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

Georges Gonthier  Beta Ziliani  Aleksandar Nanevski  Derek Dreyer
Microsoft Research  MPI-SWS  IMDEA Software Institute  MPI-SWS
gonthier@microsoft.com  beta@mpi-sws.org  aleks.nanevski@imdea.org  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:
\[
\text{inst}_\text{id} \; \bar{x} : \text{struct}_\text{id} \; \bar{a} := \text{constr}_\text{id} \; (K_1 \; \bar{u}_1) \ldots (K_n \; \bar{u}_n)
\]
where
\[
\begin{align*}
\text{inst}_\text{id} & \quad \text{is the name for the instance} \\
\bar{x} & \quad \text{are the parameters for the instance} \\
\text{struct}_\text{id} & \quad \text{is the name of the structure} \\
\bar{a} & \quad \text{are the parameters for the structure} \\
\text{constr}_\text{id} & \quad \text{is the name of the constructor of the structure} \\
K_i \; \bar{u}_i & \quad \text{are the values for each field of the structure, in the order they were declared. Each } K_i \text{ is a head constant, whose parameters are } \bar{u}_i.
\end{align*}
\]

For example, in the case of start from Section 6:
\[
\begin{align*}
\text{inst}_\text{id} & \quad \text{is start} \\
\bar{x} & \quad \text{are } x, y \text{ and } s_2 \\
\text{struct}_\text{id} & \quad \text{is implicit, and it is check} \\
\bar{a} & \quad \text{are } x, y \text{ and } \text{untag } (\text{seq2} \; \text{of} \; s_2) \\
\text{constr}_\text{id} & \quad \text{is } \text{Check} \\
K_1 \; \bar{u}_1 & \quad \text{are } \text{equate } y \\
K_2 \; \bar{u}_2 & \quad \text{are the omitted proof } (...) \text{ with parameters } x, y, s_2
\end{align*}
\]

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 \( \bullet \), \text{equate}, \ldots).\(^1\)

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 \( \rightarrow \) is treated as a symbol.