A new Type-Class solver for `Coq` in `Elpi`

Davide Fissore[1] and Enrico Tassi[1]

Inria
Davide.Fissore@inria.fr Enrico.Tassi@inria.fr

Type classes: from programming to interactive proving

Type classes [6] were first introduced in the `Haskell` language as a way to implement ad-hoc polymorphism, *i.e.* symbol overloading. The `Haskell` compiler synthesizes the appropriate implementation for the overloaded symbol by looking at the types of its arguments and by composing user-provided implementations for atomic types and type formers. We call the software component in charge of this synthesis the *type class solver*.

Type classes soon were introduced in `Coq` version 8.2, 15 years ago, by Sozeau and Oury [5]. Given the expressiveness of the language, `Coq` type classes not only relate a type to some operations, but also give them a specification. For example the `Eq` type class below links a type `T` to a boolean function `eqb` which is proved to be correct.

```
Class Eq T := { eqb: T -> T -> bool; eqP : forall x y, eqb x y = true <-> x = y }.
```

In the light of that type classes can play the role of interfaces and their solver plays the role of a pervasive form of automation, a major difference of use case w.r.t. `Haskell`. In particular by organizing theories around interfaces the application of a generic theorem to a specific example is automatically justified by the type class solver which *proves* that the example implements the required interfaces. Most modern `Coq` libraries are built around type classes, making their solver a key component of `Coq`.

Weaknesses of the current type class solver By using Type classes one quickly realizes that `Coq`'s solver is implemented by calling the `eauto` tactic which in turn iterates the `apply` tactic using a dedicated hint database containing known type class implementations.

The main advantage of the current implementation is that it reuses powerful existing features that are continuously maintained and updated. There are, however, several downsides, severe enough to justify the study of a new implementation.

First and foremost, the `apply` tactic is not designed with automation in mind: when users apply lemmas they expect them to work, hence the `apply` tactic tries very hard to unify the conclusion of the lemma with the goal, and failing to do so can take a considerable amount of time. On the other hand automatic solvers are not driven by the educated guess of the user but they rather try to explore all possibilities by brute-force. Hence most of the attempts lead to a dead end and failing quickly is crucial. Over the years the type class solver got plenty of settings to tweak its behavior such as: preventing the unfolding of constants; avoiding goals which are too unconstrained; cutting search branches that match undesired patterns; etc. We count more than ten of these.

Even so, power users are still demanding features and extra degrees of configurability. One can use `Hint Extern` to locally take over the search, but cannot change the overall searching procedure, nor easily instrument it or debug it and fix it as we are used to do with any computer code we are not happy about. Our diagnosis is that the solver lacks of a proper "language" to describe how the search is done, a language power users could take full advantage of.

A new solver written in the `Elpi` Logic Programming language

`Elpi` [1] is a dialect of λ`Prolog` enriched with constraints. It is a higher order logic programming language well suited to manipulate syntax trees with binders and holes (*i.e.* lack of information

to be inferred), such as Coq terms during type class resolution. Elpi comes in the Coq-Elpi plugin as an interpreter, and Coq can call a dedicated Elpi program whenever a type class needs to be solved. In the code block below we show, side by side, the Coq code declaring two type-class instances for the Eq class and the corresponding Elpi code (curly braces quote Coq syntax within Elpi, and lp:X escapes back, letting one bind a subterm to the variable X):

```
Instance eqBool : Eq bool := { ... }.
Instance eqProd `{Eq A, Eq B}: Eq (A * B)
  := { ... }.
```

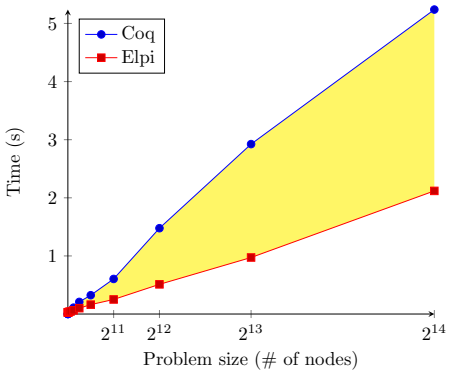
```
tc {{ Eq bool }} {{ eqBool }}.
tc {{ Eq (lp:A * lp:B) }} {{ eqProd lp:SA lp:SB }} :-
    tc {{ Eq lp:A }} SA,
    tc {{ Eq lp:B }} SB.
```

The instance for bool corresponds to a fact, a rule with no premises, while the instance for the pair (A * B) requires an implementation of Eq for A and B and hence corresponds to a rule with two premises. Whenever Coq needs to solve a type class we rephrase the problem as an Elpi query. For example solving Eq on (bool * bool) amounts to run the Elpi query "tc {{ Eq (bool * bool) }} Solution". The program above succeeds by chaining the two rules and answers "Solution = {{ eqProd eqBool eqBool }}".

We developed an automatic translator generating the Elpi code on the right starting from the Coq code on the left, but power users are free to write custom rules by hand, and even use different solvers for different type classes if they feel the need for it. Finally Elpi comes with a *trace browser* which allows to interactively navigate the steps made by the interpreter, substantially simplifying the task of debugging the type class solver.

The complexity of type classes search The Elpi's solver allows power users to have a full control on the search strategy. For example they can program a non-backtracking search for specific type classes, or pre-process the proof context by forward rewriting. In some cases this is sufficient to change the complexity class of type class search, especially in case of failure. One of such cases is a hierarchy of type classes featuring a tower of diamonds as in [4].

Synthetic benchmark We are currently testing our implementation on the stdpp [2] library to compare Coq and Elpi's solvers performances. The plot on the right shows the time taken to solve a given problem whose size is the number of nodes in the term driving the search (*i.e.* the type for which we search the Eq class). On small values Coq is faster even if any solver takes a negligible amount of time. For bigger problems our solver is about 2.5 times faster, showing that a more principled search strategy can be competitive even when run by an interpreter: our solver does not call apply, but rather uses the simpler and more efficient unification of Elpi.



Future work Currently, our solver is not in a stable and complete, but results are promising since we already cover about the 73.54% of stdpp type-class problems; we aim to cover the remaining part soon.

We also plan to make Elpi's solver work with the iris library and the setoid rewriting one. Both use type class search as a form of automation and go to great lengths to optimize the search strategy, proving an even better test bench for our approach.

Finally, whilst reasonably efficient Elpi's runtime could benefit from forms of memoization, building on the work of Pientka [3] and De Moura [4].

References

[1] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λProlog interpreter. In *LPAR-20*, volume 9450 of *LNCS*, pages 460–468. Springer, 2015.

[2] Robbert Krebbers et al. https://gitlab.mpi-sws.org/iris/stdpp.

[3] Brigitte Pientka. Tabling for higher-order logic programming. In Robert Nieuwenhuis, editor, *CADE-20*, volume 3632 of *LNCS*, pages 54–68. Springer, 2005.

[4] Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled typeclass resolution. *CoRR*, abs/2001.04301, 2020.

[5] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *TPHOLs*, pages 278–293, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[6] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, page 60–76, New York, NY, USA, 1989. Association for Computing Machinery.