make them act on the `subgoals sensitive` type. However this leaves little space for improvement. Let us take a small detour to see why we shall use a dedicated type for tactics that combine.

Formerly, the type of a proof in progress was a tree representing the tactics that were used in its course. This does not allow for goals that are solved by side effect, which we want to introduce. As a matter of fact it does not deal very cleanly with side effects at all, as the order in which the goals are solved does not appear in the proof. We propose a new implementation, where a proof is described by an existential variable context[5], goals being themselves described as particular existential variables. The state of the current proof, which we call a *view*, is one such context, together with some of its open[6] goal, said to be *under focus*, lined up in a list – so they can be addressed by their position.

```
type proofview
```

Following our policy, the type is abstract. Executing a tactic on a view returns another view. Now values of type `subgoals sensitive` act as tactics, for instance by applying them to one particular goal or to all the goals simultaneously. We could imagine other kinds of manipulation of views. For instance changing the order of its goals, which can be part of a tactic (for instance the `destruct` and `induction` tactics yield goals in a different order). There is no reason to restrict ourselves to tactics which can be encoded as `subgoals sensitive`-s. Therefore we shall consider a new type to represent tactics, which can be seen as being functions from views to views.

```
type tactic
```

which is, again, abstract.

Another feature we might want to add is the ability for tactics to communicate some information to the tactic that follows. We can imagine a tactic which never fails, but "returns" a boolean informing whether it progressed, or an introduction tactic which chooses a name for the new hypothesis and passes it to the following tactics. To reach that goal, we enrich the type of tactics with a type parameter:

```
type 'a tactic
```

which represent the "return type" of a tactic. As a matter of fact we can install a monad on the type of tactics.

[5] In Coq, the type for these contexts is called `evar_map`.

[6] *i.e.* not yet (partially) solved

```
type 'a tactic = proofview → ('a*proofview)

let tclUNIT a view = (a,view)
let (>=) t k view =
  let (a,view') = t view in
  k a view'
let (<*>) t1 t2 view =
  t2 (snd (t1 view))

let (<+>) t1 t2 view =
  try t1 view
  with _ → t2 view
```

Fig. 2. A first implementation of tactics

```
val tclUNIT : 'a → 'a tactic
val (>=) : 'a tactic → ('a → 'b tactic) → 'b tactic
```

It can be seen as a state monad, with a proof view as the state. Now the composition

```
val (<*>) : 'a tactic → 'b tactic → 'b tactic
```

can be viewed as a special case of `(>=)`. Finally the alternative

```
val (<+>) : 'a tactic → 'a tactic → 'a tactic
```

has to be implemented using some kind of exception mechanism. We propose, in Figure 2, an implementation of these primitives.

We also need a primitive internalising `subgoals sensitive` as tactics:

```
val tclSENSITIVE : subgoals sensitive → unit tactic
```

As expected, it produces tactic which returns `unit`, as return values are novelties of the `tactic` level. As a convention, we decide that `tclSENSITIVE t` applies `t` to every goal under focus[7]. Of course we also have primitives to manipulate focus, for instance:

```
val tclFOCUS : int → int → 'a tactic → 'a tactic
```

which focuses on a range of goals, applies the tactic argument, and then unfocuses back, effectively splicing the produced goals in place of the range it originally focused on. This effectively gives the ability to choose a particular goal and to apply a tactic to it, restoring the traditional approach of Coq.

3.2 Leveraging the abstraction barrier

With all this abstraction done, it becomes easy to change the underlying type of tactics to support new features. As a conclusion to this section, we shall describe briefly how to support backtracking in tactics. More explicitly, by backtracking, we mean the property that `(a<+>b)>=c` would be equivalent to `(a>=c)<+>(b>=c)`.

[7] As the order matters, because of side effects, we specify that it is applied from the last one to the first one. Also, goals that are closed by side effect before being considered are ignored

Which is not the case with the implementation we sketched earlier. Indeed it has the property that if a succeed, then (a<+>b)>=c is equivalent to a>=c.

There are many ways to implement backtracking. For the prototype, we have used a two-continuation type, much in the spirit of [?], except that it does not behave as a monad transformer. We give tactics the following type:

```
type 'a nb_tactic  = proofview → 'a*proofview
type 'r fk = exn → 'r
type ('a,'r) sk = 'a → 'r fk → 'r
type 'a tactic = { go :
        'r. ('a, 'r nb_tactic) sk → 'r fk → 'r nb_tactic
}
```

This deserves some explanation. 'a nb_tactics are non-backtracking tactics, as presented at the beginning of this section. 'r fk are failure continuations, they are passed an exception, that with which the previous tactic failed. ('a,'r) sk are success continuation, they are passed an element of type 'a which is the result of the previous tactic – which succeeded – and a failure continuation to know where to go if it fails. Actual tactics are wrapped inside a record because we want to universally quantify over the type argument 'r, something which, in OCaml, is only supported via records.

Now to see how it supports backtracking, we will show the definition of (>=) and (<+>).

```
let (>=) t k = { go = fun sk fk view →
  t.go (fun a fk → (k a).go sk fk) fk view
}
```

This reads: "t>=k executes t, if it succeeds and returns a it then executes (k a) with its success continuation and propagating the failure continuation t1 passes to its success continuation, otherwise it executes its failure continuation".

```
let <+> t1 t2 = { go = fun sk fk view →
  t1.go sk (fun _ → t2.go sk fk view) view
}
```

This reads: "t1<+>t2 executes t1 and continues with its success continuation. If it fails then the error is ignored and we go on with t2 with the current success and failure continuations."

The failure continuations act as backtracking stacks which are propagated by the atomic tactics. In particular if we have a tactic of the form (a<+>b)<+>c where a is atomic, its success continuation sk is passed the following failure continuation:

```
fun _ → b.go sk (fun _ → c.go sk fk)
```

If it fails it then tries b then sk, if it fails again it then tries c then sk. This is precisely what we expected. Also note that (<+>) is also associative.

With this implementation we have strayed far from simple functions from a goal to a list of goals. However we have given an API which is stable under changes of implementation (and which abstract away the complexity of the underlying implementation: working with double continuation values is fairly destructive to ones brain cells). As a matter of fact there might be a better suited way to support backtracking for our tactics, it won't be a problem to experiment in the future thanks to the abstraction layer.

4 Conclusion

The API we proposed in this article, which is part of the development branch of Coq, is actually composed of 25 primitives for the `'a sensitive` part (including the Refinable module) and 17 for the `'a tactic` part, at the time when this article has was written. This represents less than 800 lines of code. This should be compared to the roughly 80 primitives (plus a few additional primitives scattered throughout the code, since the type of tactics was not abstract) and around 2000 lines of code for the legacy core tactic machinery of Coq (the proof manipulation part has been shortened even more, through better code sharing with other parts of the code base).

It would not have been feasible, though, to port all the code base to this new API at this stage, hence we have built a compatibility layer which includes tactics with a similar type than earlier as a sub-case of `subgoals sensitive`. This layer breaks the abstraction a bit, but is still fairly maintainable. The trouble is that it didn't allow us to eliminate much of the old code for now.

To check the tractability of this API, we have hacked together a small prototype of a tactic language. In two weeks' worth of work it got convincing enough to be able to conclude that this new interface is complete enough. Of course, the design of an actual tactic language drawing on the new capabilities presented here is a question which has yet to be addressed.

References

1. O. Kiselyov, C. Shan, D.P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). *ACM SIGPLAN Notices*, 40(9):203, 2005.
2. Stéphane Lengrand, Roy Dyckhoff, and James McKinna. A focused sequent calculus framework for proof search in Pure Type Systems. *Logical Methods in Computer Science*, 2009. Accepted for publication.
3. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
4. S. Peyton-Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, et al. Haskell 98. *Journal of Functional Programming*, 13(1):0–255, 2003.
5. Claudio Sacerdoti and Enrico Tassi. A new type for tactics. *PLMMS'09*, page 22, 2009.