Procrastination

A proof engineering technique

> Have you seen those procrastinators the monks use?
> Wonderful things. They can move time, store it, stretch it...
> quite ingenious.
>
> — Terry Pratchett, *Thief of Time*

ABSTRACT

We present a small Coq library for collecting side conditions and deferring their proof.

1 INTRODUCTION

What are the different ways of proving an assertion of the form "∃ x, P x"? From the perspective of the final proof term, there is only one: providing a witness. However, from a "proof engineering" perspective, Coq enables a wider range of possible approaches thanks to its powerful feature of "evars" [5, §2.11].

We describe existing approaches for proving such a goal, and propose a new one, supported by a small library dubbed *Procrastination* [3], which itself relies on evars. We believe it captures a useful proof pattern, where one wishes to first continue with the proof of P x, in order to discover which constraints x must satisfy, before providing a value for x and proving the side conditions.

We used this library with success in our work on verifying the asymptotic complexity of programs [4].

2 PROVING EXISTENTIAL QUANTIFICATIONS IN COQ

Manually guessing a witness. The most straightforward option to prove a Coq goal "∃ x, P x" is to guess and provide upfront a witness, using the `exists` tactic. This is often a good way of providing external knowledge about how the proof should continue.

Delaying the instantiation using evars*.* However this falls short if the only intuition is that x should be "whatever makes the proof work". A trial-and-error process can be attempted, where the user alternatively adjusts the witness and the proof until it passes. This does not lead to maintainable proof scripts: in the end, only a "magic" witness remains; it may be large and duplicate information; it may also have to change in the future, following updates to the rest of the proof.

To allow delaying the instantiation of x, Coq provides a versatile feature: evars. An evar corresponds to a "hole" in the proof, and only exists during the pre-typing phase – it will not appear in the proof term, and only helps elaborating it. Using the `eexists` tactic on the goal "∃ x, P x" turns it into "P ?x", introducing a fresh evar ?x. Later, it can be discovered that this hole should be (definitionally) equal to some term: the evar then gets instantiated. This is generally performed automatically by Coq tactics through unification.

"Big enough" existentials. This technique again falls short if continuing with the proof of P ?x does not exhibit a term that ?x must be *equal* to, but e.g. *lower bounds* on ?x. Cohen [1, §5.1.4] shows that it is a common situation in analysis, where a proof is established by picking a "big enough x". During the proof, inequalities of the form "a ≤ x", "b ≤ x", ... are all trivially satisfied provided that "x is big enough".

Cohen formalizes this seemingly fuzzy reasoning in a small tactic library. On a "∃ x, P x" goal, a tactic instantiates x with the iterated maximum of a list of terms, this list being initially an evar. It is then possible to progressively populate this list, automatically discharging subgoals of the form "a ≤ x".

3 THE *PROCRASTINATION* LIBRARY

We are interested in the situation where arbitrary side conditions about x may be encountered – e.g. inequalities of both the form "a ≤ x" and "x ≤ b". We might also be trying to infer not an arithmetic value but a function, e.g. f satisfying "∀ n, f(n)≥ 1 + f(n−1)". In this situation, it seems difficult to guess the shape of x or f beforehand (e.g. as an iterated maximum).

We introduce a small Coq library named *Procrastination*, which generalizes the ideas of Cohen, and allows collecting arbitrary side conditions (to be proved later) about zero, one or several variables. It is more general than "big enough" in the sense that it can deal with arbitrary side conditions, but it does not try to solve these side conditions: this is typically done in a second time by a dedicated procedure (e.g. for "big enough" this procedure would take the running maximum of the lower-bounds collected).

In other words, the library enforces a pattern where the proof is separated in two phases: first side conditions are collected about some variables, then they are solved and the variables are instantiated. This is crucial in limiting the proliferation of evars, which tend to produce brittle proofs because they can might get incorrectly instantiated by unrelated tactics. [1]

The implementation provides reusable Ltac components, with the hope that they can help applying this technique even in more exotic scenarios. The code is freely available online [3].

4 EXAMPLE: SOLVING RECURRENCE (IN)EQUATIONS

Our initial motivation for this work was analyzing the time complexity of programs, where one has to find a solution for a set of inequations about a program's execution time. One approach is using Cormen *et al.*'s substitution method [2, §4]. The idea is to guess a parameterized *shape* for the solution; substitute this shape into the goal; gather a set of constraints the parameters must satisfy for the goal to hold; finally, show that these constraints are indeed satisfiable.

We demonstrate this strategy using *Procrastination* on cost inequations derived from a binary search program. We "guess" that the solution should be a monotonic function f with shape $\lambda n.$ if $n = 0$ then c else $(a \log n + b)$; where a, b and c are parameters, about which we wish to gradually accumulate a set of constraints.

```
Goal ∃(f: nat → nat),
  1 ≤ f 0 ∧
  monotonic f ∧
  ∀ n, 0 < n → 2 + f (n/2) ≤ f n.
Proof.
  begin defer assuming a b c. (1)
  exists (fun n ⇒ if zerop n then c else a * log2 n + b).
  repeat split.
  { simpl. (2) defer. (3) }
  { … (* c ≤ b *) defer. … }
  { intros.
    … (* 2 + c ≤ b *) defer.
    … (* 2 ≤ a *) defer. }
  end defer.
  (4) eomega.
Qed.
```

(1)
a: ?x
b: ?x0
c: ?x1
g: Group (?G a b c)
———————————————
∃ f: nat → nat,
  1 ≤ f 0 ∧ monotonic f ∧ ∀ n, 0 < n → 2 + f (n/2) ≤ f n

(2)
a, b, c: nat
g: Group (?G a b c)
———————————————
1 ≤ c

(3)
a, b, c: nat
g: Group (1 ≤ c ∧ ?Goal0)
———————————————
Subgoal solved

(4)
———————————————
∃ a b c, 1 ≤ c ∧ c ≤ b ∧ 2 + c ≤ b ∧ 2 ≤ a

The script starts with `begin defer assuming a b c` which introduces the parameters, as well as a "group" g, using an evar to collect the constraints (1). Note that a, b and c are not evars themselves, instead they appear as abstract variables. Note also that the type of these variables is inferred to be `nat` only further on in the proof script. After instantiating the function with the proposed solution, the first inequality simplifies to a condition on c (2). This condition is stored (in g) for later retrieval, using the `defer` tactic, which discharges the subgoal (3). Similarly, the other two inequalities yield side conditions about a, b and c that can be `deferred`. Finally, `end defer` retrieves the collected conditions, which the user must then prove. Here, this is done by an ad hoc user-defined tactic, `eomega`.

Observe that the final proof script never mentions the actual witnesses: we expect the proof to be robust in the face of minor changes in the cost inequations, as long as the *shape* of the solution remains valid.

[1] For example, we noticed that `ring_simplify` is particularly careless (or sometimes crashes) when the goal contains evars.

5 EXAMPLE: SIMPLIFYING RECURRENCE (IN)EQUATIONS

An other application of the approach, also coming from our work on complexity analysis, is to simplify (in)equations about a function. The inequalities constituting the goal of the example in Section 4 have in fact already been simplified. In practice, analyzing the complexity of a program yields a set of constraints C and a goal $\exists f, C(f)$. C has typically been elaborated automatically from the program, and one wants to simplify the goal into $\exists f, C'(f)$, where C' is a simpler set of constraints.

In this example we show that we can use *Procrastination* to collect constraints not only about natural numbers (as in the previous example), but also about functions. Starting from constraints (on the program execution time) that are close to what an automated program analysis would produce, we are able to simplify them to finally obtain the simpler constraints of the previous example.

```
Goal ∃(f: nat → nat),
  ∀ n,
    1 + (if zerop n then 0
         else 1 + max (f (n/2)) (f (n − (n/2) − 1)))
    ≤ f n.
Proof.
  begin defer assuming f. (1) exists f.
  intro n. destruct (zerop n) as [|Hn].
  { subst n. simpl. (2) defer. }
  { rewrite max_l; swap 1 2.
    { (3) defer M: (monotonic f). (4) ... }
    { ... (5) revert n Hn. (6) defer. } }
  end defer.
  (7) ...
Qed.
```

(1)
```
f: ?x
g: Group (?G f)
──────────────────────────────────────
 ∃(f0: nat → nat),
   ∀ n, 1 + (if zerop n then 0
             else 1 + max (f0 (n/2)) (f0 (n − (n/2) − 1)))
        ≤ f0 n
```

(2)
```
f: nat → nat
g: Group (?G f)
──────────────────────────────────────
 1 ≤ f 0
```

(3)
```
f: nat → nat
g: Group (1 ≤ f 0 ∧ ?Goal0)
n: nat
H: 0 < n
──────────────────────────────────────
 f (n − n / 2 − 1) ≤ f (n / 2)
```

(4)
```
f: nat → nat
g: Group (1 ≤ f 0 ∧ monotonic f ∧ ?Goal0)
n: nat
H: 0 < n
M: monotonic f
──────────────────────────────────────
 f (n − n / 2 − 1) ≤ f (n / 2)
```

(5)
```
f: nat → nat
g: Group (1 ≤ f 0 ∧ monotonic f ∧ ?Goal0)
n: nat
H: 0 < n
──────────────────────────────────────
 2 + f (n / 2) ≤ f n
```

(6)
```
f: nat → nat
g: Group (1 ≤ f 0 ∧ monotonic f ∧ ?Goal0)
──────────────────────────────────────
 ∀ n, 0 < n → 2 + f (n / 2) ≤ f n
```

(7)
```
──────────────────────────────────────
 ∃ f: nat → nat,
   1 ≤ f 0 ∧ monotonic f ∧
   ∀ n, 0 < n → 2 + f (n/2) ≤ f n
```

The first step of the proof is to start collecting constraints about `f`, using `begin defer assuming f` (1). This gives us a new identifier `f`, that we can use right away as a witness for the existential quantification at the head of the goal. Case splitting first allows extracting the inequation for the base case (2). Some work can be done in the recursive case (3), if we additionally assume that `f` is monotonic (4). In that case the `max` ... can be simplified to its first argument (5). The inequation at (5) is the simplified inequation for the recursive case. However, it is not possible to call `defer` at this point: the goal depends on the local variable `n`, while the set of collected constraints can only depend on `f`. The solution is to generalize the goal to any positive `n` (6), by moving `n` and `Hn` from the local context to the goal. Finishing the deferring process yields the set of simplified constraints, which corresponds to the initial goal of the previous example (7).

Here again, the point of using *Procrastination* is to impose a *proof discipline* that improves the robustness of the proof script. In this example, we assumed that the goal has been produced by a mechanized process. We would like the proof script to be maintainable, with respect to (non-fundamental) changes in this mechanized process. *Procrastination* is used a first time to extract a set of simplified constraints, then a second time (as detailed in Section 4) to find a solution for the simplified constraints, without relying on the exact values in these constraints. By explicitly splitting the proof into two parts that operate at two different levels of abstraction, we can hope that if the structure of the initial goal changes, only the first half of the proof will need to be updated.

6 *PROCRASTINATION* AND THE PROOF CONTEXT

As illustrated previously, one can think of *Procrastination* as a way of incrementally collecting some side-conditions `P`, `Q`, `R` about some variables `a`, `b`, `c` in order to eventually produce the goal: `∃ a b c, P ∧ Q ∧ R`.

There are two key aspects to this process.

Side-conditions are collected and gathered under a common context. Specifically, it is the context at the point of calling"`begin defer assuming a b c`", extended with the names `a`, `b` and `c`. Only side-conditions `P`, `Q` and `R` that make sense in this context can be deferred using `defer`.

More generally, to each "defer block" (beginning with `begin defer` and ending with `end defer`) corresponds a proof context in which the side-conditions will be gathered. This is in fact materialized by an assumption of type "`Group (...)`" added to the context by `begin defer`. This assumption stores the current collection of side-conditions, and can be named explicitly by the user.

Requiring all side-conditions to be expressed in the same context allows the library to collect them as a conjunction in a single final subgoal `∃ a b c, P ∧ Q ∧ R`. This is crucial, since the motivation is to find values for `a`, `b`, `c` by inspecting the side-conditions `P`, `Q`, `R` altogether (using a tactic or manually).

When collecting side-conditions, variables `a`, `b`, `c` *are rigid.* Because the end goal is to find an instantiation for `a`, `b` and `c`, one could think of first introducing evars for these, and then try to collect side-conditions. However, this quickly becomes unwieldy, as the evar for collecting side-conditions will depend itself on evars for `a`, `b` and `c`. This is in general tricky to handle, the main risk being that `a`, `b` or `c` get accidentally instantiated in undesired ways while trying to collect side-conditions about them.

Procrastination imposes a two-step process, where side-conditions are first collected with respect to "rigid" names for `a`, `b` and `c`. This limits the number of evars involved to a single one in "`Group (...)`", which is only manipulated by the library tactics. After the side-conditions have been collected, the user is free to introduce evars for `a`, `b` and `c`, if needed to prove the goal `∃ a b c, P ∧ Q ∧ R`.

7 NESTING `DEFER` BLOCKS

Interestingly, it is possible to nest defer blocks in an arbitrary way. In some sense, collecting side-conditions is done in a "well-scoped" manner. For example, it is possible to open a second defer block while being in a defer block. Then, the nested block will have a larger context than the enclosing block. When using `defer` on a side-condition, the user can control in which `Group` (and therefore in which context) the side-condition gets collected.

8 IMPLEMENTING *PROCRASTINATION*: CORE IDEAS

In essence, the implementation of *Procrastination* is extremely simple. As its core, "`begin defer assuming a`" is nothing more than "`eapply defer_1`", where `defer_1` is the following tautological lemma:

```
Lemma defer_1 : ∀ A (P: Prop) (Q: A → Prop),
  (∀ a, Q a → P) →
  (∃ a, Q a) →
  P.
```

This lemma (and the rest of the library) comes from the combination of two main ideas, described below.

8.1 Collecting side-conditions

Independently from reasoning on additional parameters a, b, c..., the first idea is the one of incrementally collecting side-conditions while progressing through a proof.

Introducing additional facts in the middle of a proof corresponds to the usual *Cut* rule (or `cut` tactic). *Cut* can be stated as the following Coq lemma "on a goal P, we can assume some facts Q if we prove them later":

```
Lemma cut_lemma : ∀ (Q P: Prop), (Q → P) → Q → P.
```

When using `cut` (the tactic) or when doing `apply cut_lemma`, the user needs to state upfront which the additional facts Q that are being assumed (and which proof is deferred for later) – e.g. `apply (cut_lemma (foo ∧ bar))`.

Now, interestingly, one can also delay giving Q, by calling "`eapply cut_lemma`". This makes use of Coq's evars mechanism, and introduces an evar for Q, since it hasn't been determined yet.

```
Goal P.
  eapply cut_lemma.
  – intro H.
    (* H: ?Q ⊢ P *)
  – (* ⊢ ?Q *)
```

In the first subgoal, we get an assumption H which type is an evar. Should P yield a subgoal P' that we may want to prove later, P' can simply be discharged by applying H. This will cause the evar ?Q to get instantiated with P' – requiring P' to be proved in the second subgoal spawned by `eapply cut_lemma` as a side effect.

```
Goal P.
  eapply cut_lemma.
  – intro H.
    (* H: ?Q ⊢ P *)
    ...
    (* H: ?Q ⊢ P' *)
    now apply H.
    (* H: P' *)
  – (* ⊢ P' *)
```

This trick can be made a bit more useful by using H to discharge not only one subgoal, but several. This can be achieved (e.g. using a small amount of *Ltac*) by instantiating H into an iterated conjunction of already discharged subgoals, and an evar for future ones. For instance:

```
Goal P.
  eapply cut_lemma.
  – intro H. (* H: ?Q ⊢ P *) ...
    (* H: ?Q ⊢ P' *) now eapply (proj1 H). ...
    (* H: P' ∧ ?Q ⊢ P'' *) now eapply (proj1 (proj2 H)). ...
    (* H: P' ∧ P'' ∧ ?Q ⊢ ... *)
  – (* ⊢ P' ∧ P'' ∧ ?Q *)
```

This is in fact precisely how the `defer` tactic is implemented.

8.2 Dealing with existentials in the goal: avoiding the proliferation of evars

Let us now consider the more specific case where P is of the form "∃ x, P x". In order to collect side-conditions that may talk about x, one possibility is to first call `eexists`, introducing an evar for x, and then use the technique described previously.

```
Goal ∃ x, P x.
  eexists. (* ⊢ P ?x *)
```

As mentioned earlier, in our experience this does not scale up very well. The risk is that might ?x get instantiated by mistake while simply trying to collect side-conditions about it, or while progressing through the proof of P ?x.

Therefore, the second main idea is to use variants of the cut lemma for collecting side-conditions "under existential quantifications". For example, we can easily prove the following lemma:

```
Lemma cut_ex1_lemma: ∀ A (P Q : A → Prop),
  (∀ x, Q x → P x) →
  (∃ x, Q x) →
  ∃ x, P x.
```

By doing eapply cut_ex1_lemma, we can collect side-conditions about x in ?Q x, without introducing an evar for x. All in all, all this does is massaging the proof so that it is setup more conveniently.

Finally, the lemma above can be generalized to support an arbitrary number of quantified variables (including 0, in which case this corresponds to the previous subsection). This is in essence how begin defer is implemented.

9 TACTICS REFERENCE

- `begin defer [assuming a b...] [in g]`
  Start a "defer" block. If `group g` is provided, give name g to the hypothesis of type Group introduced. If `assuming a b...` (where a, b, … are one or several user-provided names) is provided, also introduce in the context abstract variables of the same names.

- `defer [in g]`
  Discharge the current sub-goal, deferring it for later. If `in g` is provided, store it in the hypothesis g which must be of type `Group ...`. If `in g` is not provided, the tactic picks the hypothesis of type `Group ...` which has been introduced last (if there are several).
  The sub-goal being differed has to make sense in the context of the Group it is stored in (which is the context of the `begin defer` that introduced the Group). Otherwise Coq will raise an error involving the context of the evar in the Group.

- `defer [H]: E [in g]`
  Introduces a new assumption E, named after H, and defer its proof in g. This is equivalent to `assert E as H by (defer in g)`.
  If H is not provided, adds E in front of the goal instead of adding it to the context – i.e. on a goal G, `defer: E` produces a goal `E -> G`.

- `end defer`
  To be called on a `end defer` goal. It does some cleanup and gives back the variables and side-goals that have been procrastinated.

- `deferred [in g]`
  Adds to the goal all the already deferred propositions. I.e. on a goal G, gives back a goal `X1 -> .. -> Xn -> E`, where X1..Xn are the already deferred propositions.

- `deferred [H]: E [in g]`
  Adds an assumption E to the context, named after H, and produces a subgoal `X1 -> .. -> Xn -> E`, where X1..Xn are the already deferred propositions. As a convenience feature, when E is trivially provable from X1..Xn, this subgoal is automatically discharged using auto.
  This is equivalent to `assert E as H; [ deferred in g; try now auto |]`.
  If H is not provided, adds E in front of the goal instead of adding it to the context.

- `exploit deferred tac [in g]`
  This allows iterating a user-provided tactic tac on the propositions stored in the group g. It is fine for tac to fail for some of the propositions. The whole tactic (`exploit deferred tac`) will fail if tac did not make the proof progress overall (i.e. if nothing happened).

For example, to try rewriting with all deferred facts, one would do `exploit deferred (fun H => rewrite H)`.

REFERENCES

[1] Cohen, C.: Formalized algebraic numbers: construction and first-order theory. Ph.D. thesis, École Polytechnique (2012)
[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (Third Edition). MIT Press (2009)
[3] Guéneau, A.: The `procrastination` library (Jul 2018), https://github.com/Armael/coq-procrastination
[4] Guéneau, A., Charguéraud, A., Pottier, F.: A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In: European Symposium on Programming (ESOP) (2018)
[5] The Coq development team: The Coq Proof Assistant (2016)