1 Retrospective of the week

1.1 Trocq as setoid_rewrite

We explored how pattern selection mechanism could be extended so as to allow users to provide a minimal information about the shape of the context to be inferred. Typically, one would like Rocq to deal automatically with potential dependencies, and abstract as many subterms as needed for the context to be well-typed, but not more. Then, more relations than the one specified by the users might be needed to construct the witness and these have to be found among the registered ones/ provided. [attempt to rewrite the prev sentence] In this case the user may need to specify more relations in order to construct the witness (these additional relations are not among the globally registered ones).

1.1.1 Example:

rewriting

$$e : N \simeq N', 0_R : e\ 0\ 0', S_R : (e \Rightarrow e)SS'$$

in

$$\forall (P : N \to \text{Type}), P0 \to (\forall n, Pn \to P(S\ n)), \forall n : N, P\ n$$

We may be able to write either of `rewrite e` or `rewrite 0_R`. Pattern selection should be able to abstract at the same time N, 0 and S, in both cases.

- for `rewrite e` the inferred pattern is $(\lambda X.\forall (P : X \to \text{Type}), P0 \to (\forall n : X, Pn \to P(Sn)), \forall n : X, Pn)$ but since it does not typecheck (eg $P0$ is illtyped), the pattern selection should also abstract on 0 and S
- for `rewrite 0` the inferred pattern is $(\lambda x : N.\forall (P : N \to \text{Type}), Px \to (\forall n : N, Pn \to P(Sn)), \forall n : N, Pn)$, it does typecheck, but it has type $N \to \text{Prop}$ and cannot be applied to $0'$: the pattern typechecks but is not suitable for heterogenous replacement. The pattern selection should abstract on N and thus on S.

A correct inferred pattern could be (and it is not unique):

$$(\lambda X : Type, \lambda x : X, \lambda y : X \to X.\forall (P : X \to \text{Type}), Px \to (\forall n : X, Pn \to P(y\ n)), \forall n : X, Pn$$

1.1.2 Objective:

1. Given a goal G, and a set of relations $p_i : r_i\ t_i\ t'_i$ to rewrite with, we need to find a well typed context $P : T_P$ such that $P\ t_1\ \cdots\ t_n$ unifies with G, such that $P\ t'_1\ \cdots\ t'_n$ is also well typed.
2. Use Trocq (using solely identity translations) to get $P_R : [\![T_P]\!]^{(0,1)} PP$ so that $P_R\ t_1\ t'_1\ p_1 \cdots t_n\ t'_n\ p_n$ is a proof of the desired implication:

$$P\ t'_1\ \cdots\ t'_n \to G$$

1.1.3 Observations:

This framework involves three natures of relateness witnesses: - relating terms with their identity/parametricity/white box translations (the translations obtained by parametricity), e.g., P_R - the morphisms/black box translations, i.e. given a pair of types and a relation between them, constructions in one type have a default translation in the other e.g. 0_R and S_R. It should be guided by the relation we rewrite with. (e.g $0'_R : e'00'$ is not a valid witness) - user given translations at rewrite time, which are abitrary and non inferrable. e.g. $e : N \simeq N'$.

1.1.4 Questions:

- In general, P will not be obtained after the sole abstraction of (prescribed occurrences of) the left hand-side of the relation provided by the user (e). There should then be a backtracking process for computing a suitable P. What level of vagueness do we want to allow?
- More abstractions mean more relation witnesses, how to make sure Trocq finds them.

1.1.5 Roadmap:

Track 1.

1. fully explicit context
2. fully explicit patterns (one pattern for each lambda)
3. use second order pattern matching to find a minimal context given a set of terms to abstract
4. giving only a subset of the terms to abstract
5. do not give a pattern, piggyback on 4. using the inferred one.

Track 2. Use rewritestrat: - rewritestrat should compute the expected level from the context.

Track 3. Add subrelations to Trocq

1.2 Quentin's PR and Iris bug

With the help of Gaetan we observe that the failure in Iris is that a goal previously kept in the shelf is now presented as a goal to be immediately solved.

~~We discover that instantiation of an evar with another (pruned) evar preserves some flags attached to the evar being assigned (like being on the "shelf", or being a "future goal"). The code in `evd.ml` has a `inherit_flags` internal function to~~

~~partially do this job.~~ Although the flags do need to be preserved, the shelf and the flags are independent.

The PR by Quentin implements pruning of evars applied to terms, so now the code path is evar assigned to a beta redex (the redex drops the arguments to be pruned). In this case ~~flags are not inherited by~~ the restricted evar is not put on the shelf.

~~We add an API to restrict an evar via a beta redex that calls `inherit_flags`. Preserving the shelf flag did not help in the specific case.~~

TODO: ~~correctly set the future goal flag. This is not a regular flag, but rather a data structure in the evar map that is used by lower layers to communicate to the proof monad which evars represent the new sub goals. I guess we have to make sure that we don't add the new pruned evar in that list by mistake (unless the old evar was there in the first place).~~ adding the evar in the shelf in `evardefine.ml/define_pure_evar_as_lambda` breaks things, we need to debug it.

1.3 Unification up to structure inheritance graph

Quentin explained the optimization known as "save keys", the same that was presented at the Liberabaci midterm meeting.

We observe that some diagrammatic reasoning that seems trivial on the inheritance graph is implemented by intertwined reduction steps and calls to the CS table.

Cyril and Enrico try to detail an algorithm where coercions are elided and we reason on terms of different types but keeping some information on the side to finally reconstruct well typed terms. The result that emerges from carrying out a few examples is that it is not trivial, and seems anyway just pointing to the separation of bare unification from reasoning on the inheritance graph.

unif/CS problems of the form `proj1 .. = proj2 ..` can be solved by walking the graph (of coercions, but only the coercions that represent inheritance paths). It existed in Matita and is also implemented for reverse coercions: it is the computation of the pullback between `proj1` and `proj2`. The same walk is implemented today by a number of entries in CS table (typically synthesized by HB). These entries represent the ahead-of-time solutions to all possible pulback computations. So the lookup is fast, the table huge, unifying the solution with the `..` may be expensive and this is also what Quentin's PR tries to solve. Walking a non-transitively-closed graph can be more expensive (without a cache morally of the size of the CS table, eg quadratic[1]). But applying the solution can be done in a smarter way, eg `proj1 (proj2 t) = proj3 u` finds a solution for `t` and `u`, while the current algorithm finds one to be checked against `proj2 t` and that may require more unification work to finally assign `t` as desired.

Quentin and Erico don't think there is an immediate speedup as initially conjectured by Cyril. Enrico, Cyril, Assia, Quentin and Matthieu think that the unification algorithm is simpler to explain by separating the just-unification

[1] Cyril has an idea on how to make the cache linear.

part from the just-inheritance part. This might also help amending deep layers of a hierarchy.

Roadmap:

- tag coercions that are inheritance paths
- desgin the unif
- try to patch evar_conv ^^

1.4 Evars in evarconv

1.4.1 Folding before solving evars

If the algorithm unfolds terms and then finds that one side is an evar, it does not fold the other side back to its original state.

Doing it might have a nice side-effect on the usual refolding issue for fixpoints (hopefully).

1.4.2 Stop heuristically solving hard higher order unification problems unless specifically asked to

Full higher order unification is undecidable, things break easily as soon as we encounter such problems. We would like to stop solving them and let the user give the instantiation she wants. There are several ways to do that: - Fail and tell the user which evars were involved in the unification problem (as in which implicit arguments produced the evars) as an error message. - Open a specific goal for the unification problem, where the user can intervene. We could give the unification problem: - as an equality that needs to be solved by `eq_refl` - as an equality hidden under some existential quantifiers (one for each evar) - as a new relation without constructor that the user can not do anything about except instantiating the evars and calling "`reflexivity`"/done/over/... - as shelved evars by showing the unification constraints

Matthieu already did something around this: see `Unset Solve Unification Constraints`.