How to Make Ad Hoc Proof Automation Less Ad Hoc Technical Appendix

Georges Gonthier
Microsoft Research
gonthier@microsoft.com

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

Aleksandar Nanevski
IMDEA Software Institute
aleks.nanevski@imdea.org

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

A. Operational semantics of unification

The following section is intended as a guide for the reader willing to "take the red pill" and attempt the overloading of lemmas with different shapes than the ones described in the paper. The operational semantics of canonical instance resolution that we present here only applies to the current version of Coq (8.3).

Declaration Every instance declaration has the following components:

$$\mathit{inst\_id}\ \vec{x}\ :\ \mathit{struct\_id}\ \vec{a}\ :=\ \mathit{constr\_id}\ (K_1\ \vec{u}_1)\ \ldots\ (K_n\ \vec{u}_n)$$

where

inst_id	is	the name for the instance
$\vec{x}$	are	the parameters for the instance
struct_id	is	the name of the structure
$\vec{a}$	are	the parameters for the structure
constr_id	is	the name of the constructor of the structure
$K_i\ \vec{u}_i$	are	the values for each field of the structure, in the order they were declared. Each K_i is a head constant, whose parameters are $\vec{u}_i$

For example, in the case of start from Section 6:

inst_id	is	start
$\vec{x}$	are	x, y and s_2
struct_id	is	implicit, and it is check
$\vec{a}$	are	x, y and untag (seq2_of s_2)
constr_id	is	Check
$K_1\ \vec{u}_1$	are	equate y
$K_2\ \vec{u}_2$	are	the ommitted proof (...) with parameters x, y, s_2

When a canonical instance is declared, an entry is added to the canonical structure database, for each projector p_i, with the key (p_i, K_i) and the definition above as value. This happens if K_i is a symbol (like •, equate, ...).[1]

If K_i is not a symbol, but a variable, and u_i is empty, then the *default* key $(p_i, \_)$ is added, and it will match any expression of the type of the variable. These default instances are widely used in the *tagging* pattern. Default keys are compatible with the regular ones, and they are checked in the end. But beware! If, instead of a symbol, one has an abbreviation or synonym (like our tags), it will not be unfolded to reveal the true head constant. For this, it's better to use the tagging pattern to guide the typechecker instead of relying on unification to make the right choice.

There are two cases that prevent Coq from adding keys to the database:

1. for every unspecified projector, *i.e.,* _ , and
2. for every key (p_i, K_i) already in the database.

[1] Note that non-dependent product type $\to$ is treated as a symbol.

Instance resolution When solving for $?s$ the equation

$$p_i\ \vec{b}\ ?s\ \vec{y}\ \widehat{=}\ V\ \vec{v}\ \vec{z}$$

where $\vec{y}$ and $\vec{z}$ have the same length.

If $V = K_i$ for some (p_i, K_i), in the canonical structure database, the system will use the instance $\mathit{inst\_id}\ ?\vec{x}$ bound to the pair (p_i, K_i) in the database; otherwise, it will use the default instance (if any) with projection p_i. Then the system resolves the following equations in order.

1. $\vec{a}\ \widehat{=}\ \vec{b}$, to unify structure arguments. Serves to pass parametric inputs to subproblems and/or collect outputs.
2. $\vec{u}\ \widehat{=}\ \vec{v}$, to unify arguments of the head constant. Serves to pass inputs extracted by projection from the context of application.
   (a) In the case of a default instance, $K_i\ \widehat{=}\ V$
3. $?s\ \widehat{=}\ \mathit{inst\_id}\ ?\vec{x}$, to unify the instance variable with the particular instance.
4. $\vec{y}\ \widehat{=}\ \vec{z}$; if the projector p_i returns a function, to unify its arguments. This has not been used in this paper.

When resolving any of the equations, the whole unification procedure is invoked recursively, so in particular, new equations may be added on top of the existing stack.