Towards a better-behaved unification algorithm for Coq

Beta Ziliani
MPI-SWS
beta@mpi-sws.org

Matthieu Sozeau
Inria
matthieu.sozeau@inria.fr

June 3, 2014

The unification algorithm is at the heart of a proof assistant like Coq. In particular, it is a key component in the *refiner* (the algorithm that has to infer implicit terms and missing type annotations) and in the application of lemmas. In the first case, unification is in charge of equating the type of function arguments with the type of the elements to which the function is applied. In the second case, for instance when using the `apply` tactic, it is in charge of unifying the current goal with the conclusion of the lemma.

Despite playing a central role in proof development, there is no good source of documentation to understand Coq's unification algorithm. Moreover, in Coq currently there exist two unification algorithms, with different behaviors, challenging the intuition of the proof developer who has to make sense of why some examples work in certain scenarios, but not in others. For instance, the unification algorithm used by the refiner is different from the one used by the `apply` tactic. The reason for such bifurcation is now merely historical, as both algorithms have been converging in functionality over time.

Another thing to take into account is that Coq's unification includes resolution of *Canonical Structures* [4, 5], an overloading mechanism similar to *type classes*. This key mechanism is extensively used in the Mathematical Components library [3], on which the proofs of the four color and odd-order theorems [1, 2] depend. Supporting canonical structures resolution in unification makes the algorithm extremely sensitive to heuristics, since instance resolution depends heavily on the order in which unification problems are considered.

Unification is inherently undecidable in Coq, as it must deal with higher-order problems up to conversion. Therefore, some form of heuristic is desirable in order to solve problems that are trivial to the human eye. Otherwise, the proof developer will get easily frustrated when it finds two apparently equal terms not being unified. For instance, a desirable heuristic will equate the terms $?x +\!\!+ ?y \approx [] +\!\!+ (1 :: [])$, assigning $?x$ to the empty list and $?y$ to the singleton list $(1 :: [])$, where $?x$ and $?y$ are meta-variables and $+\!\!+$ is the list concatenation function. There exist other possible (convertible) solutions, like for instance assigning $(1 :: [])$ to $?x$ and $[]$ to $?y$, but in most of the cases

preserving the structures of terms gives reasonable solutions.

The current approach in the source code of Coq includes this heuristic but also some harmful ones, like postponing equations. Indeed, in certain cases, when an equation has multiple solutions it is delayed, waiting to have more information to solve the ambiguity. Delaying equations is commonly used, and gives in practice reasonably good results, but in combination with canonical structures it can be catastrophic, as the search for instances may run on supposedly equal terms that are actually not yet proved to be equal. Moreover, most of the postponing can be avoided if the order in which the unification problems are considered is properly controlled.

In this talk we are going to present a new unification algorithm, built from scratch, which focuses on the following main properties:

Understandable: The algorithm can be described in full in a few pages, including canonical structures instance resolution.

Sound: The algorithm, when it succeeds, provides a well-typed substitution that equates both terms (up to conversion).

Simple: The algorithm does not include heuristics that are hard to reason about.

Configurable: The algorithm adapts to different scenarios, therefore avoiding the need for different unification algorithms.

In the talk we are going to present the benefits of such an algorithm, but also what are the challenges that we face when considering "the big picture": the inclusion of our algorithm in the zoo of Coq's tactics.

References

[1] Georges Gonthier. Formal proof — the four-color theorem. *Notices of the AMS*, 55(11):1382–93, 2008.

[2] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *ITP 2013*, volume 7998 of *LNCS*. Springer, 2013.

[3] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the Coq system. Technical report, INRIA, 2008.

[4] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *Proc of POPL'97*, pages 292–301, 1997.

[5] Amokrane Saïbi. *Outils Generiques de Modelisation et de Demonstration pour la Formalisation des Mathematiques en Theorie des Types. Application a la Theorie des Categories.* PhD thesis, Université Paris 6, 1999.