Procrastination

A proof engineering technique

ARMAËL GUÉNEAU, Inria

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.

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

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 procrastination assuming a b c. (1)
  exists (fun n ⇒ if zerop n then c
                  else a * log2 n + b).
  repeat split.
  { simpl. (2) procrastinate. (3) }
  { … (* c ≤ b *) procrastinate. … }
  { intros.
    … (* 2 + c ≤ b *) procrastinate.
    … (* 2 ≤ a *) procrastinate. }
  end procrastination.
  (4) eomega.
Qed.
```

```
a: ?x                                                    (1)
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
```

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

```
a, b, c: nat                                             (3)
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 procrastination 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 `procrastinate` tactic, which discharges the subgoal (3). Similarly, the other two inequalities yield side conditions about a, b and c that can be `procrastinated`. Finally, `end procrastination` 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.

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 (Jan 2018), https://gitlab.inria.fr/agueneau/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)