



MASTER
INFORMATIQUE



UNIVERSITÉ
CÔTE D'AZUR

MASTER IN COMPUTER SCIENCE

Type-class solver in Type Theory via Logic Programming

Author:
Fissore Davide

Supervisor:
Enrico Tassi

Abstract

Since their introduction, type classes [2] have met a huge success in functional programming languages: this kind of *ad-hoc* polymorphism allows to define a function signature and provide implementations of it over different types. This feature has been firstly introduced in Haskell [5] and then in more and more functional languages. Among them we can find the Coq proof assistant [7] where type classes not only carry the signature of functions, but also their signature. Hence, instances not only provide the code for the overloaded symbols but also the proofs of its signatures. Type-class resolution is done by the Coq's type-class solver, but its search algorithm is difficult to tweak, performance is sub-optimal, no check on instance declaration is done, etc. We aim at provide an alternative and flexible implementation of a type-class solver for Coq written in the Elpi logic programming language.

Master Informatique
at Université Côte d'Azur
Academic year 2022-2023

Contents

1	Context and related works: Elaboration in Coq	2
1.1	Different kinds of polymorphism	3
1.2	A zoom on type classes	4
1.3	<i>Simple</i> and <i>Complex</i> instances	4
1.4	Parallel with object-oriented languages	5
1.5	Coq’s type-class solver	5
1.5.1	Database of instances	6
1.5.2	Goal resolution	6
1.5.3	Hint modes for type classes	7
2	Elpi: Embeddable λProlog Interpreter	8
2.1	Motivation to use Elpi for type-class resolution	9
2.2	Coq-elpi: Elpi for Coq	9
3	The Elpi’s type-classes solver prototype	10
3.1	Instance Compilation	10
3.1.1	The simple <code>tc</code> predicate	10
3.1.2	Elpi’s vs Coq’s modes	12
3.1.3	Drawbacks of using the simple <code>tc</code> predicates	13
3.1.4	A better solution: ad-hoc predicate for type classes	15
3.2	Instance priorities	16
4	Tactics and commands of our Elpi’s type-class solver	17
4.1	The <code>TC_solver</code> tactic	17
4.2	Commands to add type-class signature in <code>Elpi</code>	18
4.3	Commands to add instances in <code>Elpi</code>	18
4.4	New features of <code>Elpi</code> ’s solver wrt <code>Coq</code> ’s one	19
4.4.1	Goals reordering	19
4.4.2	Deterministic search	20
4.5	<code>Elpi</code> ’s vs <code>Coq</code> ’s term unification	21
4.6	User-defined rules	23
5	Validation of results and synthetic benchmark	24
6	Future work	26
6.1	Memoisation and tabled search	26
6.2	Indexing algorithm on pattern	27
6.3	Overlapping instances detection	27
7	Conclusion	28
8	References	28
A	Appendix	30
A.1	Term with missing information vs. full term	30
A.2	Instance insertion in <code>Elpi</code> database	31
A.3	Goal reordering	31
A.4	Some example	33

1 Context and related works: Elaboration in Coq

Coq is a proof assistant used for formally proving mathematical formulae and verifying the correctness of programs. This proof system, based on the Curry-Howard correspondence, is able to check correctness of a proof term thanks to the set of rules coming from type theory. This operation is performed by Coq's kernel and the terms it can verify for correctness must be *complete*. A term is *complete* (or *full*) if it does not contain any *holes*, i.e. missing information (sub-term).

When the user writes a term, it is not immediately passed to the kernel for typechecking, but it is elaborated. The elaboration process [11] starts with this term, a raw text in Coq's syntax, that is parsed and abstracted into a syntax tree. In Appendix A.1 we have a quantitative example expressing how compact a term can be with respect to its full representation after elaboration: the term representing the function representing the **determinant** of a matrix, shown below, takes 64 characters w/o spaces, whereas the complete term takes 1,678 characters.

It turns out that users are willing to write terms containing only the strictly necessary information needed to definite them. To make this possible, there should be a complex infrastructure capable of automatically inferring the missing holes and filling them when possible. The set of algorithms accomplishing this task is derived from all of the inference rules we can get from type theory, and the tool responsible for building the bridge between the parsed term and the kernel is called the *elaborator*. It is the elaborator that allows the user to no longer worry about explicitly specifying that, in the **determinant** function, the argument **n** is a **nat**, that **A** is a matrix of size $n \times n$ whose elements belong to a semiring, and so on.

```

1 Definition determinant n (A : 'M_n) : R :=
2   Σ (σ : perm_of n) sgn(σ) * Π i, A i (σ i).

```

A picture sketching the structure of the elaborator is given in Figure 1.

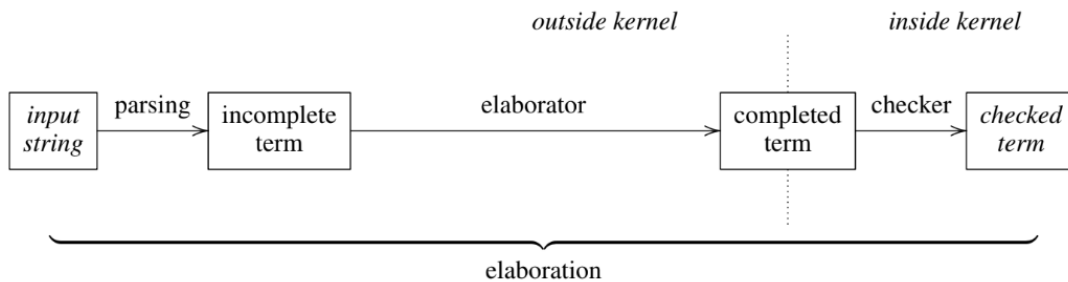


Figure 1: Elaboration stages

Tasks of the elaborator The elaborator of a proof assistant is faced with several tasks, since there are different kinds of holes that can be left implicit in a term. The more classic kind of missing information is the *type* of sub-terms, in

```
let x := 3 in Nat.succ x
```

we can deduce $x : \mathbf{nat}$, since 3 is an implementation of the inductive type **nat**, **Nat.succ** : **nat** → **nat**, since, once defined, **Nat.succ** is typechecked as a function taking a **nat** and returning a **nat**. Finally, the application of the **Nat.succ** term to **x** will determine the type of the full term, which is **nat**. A more detailed representation of the inference rules is shown in Figure 2.

$$\begin{array}{c}
 \frac{}{\mathbf{true} : \mathbf{Bool}} \quad \frac{}{\mathbf{false} : \mathbf{Bool}} \quad \frac{e_1 : \mathbf{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\mathbf{if } e_1 \mathbf{ then } e_2 \mathbf{ else } e_3 : \tau} \\
 \\
 \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x.e) : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x_1 : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{let } x = e_1 \mathbf{ in } e : \tau_2}
 \end{array}$$

Figure 2: Inference rules

Another kind of implicit information that can be deduced by the elaborator is *coercion*. Coercion in functional languages is used to transform a value of type **T** into a value of type **T'**. Let's imagine having

a function `bool_in_nat: bool -> nat` returning 0 if the input is `false`, and 1 otherwise. In Coq we can define

```
Coercion bool_in_nat : bool >-> nat
```

and from that moment on, it will be possible check and compute the equality `0 = true`.

Finally the elaborator allows to reason with both *canonical structures* and *type classes*, two mechanisms used in Coq to implement function overloading [2, 7, 10]. These two kinds of overloading are rather similar, except that with type classes the user has more freedom. In fact, if multiple canonical structures share the same field, then only the structure declared first is considered. This limitation is not present in type classes, but, as always, the price for high freedom is a more sophisticated algorithm. However, this algorithm must be controlled and correctly managed by the user to avoid loss of performance or even infinite loops.

In this internship, we will focus on the segment of the elaboration process doing type-class resolution.

Our goal is to first introduce the reader to a clear definition of type classes. We will point out what it means to do type-class resolution and how this resolution is done in Coq. In a second moment, we will dedicate a large part of our report to explaining 1. the motivations that led us to reimplement Coq's type-class solver in the high-level logic programming 2. how this implementation is made in Elpi 3. what are the features we have added 4. some performance comparison between our type-class solver and Coq's one.

In the following sections, we will try to simplify the technical details, and when needed, small examples will be proposed to support our explanation and ease the reader's understanding as much as possible.

1.1 Different kinds of polymorphism

Before starting to talk about type classes, it is worth talking about their first appearance and, in particular, explain how polymorphism is defined in functional programming languages. In [2], Wadler and Blott differentiate two kinds of polymorphisms: **parametric** and **ad-hoc**.

The former is a polymorphism in which a function `f` behaves the same way for all the parametric types. In Coq, the type associated with lists is declared as follows

```
Inductive list (A: Set) : Set :=
| nil : list A
| cons : A -> list A -> list A.
```

where `A` is a generic-type parameter. A parametric function on lists is a function that computes some operation on the `list` structure ignoring the concrete instance of `A`. The `length` function is a clear example of that. We can call it on a list of integers $[i_1, \dots, i_n]$, of strings $[s_1, \dots, s_n]$, of binary search trees $[t_1, \dots, t_n]$. In any of these cases, `length` will be able to compute the wanted result without caring about the types integer, string and tree. In fact, with parametric polymorphism, functions' output only depends on a particular property of the data structure. By way of example, other parametric-polymorphic functions are: the `top` function returning the first element of a queue, the `root` function returning the root of a binary search tree... Parametric polymorphism is at the heart of functional programming languages such as OCaml and Haskell where a higher-order style of programming is encouraged.

The second kind of polymorphism is referred to function overloading. It is meant to create different implementations of a function `F` all sharing the same name. The behavior of those implementations will depend on the concrete type of the argument passed to `F`. This notion of polymorphism is widely used by several scientific disciplines such as maths, physics... In these fields of research, the following two expressions, $e_1 := \pi + \varphi$ and $e_2 := 2 + 4$, are valid expressions and the role of the `+` symbol is absolutely clear. However, in computer science, and especially in strongly typed programming languages, this expressiveness is not always easily represented. In OCaml, for example, the `+` operator should be exclusively employed to compute addition between integers and the `+. operator` (the dot is part of it) to compute the same operation over floating numbers. However, the `double` function, such that `double x = x + x`, should have two implementations to both work with the type `int` and the type `float`. This is quite annoying, in particular, if we want to go further and want to write the function `doubles x y z = double x, double y, double z`, which should have 8 different implementations (with 8 different function names) for all the possible combinations of `int` and `float`.

The goal of type classes is to solve this problem: we define only one ad-hoc polymorphic function \mathcal{F} , give an implementation of it for the targeted types and charge the elaborator to find the right implementation of \mathcal{F} regarding the argument applied to it.

1.2 A zoom on type classes

The first functional programming language to introduce type classes was Haskell [5]. A type class is a type structure that is added to the set of types of the language and an instance is an implementation of a type class. In Haskell the classic operators `+`, `*`, `-`, `abs`, `signum`, `fromInteger`, are all methods of a type class called `Num` and `Double`, `Float`, `Int` and `Integer` are all instances of `Num`. This means that $e_1 = \pi + \varphi$ launches a type-class resolution problem where the goal is to find the implementation of the `+` operator over the `Float` type (since π is a float). A similar search is done when we want to compute $e_2 = 2 + 4$, where this time `+` is applied to the `Int` type.

In Coq, type classes were introduced back in 2008 by Sozeau and Oury [7]. If we want to overload the addition symbol `+`, we can build, the type class `Add` in the following way:

```
Class Add (T : Type) := { (+) : T -> T -> T }.
```

As expected, `Add` takes a type `T` as an argument and has the `+` method with the signature `T -> T -> T`. Now we can call `+` on any type `Ti` and compute the result of an addition of this particular type if there exists an implementation of `Add` for `Ti`. By means of example, we can create an instance of `Add` for the type `nat` in the following way

```
Instance addNat : Add nat := {(+) := Nat.add}.
```

where `addNat` is the name of the newly created instance and the implementation of `+` is `Nat.add`¹. Thanks to this implementation, we are allowed to compute `2 + 4` and get `6` as a result. On the other hand, it is not yet possible to get the result of `3.14 + 1.61` since we don't have an implementation of `Add` for the type `float`.

Since we are working in Coq, a natural question that may come to mind is *why not using type classes to also contain theorems and let instances carry their proof?* In fact in Coq, the idea of putting proofs in type classes/instances is becoming more and more common in libraries with very complex inter-dependencies between data structures. Type classes in fact not only allow a cleaner representation of these dependencies but also the possibility of doing an automatic proof search.

For example, we can modify the `Add` type class defined before and add the proof that any time we are computing `a + b`, then the same result can be obtained by commutativity if we compute `b + a`. In Coq's syntax we obtain something like this:

```
Class Add (T : Type) := {
  (+) : T -> T -> T;
  add_is_comm: forall (a b : T), a + b = b + a
}.
```

The theorem `add_is_comm` uses the overloaded symbol `+` of the type class `Add` to represent the commutativity of `+`.

1.3 Simple and Complex instances

In the world of type classes, we can classify instances into two different categories. On the one hand, the user may want an instance to be a direct implementation of that type class with no further constraints. The `addNat` instance defined above is what we call a *simple* instance. On the other hand, there are instances where some hypotheses need to be verified before being fully applied. Let's suppose we want to work with addition over the `prod` type, noted `(_ * _)`. Let `P1: (T1 * T2) = (a, b)` and `P2: (T1 * T2) = (c, d)`², we may want to create an instance for `Add` for the type `prod` such that `P1 + P2 = (a + c, b + d)`. This implementation is possible only if it is possible to add `a` and `c` (resp. `b` and `d`): in other words, if there is an instance for `Add` over the type `T1` (resp. `T2`).

The `Add` instance for the type `prod` is:

¹We assume `Nat.add` to be the binary function computing the addition between two values of type `nat`

²`P1` and `P2` have the same type which is `(T1 * T2)`

```

Instance addProd (T1 T2 : Type) `(i1: Add T1, i2: Add T2) : Add (T1 * T2) := {
  P1 + P2 := match P1, P2 with
    (a, b), (c, d) => (a + c, b + d)
  end;
  add_is_comm := ...
}.

```

This instance says that a pair of type $(T1 * T2)$ is an instance of `Add` as long as there exists an implementation of `Add` for both the type `T1` and `T2` (in the code snippet, these constraints are labeled respectively `i1` and `i2` in the code snippet).

It is maybe clearer to make an example to point some of the tasks accomplished by the elaborator, going from the filling of the implicit type information to the type-class resolution. Let's take the Listing 1.3.1 and let $G = (3, 2) + (1, 0)$ the goal to be typechecked.

Listing 1.3.1: Example of type class resolution

```

1 Check ((3, 2) + (1, 0))
2 Check ((+) ?Type ?Instance (3, 2) (1, 0))
3 Check ((+) (nat * nat) ?Instance (3, 2) (1, 0))
4 Check ((+) (nat * nat) (addProd nat nat addNat addNat) (3, 2) (1, 0))
5 ((3, 2) + (1, 0)) : prod nat nat

```

The second line of this code shows that, in fact, the original term contains two holes corresponding respectively to the type `?Type` applied to the `+` operator and to the instance of `Add` applied to the `?Type`. The elaborator will start at first to fill the hole `?Type` which, by inference rules, is set to `nat * nat` (see line 3). From that type the type-class solver can start to fill the second hole `?Instance`. The instance to be applied on the type `Add (nat * nat)` is `addProd` where `T1: nat` and `T2: nat`. Now the constraints `i1` and `i2` are both replaced by the sub-goals `Add nat` and `Add nat`, which are both satisfied by the instance `addNat` (see line 4). The type of the original goal (line 1) is shown at line 5.

Note: In next sections, when talking about instances, we will use the words *constraints*, *hypothesis* (and later *premises* when reasoning in term of logic programming) as synonyms.

1.4 Parallel with object-oriented languages

The usefulness of type classes is uncontested for researchers working with functional programming languages, but to convince developers who are more used to working with object-oriented programming languages (OOP), we can draw the following relations between these two paradigms:

$$\text{type class} \Leftrightarrow \text{interface}$$

$$\text{instance} \Leftrightarrow \text{concrete class}$$

A Java's (Scala ...) user can at any time translate the notions of type classes and instances with interfaces and concrete classes with the difference that an instance (concrete class) cannot exist without a type class (interface). It is also interesting to talk a little about parametric polymorphism which in OOP can be translated into the generic methods parametrized by a type `T`. We want to finally point out that the elaboration process explained in this report can be transposed rather easily to any programming paradigm. For example, in object-oriented languages the call of an object's method will launch a search to find its concrete implementation among the set of methods defined for the type of that particular object.

1.5 Coq's type-class solver

In this section we cover the state of the art for the Coq system by explaining some of the most important components of the Coq's type-class solver. This will be the basis for better understanding the choices that have been made in our type-class solver prototype. In particular, we aim to provide a definition of:

- what a database of instances is, why we need it, and how goals are solved

- what the visibility of a type class is and how the context of a proof can introduce new local instances
- what hint modes are and why we could need them in type-class definitions

1.5.1 Database of instances

It is fundamental to store each instance in a database so that the type-class solver is able to find them while solving a goal. In Coq, the default database of the instances is called `typeclass_instances` and it is represented as a dictionary associating each type class with the list of its instances.

The list of instances of each type class T_C is traversed any time a goal G has to be solved. In this case, the solver tries to *apply* one by one the instances of the list. An instance can be applied to G if it unifies (see Section 4.5) with G . It is important to note that the order in which instances are stored may impact the time of resolution and the type of the solution. To control the order in which instances are added to the database, Coq allows the user to explicitly assign a priority to an instance. This priority can be set by simply adding a `| N` (pipe N , N is an integer) after the type of the instance. Let I_1 and I_2 be two instances of the same type class. If I_1 has priority N and I_2 has priority M , then I_1 has a higher priority if $N < M$. If $N = M$ then I_2 has been declared before I_1 then I_2 will be placed before I_1 in the list of instances. Finally, if the instance has no user-defined priority, Coq computes the priority thanks to the following heuristic: the priority of an instance I is given by the number of variables in the type of I that do not appear as arguments of the type class implemented by I .

To better understand the priority system in Coq, we can take the instances defined in the following code snippet.

```
Class A (n: nat) = {...}.
Class B (n: nat) = {...}.
Instance A1' `(B 2) : A 1 = {...}.
Instance A1 `(B 1) : A 1 = {...}.
Instance A5 : A 5 | 5 = {...}.
Instance B2 : B 2 = {...}.
```

In that example, only the instance `A5` has the user-defined priority 5, the other instances' priorities will be attributed automatically by Coq. The priorities for both `A1` and `A1'` are equal to 1 since both instances have one hypothesis (respectively `B 2` and `B 1`) not used in as argument of the implemented type class. However, since `A1` has been defined after `A1'`, `A1` will be placed before `A1'` in the list of instances for `A`. Regarding the instance `B2`, since it has no constraints, it will receive priority 0. The database of instances will have the following shape:

```
Type class A ->
  apply A1 (priority 1, pattern A 1)
  apply A1'(priority 1, pattern A 1)
  apply A5 (priority 5, pattern A 5)
Type class B ->
  apply B2 (priority 0, pattern B 2)
```

The previous code is printed by Coq via the command `Print HintDb typeclass_instances` which allows not only to see how instances are stored in the database, but also what pattern the input goal should unify so that the considered instance can be applied. For example, if the goal is `A (0 + 1)`, at first Coq's reduces the sub-term `(0 + 1)` to `1` and then the search is launched on the equivalent goal `A (0 + 1)`.

1.5.2 Goal resolution

In this paragraph, we want to show the link between this database and the type class resolution. For that purpose we propose two examples based on the small database depicted in the previous sub-section.

Let's start with $G_1 = A\ 1$. This goal is asking the type-class solver to find if there exists an instance of the type class `A` that can be used on G . The first that could be used is `A1` on the constraint to satisfy the constant `B 1`. This new sub-goal is tried for resolution by looking inside the database of instances for the type class `B`. However no solution for `B 1` is found. The algorithm of type-class resolution has encountered a fail: the application of `A1` has led to a dead end. However, there might exist another possible instance applicable to the original goal leading to a success. In fact the search algorithm of Coq is based on a DFS search: we have a goal to solve, we apply an instance and try to recursively

solve its constraints. If a constraint has no solution the algorithm backtracks to the nearest sub-goal with other applicable instances. This process is repeated recursively until either a solution is found, or there exists no solution, so the original goal is not provable. We can in fact see the set of instances and their constraints as a tree where a node N represent a goal, an edge corresponds to the application of an instance of that goal. The children of N are the list of constraints to be satisfied. Going back to the previous example, the choice of $A1$ is not valid, since the constraint $B1$, has no solution. After backtracking, $A1'$ is tried on the original goal. This generates the sub-goal $B2$ which is solvable after the application of the instance $B2$. The solver has finally found a valid solution in the exploration tree: $A1' B2$. This elaborated solution is finally transmitted to the kernel for validation.

In this second example, we want to show how the context of a proof can create new local instances and how these instances are added to the database to solve goals. Let $G_2 = B2 \rightarrow A1$ be our second goal to be solved. The first thing the Coq's solver does is to introduce the hypothesis $B2$ in the context of the current proof. It is important to note that any instance added in the context, has the highest priority in the database. After this introduction, the resolution starts by looking for an instance to apply on $A1$, as before, the instance $A1$ is tried generating the sub-goal $B2$. This time, $B2$ has a solution since there is a hypothesis in our context proving exactly this sub-goal. Once the proof is over, the instance $B2$ is removed from the database.

To resume, the search algorithm for type-class resolution is based on a DFS search in which instances are applied following their priorities. Any time a dead end is reached during the exploration of the tree, the algorithm will try to backtrack to the previous node with at least one child not yet explored. Finally a solution is found if we have reached a leaf of the tree and it is possible to prove it. Otherwise if the tree has been entirely explored and no solution is found, a failure is returned.

1.5.3 Hint modes for type classes

During type-class resolution, it may happen that the arguments of a type class are not all defined, they are holes that should be fixed during the search. For example, in the goal $G = A _$, the first argument passed to the type class A is not defined, it is called a flexible term. This means that during the search the $_$ hole is considered a meta-variable that can be replaced with any other Coq's term, this operation is called unification and is explain in detail in [Section 4.5](#). For instance, the solutions of G are both $A1'$ $B2$ and $A5$, since G can be unified to any goal $A N$, for all $N \in \mathbb{N}$. However, in Coq the user may want to control the nature of the arguments passed to a type class. This is possible thanks to the so-called modes, where before doing the search, the elaborator makes some checks on the arguments. The existing modes in Coq are explained in [\[17\]](#) and are represented by the symbols $-$, $+$ and $!$ such that: “*mode - matches any term, mode + matches a term if and only if it does not contain existential variables, while mode ! matches a term if and only if the head of the term is not an existential variable*”.

Listing 1.5.1: A simple code causing infinite loop

```
Class Num (I : nat).
Instance is_zero : Num 0 | 10.
Instance is_num (n : nat) `(Num n) : Num (S n) | 0.
```

Let's take the code in [Listing 1.5.1](#) where we represent natural numbers with the type class `Num`. The instance `is_zero` is an implementation of `Num` for the natural 0 whereas `is_num` says that for all n , $(S n)$ is a number if it is possible to prove `Num n`. In the example, to make modes useful, we have forced the priorities of both instances such that `is_num` has higher priority than `is_zero`. Now let's consider the goal $G = \text{Num } ?n$ (for commodity, instead of putting an underscore, we call $?n$ the hole after `Num`). To solve G , Coq will start to apply `is_num` being the first applicable instance of the database. This choice fixes $?n$ to the term `S ?n1` and the sub-goal $G' = \text{Num } ?n1$ will be launched for resolution. Due to the given priorities, the solution that Coq applies to G' is `is_num` which fixes $?n1$ to `S ?n2` and generates the new sub-goal `Num ?n2`. We can see that the resolution of G gives rise to an infinite loop. However, if we try to solve the goal $G_2 = \text{Num } 2$, the solution `is_num 1 (is_num 0 is_zero)` is immediately found. We can remark that any time we call `Num` with a flexible argument, an infinite loop is thrown and if that argument is not flexible (ergo rigid) a solution is found. It is, therefore, interesting to add the constraint that a goal on `Num` can be solved only if its first argument has the mode $+$. Modes can be added through the following command:

```
Hint Mode Num + : typeclass_instances
```

where `typeclass_instances` is the name of the instance database.

By default, all the arguments are marked with the mode `-` if the considered type class has no user-defined mode.

2 Elpi: Embeddable λ Prolog Interpreter

The Coq's type-class solver is in fact a very powerful tool to allow instance search. However, as explained in Section 2.1, we think that the logic programming language Elpi is a good choice to provide an alternative to the type-class solver. Before going to these motivations, we would like to give the reader a rapid introduction to Elpi and Coq-elpi to fuel the reasons pushing us to undertake our project.

Introduction to logic programming Elpi is a logic programming language (LP) based on λ Prolog [1] equipped of Constraint Handling Rules [13]. Logic programming languages are based on a database of clauses and the user can interact with them through the so-called queries. Each clause in the database is an implementation of a predicate and must respect its signature. For example, the factorial function can be defined as follows:

```
1 type factorial int -> int -> prop.
2 mode (factorial i o).
3 factorial 0 1.
4 factorial N Res :-
5   N > 0, N1 is N - 1, factorial N1 Res1, Res is N * Res1.
```

In this code we see the definition of the predicate `factorial` which has two arguments. The first one is an *input* argument, it means that this argument should always be rigid to avoid execution errors; the second argument is in output and will contain the output of the computation. An example of query is `factorial 3 R` where, after the execution of the program, the variable `R` will store the result. Clauses in logic programming are divided in two categories:

- *facts*, clauses with no premises, such as line 3 in the previous code snippet. A fact is applied if the all the arguments of the query unify with the argument of the fact. The query `factorial 0 R` unifies with line 3 and `R` is fixed to 1, which is the expected value for $0!$ math formula.
- *rules*, clauses with premises, such as line 4. In logic programming a clause is written on the form `head :- body` where `head` is the head on the clause which should unify with the current query and `body` is the list of conditions, called premises, to be satisfied in order to apply the current rule. A rule can be read “*if body then head*”. In Elpi this is equivalent to `body => head`. The query `factorial 3 R`, doesn't unify with line 3, since the argument 3 does not unify with 0, but it does with line 4. In this case, `N` is unified with 3, the proposition `N > 0` holds, `N1` is evaluated to 2, the recursive call to `factorial 2 Res1` will fix `Res1` to 2 and finally `Res` is evaluated to 6, the result of $3!$.

In Elpi, variables start with an uppercase letter, otherwise the considered term is a constant. Moreover, Elpi does not allow free variables, that is, all variable is bound, and therefore quantified. The \forall -quantification is introduced with the `pi` keyword and the \exists -quantification with the `sigma` keyword. If not specified differently, any variable in the body of a clause is implicitly \forall -quantified before the `head` of the clause. This means that the second implementation of the `factorial` function is equivalent to

```
pi N Res N1 Res1\
  factorial N Res :-
    N > 0, N1 is N - 1, factorial N1 Res1, Res is N * Res1.
```

In Elpi we can pass anonymous rules to higher order predicates. Let `forall` be the predicate with signature `pred forall i:list A, i:(A -> prop)` which checks if all elements of the list satisfy a given predicate. Now, let's take the query

```
forall [1, 2] (x\ sigma TMP\ TMP is x mod 2, TMP = 0)
```

where `x` is the binder replaced at each iteration of `forall` to represent the current element of the list and `TMP` is a \exists -quantified variable (fresh at each iteration).

A final rule that is proper to logic programming languages and has no equivalent in relational model, is the cut operator, represented with the `!` symbol (to be read *bang*). The search done in Prolog to solve a query is a DFS search, where clauses are tried one by one for unification until either a fail or a

success is encountered. In the former case, the algorithm, such as the Coq's type-solver, tries to find a new possible solution using backtracking. In certain scenarios, it may be the case, that, when a clause is chosen during the search to solve a certain goal, it is better to forbid backtracking on that particular node of the search tree. This operation of discarding some solutions can be done with the `cut` operation.

A final remark, in Elpi, such as in λ Prolog, it is allowed to freely nest propositions using the `=>` operator. Thanks to this operator we can build proposition on the form `(A => B), C` where the clause `A` is charged locally and visible only by `B`, this is equivalent to the prolog syntax `assert A, B, retract A, C`. In this way we can encode more complex formulas than the Horn clauses, due to the nesting of `=>`, this set of clauses we can build are called Harrop formulas (chapter 3 of [9]).

2.1 Motivation to use Elpi for type-class resolution

Since the beginning of this report, we have talked about the need for the elaborator in Coq, on how the elaboration can be useful to avoid writing *complete* terms since some missing information can be automatically inferred through the rules coming from the type theory. The actual Coq's type-class solver is, therefore, particularly powerful and lots of Coq's libraries benefit from it. However, the Coq's type-class solver during the time has become more and more complex and more and more difficult to be managed. The user can interact with it through different flags allowing to tweak a bit the search strategy of the solver. If we take a look at the official page of the Coq's type-class solver³, we can count about a dozen flags to change the search algorithm from a DFS to a BFS [6], verify if the solution of the search is unique, add non-backtrack search when a solution is found on the current goal, check that no suspended goal remains and so on.

However, our claim is that all of these flags are on the one hand too many to be fully understood and efficiently exploited by the power user, and on the other hand, these flags are too few to have full control over the type-class solver. The first remark has not to be commented, since it is quite clear that finding the right combination of the 2^{10} different combinations of flags is undoubtedly too difficult; the second claim, even if a bit oxymoronic with the first, needs perhaps a small clarification. As the developer of his library, a Coq's power user often may wish 1. to be warned if an instance is meaningless with respect to its hint modes (Section 4.4.1); 2. to have a control on the backtracking on particular type classes, called *deterministic* (Section 4.4.2); 3. to build his custom rules of search for particular goals in order to improve the time of our search (Section 4.6); 4. and so on depending on his needs.

All of these are just a few the problems that type-class users encounter while developing their library in Coq. Our goal is to propose a more flexible and predictable type-class solver written in Elpi. As explained in the previous section, Elpi is well suited to solve queries using the DFS algorithm from the set of clauses of a database; clauses in the database can be added and removed at any time at runtime; binders allow to create dependencies between clauses... All of these ingredients are a good starting point to emulate Coq's solver. The only tool we need is to be able to communicate with Elpi in Coq program, and this is possible thanks to Coq-elpi.

2.2 Coq-elpi: Elpi for Coq

In this last section before introducing our Elpi prototype for type-class resolution, we would like to spend a few words on the Coq-elpi plugin. This library bridges the communication between Elpi's and Coq worlds. Firstly, the runtime of Elpi in Coq is controlled by OCaml code whose goal is to manipulate the Coq's terms and translate them into the Elpi representation. Let $\llbracket C \rrbracket$ be a function taking a Coq term `C` and translating it into its corresponding Elpi representation. The most important and frequent terms that are translated between these two languages are:

Term	In Elpi	In Coq
Constant	<code>global (const C)</code>	<code>C</code>
Inductive type	<code>global (indt T)</code>	<code>T</code>
Inductive constructor	<code>global (indc T)</code>	<code>T</code>
Function application	<code>app $\llbracket F \rrbracket$, $\llbracket x_1 \rrbracket$, $\llbracket x_2 \rrbracket$, $\llbracket \dots \rrbracket$, $\llbracket x_n \rrbracket$</code>	<code>F x_1 x_2 ... x_n</code>
Lambda abstraction	<code>fun 'x' $\llbracket T \rrbracket$ (x\$\llbracket Body \rrbracket$ x)</code>	<code>fun (x: T) => Body</code>
Let-introduction	<code>let 'x' $\llbracket T \rrbracket$ $\llbracket Value \rrbracket$ (x\$\llbracket Body \rrbracket$ x)</code>	<code>let (x: T) := Value in Body</code>

There are also representations for the `match` and `fix` definitions in Coq, but they are not useful for our purposes. Note also that the 'x' in both the lambda abstraction and in the let-introduction are both

³<https://coq.inria.fr/refman/addendum/type-classes.html>

placeholders used for pretty printing. The binder associated with `fun` and `let` are both transposed inside the body of the function via `x\`. A last remark, is that constants and inductive types are both called `gref`, a Coq's `gref`, in Elpi can be obtained via the dedicated syntax `{:gref X}`, where `X` represents a constant, an inductive type, or an inductive constructor.

We want to point that `lp:{{ Elpi }}`⁴ is a quotation to enter Elpi code inside Coq and `{{ Coq }}` is a quotation to write Coq code inside Elpi. For example, in Elpi, we can write the following:

```
X = {{list lp:T}},
T = {{nat}}
```

where `X` is unified to the Coq's term `C = {{list lp:T}}`, where in `C`, `T` is an Elpi variable. In the end, `X` is unified to the Elpi term `app [global (indt «list»), global (indt «nat»)]`

Finally in Coq-elpi there exists a bunch of builtin functions callable in Elpi that are listed principally in the file listed in [18] and [15].

3 The Elpi's type-classes solver prototype

The prototype we are working on is a type-class solver written in Elpi. In Elpi we have created a database, called `tc.db` where all the Elpi's clauses are stored. To make type-class resolution, we need a way to *translate* the Coq's instances into Elpi's clauses, to add them into `tc.db` and to have a tactic to override the instance search of Coq with Elpi's.

3.1 Instance Compilation

A first interesting point from which we want to start, is the way instances are programmatically translated (or compiled) from their Coq's representation to Elpi's. This operation is done by looking at the type of the instance we are dealing with. In Elpi we can obtain the type of the `gref` of an instance via the predicate `coq.env.typeof GR T`, where `GR` is the `gref` and `T` is the term representing the type of `GR`. During the development of our prototype, we have experienced two different implementations of Coq's instance compilation in Elpi. We want to show both of them, the first is a simpler version and therefore a good introduction to how complex Coq's terms can be abstracted in Elpi.

3.1.1 The simple tc predicate

In this first implementation, we work with `tc` predicate whose signature is `pred tc i:term, o:term`. This predicate takes as input the goal to be solved and returns the instance solving it. To compile an instance in Elpi we use the `compile` predicate the code of which is presented below in a simplified version.

Listing 3.1.1: A simplified compiler of Coq's instances for Elpi

```
1 pred compile i:term, i:term, i:list term, i:list prop, o:prop.
2 compile InstTerm (prod _ Ty Body) Vars Premises (pi x\ NewClause x) :-
3   pi x\ sigma NewPremise NewPremises\
4     if (has-class Ty)
5       (compile x Ty [] [] NewPremise,
6         NewPremises = [NewPremise | Premises])
7       (NewPremises = Premises),
8     compile InstTerm (Body x) [x | Vars] NewPremises (NewClause x).
9 compile InstTerm Goal RevVars RevPremises NewClause :-
10  std.rev RevPremises Premises,
11  std.rev RevVars Vars,
12  coq.mk-app InstTerm Vars Solution,
13  NewClause = (tc Goal Solution :- Premises).
```

Let `I` be the instance taken into account for compilation, the `compile` predicate needs 5 arguments `[InstTerm, InstType, Vars, Premises, NewClause]`:

`InstTerm` is the term representing `I` and can be obtained by `coq.locate I InstTerm`

⁴`lp:X` is a shortcut if `X` is an Elpi's atom

InstType is the type of **I**,

Vars is the list of \forall -quantified variables in Elpi corresponding to the \forall -quantified variables in the Coq term

Premises is a list of Elpi proposition corresponding to the constraints of the current instance

NewClause is the clause representing the Elpi compiled version of **I**

The two clauses implementing `compile` aim to respectively treat \forall -quantified terms and the application of terms as explained in the two following paragraphs.

\forall -quantified terms If the term to be treated is $\forall (x : \text{Ty}), \text{Body}$, we have to quantify a fresh \forall -variable to represent x inside the new Elpi's clause so that it will have the shape `pi x \ NewClause x`. The `NewClause` variable is crafted inside the body of the first implementation of `compile`. The \forall -quantified variable x at line 3, represents the Elpi's term that is applied to the `Body`. x is added to `Vars` since it will be needed to produce the solution of the instance we are compiling. Next, if the type `Ty` of the quantified variable represents a type-class goal, we compile and add it to the list of `NewPremises`. Finally we do a recursive call to `compile Body` where the list of variables passed as arguments sees x prepended to the list `Vars`.

Application of terms The base case of `compile` receives a term of the form `app _`. At first the two lists of `Vars` and `Premises` are reversed to get the right order (note that in the recursive case of `compile`, the new \forall -quantified variable and the new premises are each time prepended to the list to get higher performances). The new `tc` clause will be on the form `tc Goal Solution :- Premises`, where `Goal` is the type of the instance where the Coq's \forall -quantified variables have been replaced with the Elpi's binders, `Premises` is the list of premises created as explained before. In the end, `Solution` is the solution to be returned for the current goal. It is built thanks to the `coq.mk-app` function, so that it has the following Elpi's shape: `app [InstTerm | Vars]` if the list `Vars` is non-empty, otherwise `Solution` will be simply the term `InstTerm`.

Examples with `addNat` and `addProd` To clarify better the role of the `compile` predicate, we will take the two instances of the type class `Add` from Section 1.

Let's start with the *simple* instance `addNat`. To compile it, we need its type. Let `Gr` be the variable associated to the `gref` of the current instance, we can get the term associated with the instance `InstTerm = global Gr` and its type `coq.env.typeof Gr Type`.

The initial call to `compile` is:

```
compile InstTerm Type [] [] Clause
```

The two empty lists correspond to the lists of variables and premises. We can note that `InstTerm` is `addNat` and `Type` is `Add nat`. This term is not a \forall -quantification, therefore, we can apply the base case of the `compile` implementation. The resulting clause is:

```
tc {{Add nat}} {{addNat}}
```

This fact says that any goal unifying with `Add nat`, will output `addNat` as solution.

Instances with constraints are way more interesting to compile. Let's consider the instance `addProd`. Its Coq's type is:

$$\forall T1 T2 (i1 : \text{Add } T1) (i2 : \text{Add } T2), \text{Add } (T1 * T2)$$

Its type in Elpi is:

```
prod `T1` (sort (typ «add.5»)) c0 \
  prod `T2` (sort (typ «add.6»)) c1 \
    prod `i1` (app [global (indt «Add»), c0]) c2 \
      prod `i2` (app [global (indt «Add»), c1]) c3 \
        app [global (indt «Add»), app [global (indt «prod»), c0, c1]]
```

It contains 4 nested \forall -quantifications, two of which are type-class hypotheses. When we start to build the Elpi's clause for `addProd`, we go through the first two Coq's variables `T1` and `T2`. The recursive call to `compile` after these first two iterations will make the following recursive call:

```
compile {{addProd}} Type [x2, x1] [] Clause
```

where `Type` is

```
prod `i1` (app [global (indt «Add»), x1]) c2 \
  prod `i2` (app [global (indt «Add»), x2]) c3 \
    app [global (indt «Add»), app [global (indt «prod»), x1, x2]]
```

We see that T1 and T2 have been consumed and their respective binders `c0` and `c1` have been replaced by the two \forall -quantified variables `x1` and `x2` of Elpi; these two variables are stored in the list `Vars` in reversed order. The clause partially built is `pi x2 x1 \ Clause x1 x2`. The sub-term of `Type` to be treated, has `i1` as \forall -quantified variable which is a type class. The algorithm starts to abstract `i1` with a fresh Elpi variable, say `x3`. The `has-class` function applied to the type of `i1` succeeds and therefore `app [global (indt «Add»), x1]` is recursively compiled into a new premise of the final clause.

As a small digression, the type of `i1` has no \forall -quantified variables, so its compilation will simply pass through the base case. Similarly to the `addNat`, will be compiled to:

```
tc {{Add x1}} x3
```

where both `x1` and `x3` are \forall -quantified variables in Elpi. The variable `i2` is compiled in the same way as `i1` using the fresh variables `x2` and `x4` instead of `x1` and `x3`.

We are left with the last sub-term `app [global (indt «Add»), app [global (indt «prod»), x1, x2]]`⁵, which is solved in the base case of the `compile` predicate. The arguments of this recursive call are now:

```
compile {{addProd}} {{Add (lp:x1 * lp:x2)}} [x4, x3, x2, x1] Premises Clause
```

where `Premises = [tc {{Add x2}} x4, tc {{Add x1}} x3]`.

To finish the compilation of `addProd`, we reverse the lists of the premises and the \forall -quantified variables. The variable `Goal` is unified to `{{Add (lp:x1 * lp:x2)}}` and `Solution` is `app [{{addNat}}, x1, x2, x3, x4]`.

The final clause built after compilation is:

```
pi x1 x2 x3 x4 \
  tc {{Add (lp:x1 * lp:x2)}} (app [{{addNat}}, x1, x2, x3, x4]) :-
    tc {{Add x1}} x3,
    tc {{Add x2}} x4.
```

equivalent to

```
tc {{Add (lp:X1 * lp:X2)}} (app [{{addNat}}, X1, X2, X3, X4]) :-
  tc {{Add X1}} X3,
  tc {{Add X2}} X4.
```

Thanks to these two instances we can now solve, in Elpi, the goal

```
tc {{Add (nat * (nat * nat))}} Solution.
```

We apply the clause for `addNat` which will unify `X1` and `X2` to `nat` and `nat * nat`. This will produce the partial solution `{{addNat nat (nat * nat) lp:X3 lp:X4}}`. Now we have to solve the two premises.

The first premise is satisfied by the rule `addNat` fixing `X3` to `addNat`. The second premise, after two recursive calls, will produce the solution for `X4` which is `{{addProd nat nat addNat addNat}}`. The final solution of the original goal produced by the Elpi is;

```
addProd nat (nat * nat) addNat (addProd nat nat addNat addNat)
```

3.1.2 Elpi's vs Coq's modes

Before introducing the second implementation of the `tc` predicate, we prefer to provide some explication on how modes are represented in Elpi. Elpi modes are two and called `input` and `output`, we have seen them rapidly in the signature of the few predicates we have defined in some of the previous sections. An argument A_i of a function in input mode (syntax `i:`) means that any time we call that predicate A_1 should be a rigid term, otherwise at runtime, Elpi will throw an error. On the other hand, an argument A_o is in output mode (syntax `o:`) if the user is allowed to pass a either a flexible or a rigid variable. In particular output mode is used to contain the result of a computation and if we provide a rigid variable V to A_o , this is like a check to verify that the computation returns a variable V' at A_o that unifies with V . These modes are semantically different from Coq's one, and we have decided to keep Elpi's one, since we think it is more meaningful in the context of type-class problems. In our solver, we do the following pairing: the `+` and `!` modes of Coq are translated in Elpi's `input` mode and vice versa, the `-` mode of Coq is translated into Elpi's `output` mode.

⁵Recall that this term can be equivalently represented with the term `{{Add (lp:x1 * lp:x2)}}`

The modes of a type class can be obtained in Elpi via the command `coq.hints.modes` which returns a value of type `list (list hint-mode)`. This is because a type class in Coq can have multiple modes (even if in rare cases) and each of them is a list $[+, -, !]^n$ where n is the number of arguments of the type class taken into account. The type `hint-mode` in Elpi is defined as follows:

```
kind hint-mode type.
type mode-ground hint-mode. % No Evar,      AKA !
type mode-input hint-mode.  % No Head Evar, AKA +
type mode-output hint-mode. % Anything,     AKA -
```

Let's consider the following code snippet as example.

```
Class Clone (i1: Type) (i2: Type).
Instance clone (v: Type) : Clone v v.
Hint Mode Clone + -: typeclass_instances.
Hint Mode Clone - +: typeclass_instances.
```

The instance `clone` is meant to take a value v as the first or second argument and output a copy of it in the second or first argument. Just for clarity, we can analyze some goals referring to this type class to see what Coq does.

- `Check (_ : Clone nat nat)` will typecheck to `clone nat`
- `Check (_ : Clone _ nat)` will typecheck to `clone nat` and the second hole is fixed to `nat`
- `Check (_ : Clone nat _)` will typecheck to `clone nat` and the second hole is fixed to `nat`
- `Check (_ : Clone nat bool)` will not succeed since the two arguments are not the same
- `Check (_ : Clone _ _)` will produce no result since there is no mode matching

In Elpi the query `coq.hints.modes {{:gref Clone}} "typeclass_instances" Modes` sets `Modes` to `[[mode-output, mode-ground], [mode-ground, mode-output]]`. In the version of the prototype using the simple `tc` predicate, mode verification is done via a very high-priority ad-hoc rule for each type class. We put this highest priority in order to test the validity of a goal with respect to its mode, before starting its resolution. The shape of that kind of clause for the type class `Clone` is:

```
1 tc {{Clone lp:T1 lp:T2}} _ :-
2   L = [[T1], [T2]],
3   std.forall L (c3\ std.exists c3 var), !,
4   coq.error "Invalid mode for" {{Clone lp:T1 lp:T2}}
```

Where the list `L`, at line 2, is built from the list of modes of `Clone`: the sub-lists contain the set of arguments in either `+` or `!` mode. In particular the first sub-list of `L` contains `T1` since the first hint mode declaration of `Clone` wants the first argument to be in `+` mode; a similar reasoning can be done for the second sub-list. At line 3, we test if every sub-list of `L` contains at least one variable that is flexible. In this case, a fatal error is thrown with a message saying that the arguments of `Clone` are not valid regarding the defined modes.

3.1.3 Drawbacks of using the simple `tc` predicates

Even though the `tc` predicate and its clauses implementing Coq's instances seems like a good approach to type-class resolution, there are some drawbacks leading us to consider a more efficient implementation. This implementation is based on the creation of an ad-hoc predicate per type class with the signature `tc-TC_Name A1 A2 ... An Sol` where `TC_Name` is the name of the type class, `A1 A2 ... An` are the arguments passed to `TC_Name` and `Sol` is the solution to apply to the current goal.

Too many rules for mode verification The mode verification we use in the `tc` implementation respect the chosen semantic for mode verification. It allows not only to check the correctness of the arguments with respects to their mode, but also to throw an error justifying why the search failed. However, if we work on a very big Coq's library with a considerable number, say N , of type classes, we need to create N clauses to make mode verification for all the N type classes. This will cause the search engine to try any of these clauses for unification, which, in terms of performance, is not optimal. Consider also that even without modes, having only one predicate for N type class is too time-consuming. We can remark, however, that Elpi has builtin mode verification to check their flexibility/rigidity. This

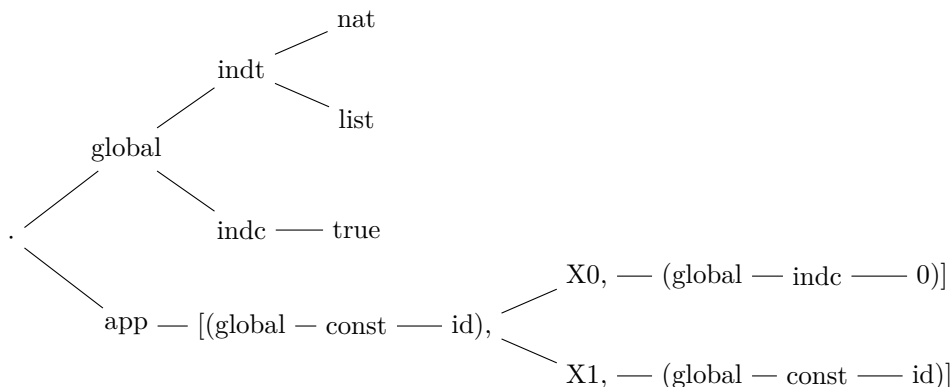


Figure 3: Discrimination tree for Listing 3.1.2

verification has very high performance, and therefore, to make this check, it is needed to spill the arguments of any type class as parameters of new ad-hoc predicates.

Inefficient indexing An interesting feature of Elpi is the possibility to improve the search of clauses by using an indexing algorithm on the arguments of the predicate. This strategy is based on a dictionary where values represent the hashed prefix of a term and the keys contains the full term. This data structure allows requires an amortized constant time to access a key in the dictionary. On the other hand, we have to consider that in databases with lots of instances, the probability of having hash collisions will increase, causing a potential loss of time complexity.

In particular, in Elpi if we do:

```

:index (_ 4)
pred foo o:int, i:term.

```

we are indexing on the second argument A_2 of the predicate with depth 4. It means that the first 4 sub-terms of A_2 are used to discriminate against it. If A_2 contains more than 4 sub-terms, the classic process of unification will be applied to the 5th sub-term.

Listing 3.1.2: Elpi code for discrimination tree

```

foo 1 {{nat}}.
foo 3 {{list}}.
foo 2 {{true}}.
foo 4 {{id 0}}.
foo 5 {{id id}}.

```

For instance, let's code the small example above. The corresponding discrimination tree is depicted in Figure 3 where the terms associated with `nat` and `list` share the same prefix, and are discriminated with `true` since `true` is an inductive constructor `indc` and not an inductive type `indt`. Moreover, we can see that big terms, like the facts at line 4 and 5, have lengths bigger than 4 and, therefore, their 5th, ..., n^{th} arguments left for the classic unification.

Indexing is in fact interesting in instance search since, often, there is an argument among the parameters of a type class leading the search. It means that this argument is the one that should be taken into account when indexing. An example of this can be seen in the type class `Inj` taken from the library `stdpp` which is defined as follows:

```

Class Inj {A B} (RA : relation A) (RB : relation B) (f : A -> B) : Prop :=
  inj x y : RB (f x) (f y) -> RA x y.

```

This type class defines the injective property of functions. The arguments `A` and `B` are respectively the domain and the codomain of the function `f`, whereas `RA` and `RB` are two relations respectively on `A` and `B`. This implementation is equivalent to the math notation $\forall x y \in A, f(x) = f(y) \Rightarrow x = y$ (note that, to have more flexibility, instead of using the general equality relation between x, y and $f(x), f(y)$ in `stdpp` they prefer to use a general relation `RA` and `RB` on the domain and the codomain).

A lot of instances representing injective functions can be implemented, including the identity and the successor functions, the composition of injective functions and so on. These instances can be used to make an automatic proof search, as explained at the very beginning of this report. For example, the following goal can be automatically proved.

```
Goal Inj eq eq (compose (fun x => (S (id x))) (fun x => (S (id x)))).
```

In order to prove that $\mathcal{F} = \text{compose } (\text{fun } x \Rightarrow (\text{S } (\text{id } x))) (\text{fun } x \Rightarrow (\text{S } (\text{id } x)))$ is injective, we apply the instance `compose_inj` and then we verify that its left and right argument are both injective functions.

In this example we can see that to verify if the function is injective, we care about `f` while the arguments `A`, `B`, `RA` and `RB` are not so useful in this kind of instance search. That's why if we make an indexing on the 5th argument of `Inj` the search has better performances. However, doing indexing on that argument on the `tc` predicate becomes rapidly infeasible since the `Elpi`'s term for \mathcal{F} is simply too big to be efficiently indexed 63 bits: when lots of instances of `Inj` are implemented, lots of collision will be likely since a lot of the bits employed for the indexing are spent to encode the first *useless* sub-terms (the `elpi` term representing `f` is depicted in [Listing A.4.1](#)).

3.1.4 A better solution: ad-hoc predicate for type classes

As we can see, the `tc` predicate is good as a first implementation of `Coq`'s instances in `Elpi`, however, a better approach would be to create a new predicate for each type class in which the arguments applied to a type class are transformed into `Elpi`'s arguments. These predicates will solve the drawbacks mentioned before.

In `Elpi`, from version 1.18.0, we are allowed to declare new predicates with their signatures into a database on the flight via the predicate `coq.elpi.add-predicate`. This predicate is particularly useful since the signature (argument number and modes) of a type class is known only after its declaration. In this way, when a type class is defined, the new predicate can be added at runtime. For instance, the `Inj` type class will generate the predicate:

```
%
pred tc-Inj o:term, o:term, o:term, o:term, o:term, o:term.
```

We add the prefix `tc-` because predicates in `Elpi` should start with lowercase letters, since type classes can be declared with a starting uppercase, we guarantee to build a valid `Elpi`'s predicate. The `compile` function defined in its simplified version in [Listing 3.1.1](#) will remain unchanged, except for line 13, since `NewClause` will be generated with respect to the type class it is dealing with. Line 13 will change as follows:

```
app [global TC_Gref | Args] = Goal,
make-tc-name TC_Gref TC_Str,
std.append Args Solution ArgsSol,
coq.elpi.predicate TC_Str ArgsSol ClauseHead,
NewClause = ClauseHead :- Premises.
```

Where the `gref` of the type class (`TC_Gref`) and the list of arguments `Args` are extracted from the current goal. `TC_str` represents the string of the name of the new predicate we want to build; it is prefixed with `tc-` and is built from the `gref` of the type class; `ArgsSol` is the list of arguments to which the solution of the current goal has been appended. Finally, we can build the head of the clause (`ClauseHead`) via the `coq.elpi.predicate` API⁶.

In this way for example, the instances:

```
Instance id_inj {A} : Inj eq eq (@id A).
Instance compose_inj {A B C} R1 R2 R3 (f : A -> B) (g : B -> C) :
  Inj R1 R2 f -> Inj R2 R3 g -> Inj R1 R3 (compose g f).
```

are translated in the following way:

⁶its signature is `pred coq.elpi.predicate i:string, i:list any, o:prop`

```

pi c0 \
  tc-Inj c0 c0 {{eq lp:c0}} {{eq lp:c0}} {{id lp:c0}} {{id_inj lp:c0}}.

pi c0 c1 ... c8 c9\
  tc-Inj c0 c2 c3 c5 (app [{{compose}}, c0, c1, c2, c7, c6])
  (app [{{compose_inj}}, c0, c1, c2, c3, c4, c5, c6, c7, c8, c9]) :-
    tc-Inj c0 c1 c3 c4 c6 c8,
    tc-Inj c1 c2 c4 c5 c7 c9.

```

Thanks to this implementation we have better performances due to the builtin mode verification of `Elpi` and the faster clause discrimination due to the ad-hoc predicate. Moreover, in this particular case, we can also apply indexing to the 5th argument of `Inj` to further reduce the time of computation.

A little drawback of the `tc-XX` predicates is that in the current `Elpi` version, a predicate can only have one signature and, therefore, only one mode declaration is allowed for any type class. As said before, multiple modes are however rarely used and if a type class has more than one mode, our prototype will reject to compile it with a compilation error.

3.2 Instance priorities

As explained in [Section 1.5.1](#), the order in which instances are added in the database can change the performance of the search. To avoid this unwanted behavior, the user either writes instances in a specified order to use `Coq`'s default priority system of instances or adds an explicit priority to the instance. Our `elpi`'s prototype must respect the same instance order of `Coq` to obtain the same behavior the power user expects from the type-class solver. In `Elpi` we can achieve this by using clause grafting, by which clauses can be added *before* or *after* to a named clause.

```

kind clause type.
type clause id -> grafting -> prop -> clause.
pred coq.elpi.accumulate i:scope, i:id, i:clause.

```

The insertion of clauses in an `Elpi` database is done via the API `coq.elpi.accumulate` where we specify the `scope` (not useful for our needs), the `id`, which is the database in which we want to add our new clause and the `clause` we want to add. `clause` is a data type that takes:

- an `id`: the name of a clause, by default clauses have no name;
- a `grafting` which is either `before C_name` or `after C_name`, meaning that the clause will be added respectively before or after the clause named `C_name`, if not specified a clause is added at the end of the database;
- a `clause` which is the implementation of the clause we want to add in the database.

To reproduce the correct instance ordering in `Elpi`, we use the `hook` predicate, and we give 1000 implementations of it, each having a name representing its position.

Therefore the database will have the following clauses:

```

pred hook.
:name "1" hook.
:name "2" hook.
...
:name "1000" hook.

```

This means that an instance with priority 10 will be added *after* "10". Recall that any two instances I_1 and I_2 with same priorities in `Coq` are added so that I_2 is placed before I_1 if I_1 has been declared before I_2 . A clearer example of this is shown in [Appendix A.2](#).

In the end, instances with no user-defined priorities receive a priority computed by `Coq` with the heuristic explained in [Section 1.5.1](#). In `Elpi` this priority is computed by walking through the type of the instance, accumulating all binders of the \forall -quantified variables in the list `L` and returning the number of those binders not occurring in the goal.

A particular case: sections It is worth noting that in `Coq`, we can define sections where we can define an auxiliary context to better structure our code. In sections, instances are temporarily added

to their database, and at the end of the section, they are removed and re-added after all the section variables have been abstracted. This is because an instance depending on a context variable can have a priority that is different from the one it could have outside the corresponding context.

```
Section foo.
Context (A B C: Type)
  (R1 : relation A) (R2 : relation B) (R3 : relation C)
  (f : A -> B) (g : B -> C)
  (H: Inj R1 R2 f -> Inj R2 R3 g).

Instance compose_inj : Inj R1 R3 (compose g f).

About compose_inj.
Print HintDb typeclass_instances.
End foo.
```

If we take back the instance for the composition of injective functions in the code above, we see that all its arguments are declared as context variables. Therefore, inside the section `foo`, `compose_inj` has priority 0 because its definition does not have any quantified variables. After the section end, all the variables of the context are universally quantified. This makes the priority of `compose_inj` to become 3 since the implication `Inj R1 R2 f -> Inj R2 R3 g` in the context is made by the two constraints `Inj R1 R2 f` and `Inj R2 R3 g`. The third variable considered in computing the priority of `compose_inj` is the relation `R2`. In `Elpi`, after the end of a section we do the same: instances of a section are added *locally* and are re-added after the end of the section.

Note: In our solver the lowest priority we can give to instances is 1000. But, since the power user is free to interact at any time with `Elpi`'s database, he can manually add new hooks with the wanted priority at the wanted position.

4 Tactics and commands of our `Elpi`'s type-class solver

Until now, we have explained how instances are compiled in `Elpi`, but we have to give some details to show the way a goal is redirected from `Coq`'s solver to `Elpi`'s, how instances can be compiled automatically with their type-class signature and how they are added to the `tc.db` database. In the following sub-sections, we want to illustrate the API we have built in `Elpi` to accomplish these tasks.

4.1 The `TC_solver` tactic

In `Elpi`, it is possible to declare tactics runnable in `Coq`. A tactic can be introduced with the following syntax:

```
Elpi Tactic Name lp:{{
  % Here elpi code
}}
```

This creates a tactic named `Name` that, to be effective, needs the predicate `msolve` and `solve` to be implemented. The former is a predicate called to solve a list of `Coq`'s goals. On that list we can do some pretreatment like reordering the list of sub-goals, as we will explain in [Section 4.4.1](#). Then, after this pretreatment, each goal can be solved through the predicate `solve`. `solve` receives a goal and provides in the output a list of new sub-goals to be solved.

A goal is a tuple as shown below

```
(Ctx, RawSol, GoalT, Sol, Args)
```

The tuple is made by `Ctx`, that is the context in which the current goal lives, the `RawSol`, that is the raw solution used to elaborate the solution against `Ty`, `GoalT`, that is the type of the current goal, `Sol`, that is the solution of the current goal, `Args`, that is the list of arguments that can be passed to an `Elpi`'s

tactic to tweak its behavior. In our case, both `RawSol` and `Sol` are flexible variables and should be fixed via the instance search.

In our solver, `solve` has two implementations, the first aims to recursively treat goals on the form `Hyp -> Goal` or `forall x, Goal`. In both cases, in both `Hyp` and `x` are introduced in the context. In the base case, of `solve`, the type of the goal is the application of a type class to its arguments from which an `Elpi` query is built to start the type-class resolution. However, just before starting the search, we abstract in `Elpi` the variables of both the context and the current section if they represent an implementation of a type class. Finally, when a solution is found, it is refined with `Coq`'s initial goal and the new sub-goals, if any, will be returned to `Coq`.

For example, in the following goal,

```
Context (T : Type) (HT : Add T).
Goal Add bool -> Add (nat * (bool * T)).
```

we introduce the hypothesis `H : Add bool` in the proof context. Then to prove `Add (nat * (bool * T))` we compile `HT` from the section variables and `H` from the context. Thanks to these two hypotheses, a solution can be found for the goal `Add (nat * (bool * T))`:

```
addProd nat (bool * T) addNat (addProd bool T H HT)
```

Override of Coq type-class solver In order to use our solver instead of `Coq`'s one, we use the `Coq-elpi` feature, allowing us to capture `Coq`'s queries and redirect them, if needed, to `Elpi`. For our purpose, we intercept queries linked to type-class resolution and verify if all the sub-goals we receive are to be solved in `Elpi`. In fact, our solver coexists with `Coq`'s one. Through the command

```
Elpi Override TC TC_solver [All | None | ClassName+]
```

the user can either override all type-class problems, in this case any goal will be redirected to `Elpi`, none of them and therefore `Coq` will be charged to solve any goal, or we can override only some type classes. This means that when we receive the goals, we verify that all of these goal contain type classes that are overridden in `Elpi`. If the answer is positive, `Elpi`'s solver will try to solve these goals; otherwise, the task will be attributed to `Coq`. It is interesting to note that the `Override` command can be called and as often as the user wishes. This means that we can delegate `Elpi` to solve type-class goals in one part of the code, go back to `Coq` in another, and so on. Even if not yet tested, it should be possible to write different solvers in `Elpi` and let them co-exist with ours and `Coq`'s one... in this way, we could have, for example, specialized solvers for particular type-class goals.

4.2 Commands to add type-class signature in Elpi

Together with tactics, the user can also write `Elpi`'s commands. A command can be introduced via the syntax:

```
Elpi Command Name lp:{{
  % Here elpi code
}}.
```

This creates a command named `Name` that should have at least one implementation of the `main` predicate to be used as its entry point.

Before inserting instances in the database, we must add in `tc.db` the signature of their corresponding type classes so that instances can be accumulated. To do so, since in the actual `Coq`'s version, `v8.18`, we have not yet a hook to immediately know in `Elpi` when a type class has been declared, we have created two commands to:

- add all type classes at once, which takes all the type classes defined in `Coq` and add their signature in `Elpi`'s database (command: `Elpi AddAllClasses`);
- add a list of type classes passed as arguments (command: `Elpi AddClasses ClassName+`).

At any time, before adding the signature of a type class, `Elpi` verifies if it is already present to avoid duplicates.

4.3 Commands to add instances in Elpi

Concerning instances, we have created really similar commands to those dedicated for the insertion of type-class signatures, since we can:

- add all instances at once, which takes all the instances defined in Coq and compile them in Elpi's database (command: `Elpi AddAllInstances`);
- add a list of instances passed as arguments (command: `Elpi AddInstances InstanceName+`);
- add a list of type classes, which takes all the instances of the passed type classes and add them in `tc.db` (command: `Elpi AddInstance ClassName+`);
- the combination of the two previous commands (command: `Elpi AddInstances [InstanceName | ClassName]+`).

As for type-class signature compilation, Elpi verifies that no duplicates are added when inserting instances.

In a future version of Coq, we should have the hook to know when type classes and instances are created and be able to insert them automatically once defined.

4.4 New features of Elpi's solver wrt Coq's one

As pointed out in the previous sections, in Elpi we are completely free to tweak the type-class solver as we want. While talking with our power users of Coq, we have noticed that we could solve some of their issues in our solver. This confirms once more the flexibility of using Elpi in Coq and in the following sub-section we want to show the new features added in our solver to propose a solution to the issues we have been addressed.

4.4.1 Goals reordering

Solving a type-class goal often requires recursive searches to satisfy its constraints. The order in which these constraints are executed is influenced by the order in which they have been declared. For example, the `addProd` instance of [Section 1.3](#) generates two sub-goals, the first aims to prove `i1` and the second `i2`. This means that the user should pay attention to how he defines the constraints of an instance, since this choice can seriously impact the search performance. Most importantly, this order can define if the declared instance is meaningful with respect to its modes. In particular, for any argument in the input mode of a constraint, there should be a way in the instance to determine it. Otherwise, the instance is meaningless. Our prototype helps the developer by verifying this property during instance compilation and possibly reordering the constraints. If no valid order is possible, an error is thrown.

Listing 4.4.1: Reordering of goals

```

Class A (T : Type).
Class B (T : Type).
Class C (T1 : Type) (T2 : Type).
Hint Mode A + : typeclass_instances.
Hint Mode B + : typeclass_instances.
Hint Mode C + - : typeclass_instances.
Instance toSortInst (T1 T2 : Type) `(B T1, C T2 T1) : A T2.

```

Let's take the instance `toSortInst1`, defined in [Listing 4.4.1](#). Its implementation has two premises: there should exist an instance for B applied to the type T1 and an instance for C applied to T1 and T2. If we look at the mode of the type class B, we see that its argument should be in input mode.

This means that the goal

```

Context (HC : C nat nat).
Goal forall (T : Type), B nat -> A nat.

```

even if it has the solution `toSortInst1 nat nat H HC`, the automatic type-class search will not find it. This is due to the order in which the premises of `toSortInst1` are defined by the user. That's because Coq tries at first to solve the sub-goal `B T1`, but, since T1 is flexible, the mode mismatch will cause the search to reject this instance and therefore the initial goal will fail. On the other hand, if we look closer to `toSortInst1`, we could remark that solving $G' = C\ T2\ T1$ at first, will fix T1 after finding an implementation for G' . This implementation will fix T1 and therefore `B T1` could be solved. The Elpi compiler detects this invalid constraint order and rearranges them at compile time.

The procedure we adopt to sort goals is a slightly modified topological sort where an instance is represented with a *dependency graph*. The nodes are made from the constraints and the goal of the current instance. Every node N represents a type class TC and the list of its arguments $A = [a_1, \dots, a_n]$,

n is the arity of TC . Each N node has n_1 exiting arcs, one for each argument a_o in output mode, and n_2 arcs entering arcs, one for each argument a_i in input mode (note $n = n_1 + n_2$). The node representing the type class implemented by the current instance (we will call it G for simplicity) is a special node where all arguments (even those in input mode) are represented with exiting arcs. This is because in the goal, the arguments in input mode should not be flexible.

The second step in the construction of the graph is to connect each exiting arc labeled with l to each entering arcs labeled with l .

From that graph, we compute a slightly modified topological sort on the nodes of the graph. We start by putting all the nodes with no entering arcs in the stack of nodes to treat. Any time we treat a node N we add it to the resulting list R , we remove it from the graph, and if L is the list of the labels of the exiting arcs of N , for any label $l_i \in L$, we remove all the arcs labeled l_i from the graph. This means that it is possible to fix l_i after the resolution of the goal associated with N . We repeat this procedure until there exists at least one node to treat and at the end we remove G from R so that R is a possible valid order of the constraints.

Just as a small remark, we should keep attention on the exiting arcs a_o of G in output mode, we should check if they are needed by other nodes as input and, if so, a_o should be marked in the clause as a not flexible variable. A more complex example of goal ordering is depicted in [Appendix A.3](#).

Particular cases of invalid instances An interesting property of the topological sort is that it is not always possible to have a valid ordering. For instance, we can have graphs with cyclic dependencies. An example of this is depicted in [Listing A.3.3](#).

Another kind of instance that Elpi's solver does not allow to compile are those instances that have at least a node with an argument in input mode and no other premise allowing to fix that argument. An example of this is given in [Listing A.3.4](#).

In both cases, the Elpi's type-class solver will raise a compilation error.

4.4.2 Deterministic search

During the search of instances, a goal may have multiple solutions.

Listing 4.4.2: Type class with multiple solutions

```
Class c (n : nat).
Instance c_2 : c 2 | 1.
Instance c_1 : c 1 | 10.
```

Let's take the code above as an example. We note that the goal (`c _`) has both `c_2` and `c_1` as valid solutions. It is frequent, however, that a developer may want a type class to be *deterministic*. This means that when an instance search is performed, if the solver finds a goal using a deterministic type class and if a solution is found for that goal, any backtracking is forbidden. This behavior is easily reproducible in logic programming: the `cut` operator does exactly this.

In our prototype, if the type class is marked as deterministic with the pragma `det` (like this: `#[det] Class c (n : nat)`) we modify the way instances are compiled. In particular, when we build the list of premises of an instance, if a premise is a sub-goal referred to a deterministic type class, it is wrapped inside the predicate `do-once`. The implementation of `do-once` is shown below.

```
pred do-once i:term.
do-once P :- P, !.
```

Let's take a more complete example to show how the deterministic search works

```
Class C (n : nat).
Instance c_2 : C 2 | 1.
Instance c_1 : C 1 | 10.

Class D (n : nat).
Instance d_1 : D 1.

Class E (n : nat).
Instance e_n {n} : C n -> D n -> E n.
```

and let the goal to be solved (E ?n). In this case, the solution E 1: e_n c_1 d_1 can be found via the following trace:

Listing 4.4.3: Trace of instance search for the previous code snippet

```

1 Goal E ?n -> apply e_n -> sub-goals = C ?n, D ?n
2   sub-goal1 C ?n -> apply c_2 -> ?n is now 2
3   sub-goal2 D 2 -> no solution, so backtrack
4
5   sub-goal1 C ?n -> apply c_1 -> ?n is now 1
6   sub-goal2 D 1 -> apply d_1
7 Solution = e_n 1 c_1 d_1

```

We see that at line 2, we choose to fix ?n to 2, which will cause a fail at line 3. This means that the choice of applying c_2 is not a good. The algorithm backtracks to C ?n to find a new possible solution. c_1 is the new candidate to be applied, and this will allow to find a solution for d_1.

Let's imagine now, that C is declared as a deterministic type class. This means that its definition should be replaced with `#[det] Class C (n : nat)`.

As said before, if a type class is deterministic and it is used as the premise of an instance, we wrap it inside the `do-once` predicate. That means that the instance e_n will be compiled in the following way:

```

tc-E N {{e_n lp:N lp:SolC lp:SolD}} :-
  do-once (tc-C N SolC),
  tc-D N SolD.

```

If we try to solve again the goal (E ?n), we will obtain no solution. This is because when we arrive at line 3 of Listing 4.4.3, we try to backtrack to the goal C ?n, but due to the cut inside `do-once`, we will not be able to find c_1. We try to backtrack again to the original goal E ?n, but since we have no other possible solution, we fail to solve it.

Note: The pragma `det` is not present in Coq and the deterministic search for a particular type class is proper to our prototype

4.5 Elpi's vs Coq's term unification

In the previous sections, we have rather quickly spoken about the unification done in Elpi between two terms, but it is important to spend a few words on it, since we want to compare it with the unification done in Coq.

Elpi's unification of terms Let X be a flexible variable, τ, f rigid terms, α, β either flexible or rigid variables, $V(t)$, the function returning the variables inside t and $\mathcal{G}[X \leftarrow \tau]$ the replacement of all the variables X in \mathcal{G} by t , the first-order unification algorithm in Elpi is summed up in Algorithm 1.

Together with first order quantification, where only variables are quantified, Elpi does an higher order unification where any kind of term can be quantified, [3].

Coq's unification of terms The Elpi's unification algorithm is faster than Coq's since it contains only a subset of the unification rules used by Coq. [12, 16]

Pros and cons of both unification styles The Elpi unification algorithm is faster than Coq one since it contains only a subset of the unification rules used by Coq.

However, if we take the `Add` type class and implement an instance to make addition between booleans, as defined in Listing 4.5.1, in Elpi we are able to solve the goal `Add bool'`, but not `Add bool`.

Listing 4.5.1: Instance with type alias

```

Definition bool' := bool.
Instance boolAdd : Add bool' := {
  B1 + B2 := match B1, B2 with

```

Algorithm 1: High-level representation of Elpi's first-order unification process

```

if  $\mathcal{G} = \{t = t\} \cup \mathcal{G}'$  then
  |  $\mathcal{G} \leftarrow \mathcal{G}'$ 
else if  $\mathcal{G} = \{t = x\} \cup \mathcal{G}'$  then
  |  $\mathcal{G} \leftarrow \mathcal{G}' \cup \{x = t\}$ 
else if  $\mathcal{G} = \{x = s\} \cup \{x = t\} \cup \mathcal{G}'$  then
  |  $\mathcal{G} = \mathcal{G}' \cup \{x = s\} \cup \{s = t\}$ 
else if  $\mathcal{G} = \{f(\alpha_1, \dots, \alpha_n)\} = \{f(\beta_1, \dots, \beta_n)\} \cup \mathcal{G}'$  then
  |  $\mathcal{G} = \mathcal{G}' \cup \{\alpha_1 = \beta_1\} \cup \dots \cup \{\alpha_n = \beta_n\}$ 
else if  $x \notin V(t) \wedge x \in V(\mathcal{G}) \wedge \{x = t\} \cup \mathcal{G}'$  then
  |  $\mathcal{G} = \mathcal{G}' [x \leftarrow t] \cup \{x = t\}$ 
else if  $\mathcal{G} = \{f(\alpha_1, \dots, \alpha_n)\} = \{g(\beta_1, \dots, \beta_n)\} \cup \mathcal{G}'$  then
  | Error: Cannot unify  $\rightarrow f \neq g$ 
else if  $x \in V(t) \wedge \mathcal{G} = \{x = t\} \cup \mathcal{G}'$  then
  | Error: Cannot unify  $\rightarrow$  cycle
end

```

```

  false, false => false
  | _, _ => true
  end;
  add_is_comm := ...
}

```

This is because Elpi is not able to unify `bool` with `bool'`. To overcome this problem, we have created a new predicate `pred alias i:term, o:term`, so that the user can add in the database (via the command `AddAlias`) the redefinition of types. In our previous example, we can add `alias {{bool}} {{bool'}}` so that when the goal in input contains the type `bool` it is replaced programmatically with `bool'`.

Another kind of unification problem that has caused lots of problems is linked to the η -reduction since, often, the goal G is on the form `fun x => F x` which is η -equivalent to F , Elpi is not able to unify G with F . The η -reduction predicate to make it possible to unify these two terms is not easy to implement efficiently: there are a lot of recursive cases asking to traverse very big terms. Elpi is not well suited to work with this kind of recursive function, but, thanks to `Coq-elpi`, a faster implementation can be built in OCaml.

Higher order unification A final unification problem we have treated in Elpi is the so-called pattern fragment unification [3], where a term `(F a_1 ... a_n)` represents a flexible function applied to a list of all-distinct names, atoms, variables or constants. Elpi is able to manage this kind of unification, but the terms received by Coq where F is flexible, have not a flexible head.

Listing 4.5.2: Unification with pattern-fragment

```

Class Unif (A: Type).
Global Instance unifInst (a : Type) F: Unif (F a). Qed.

```

The type of the instance `unifInst` is

$$\text{unifInst} : \text{forall } (a : \text{Type}) (F : \text{Type} \rightarrow \text{Type}), \text{Unif } (F \ a)$$

where F is a function of type `Type -> Type`, and if we translate `unifInst` to its Elpi representation using the `compile` function, the following clause is created:

```

tc-Unif {{lp:F lp:A}} {{unifInst lp:A lp:F}}.

```

it is important to note that the first argument of that clause is translated in Elpi in the term \mathcal{T} `app [F, A]` since Elpi gets the Coq's application of F with A as argument. With this implementation, we can solve any goal where F is a function of arity one. For instance, `Unif (list nat)` is solvable in Elpi, since `list nat` is translated in `app [{{list}}, {{nat}}]` which is unifiable to \mathcal{T} : F is unified with `list` and A with `nat`. On the other hand, if the function F is a function of arity n , any term on the form $Fx_1 \dots x_{n-1}$, will not unify with \mathcal{T} . Just to enforce our claim, the goal `Unif (prod nat nat)` has the first argument which is translated in Elpi in `app [{{prod}}, {{nat}}, {{nat}}]` which cannot

be unified with `app [F, A]`: the two functions have the same rigid head but their arities are different. This failure is not what we want, since `F A` should unify with `prod nat nat` by setting `F` to `prod nat` and `A` to `nat`.

The solution we propose is to transform a Coq's function application into an Elpi's function application and solve the unification by taking advantage of the Elpi higher-order unification algorithm. Once unified, the resulting term is again turned in Coq's term representation.

```
tc-Unif (app L) {{unifInst lp:A lp:F}} :-
  std.appendR H [A] L,
  if (H = [X]) (F = X) (F = app H).
```

in the representation above we are saying:

- The first argument of the type class `Unif` is the Coq's application of a list of arguments `L` (the head of `L` is the function `F`, the tail is the arguments applied to `F`)
- In Elpi the list `L` is destructed in the two sub-lists `H` and `[A]` such that their concatenation equals `L`
- If `H` has length 1, then `F` is an unary function and we set `F` to the first element of `H`. Otherwise, the function `F` represent a partially applied function, we transform the list `H` in the Coq's term `app H`
- The returned solution is `unifInst` applied to the Elpi variable `A`, which is, in this case, the last element of `L`, and `F`.

The compilation of clauses with pattern fragment unification in Elpi is automatically done any time the type of an instance contains a function the head of which is represented by a \forall -quantified variable. As examples, let's focus on the resolution of the previous two goals with this newly created rule:

1. $G = \text{Unif (list nat)}$: the term `list nat` is translated into `app [list], [nat]` in Elpi. After unification, `L = [list], [nat]`. The second line of the code, will unify `H` to `[list]` and `A` to `[nat]`. Since `H` has length 1, `F` is unified to `[list]`. The solution to this goal is `unifInst nat list`
2. $G = \text{Unif (prod nat nat)}$: the term `prod nat nat` is translated into `app [prod], [nat], [nat]` in Elpi. After unification, `L = [prod], [nat], [nat]`. The second line of the code, will unify `H` to `[prod], [nat]` and `A` to `[nat]`. Since `H` has not length 1, `F` is unified to `app [prod], [nat]`. The solution to this goal is `unifInst nat (prod nat)`

4.6 User-defined rules

As explained before, a big feature of Coq-elpi is the possibility to easily interact with and modify the behavior of the solver by adding (and potentially removing) clauses to the database. This possibility is similar but way more powerful to the `Hint Extern` command in Coq allowing to introduce new rules in the Coq's database of instances. In Elpi the rules we can add are meant especially to solve particular complex goals by guiding the search in a more efficient way and sometimes to obtain an improvement in time performance.

A problem we can focus on may be the resolution of goals dealing with the `Inj` type class (see Section 3.1.3). In particular, if we have very large terms of compositions of functions, we can fall into a situation where the tree representing this composition of functions contains a lot of symmetries.

Let f an injective function, \circ the composition function, and `compose_inj` the instance saying that if $f_1 : B \rightarrow C$ and $f_2 : A \rightarrow B$ are injective then $f_1 \circ f_2$ is also injective. We can take as an example the term

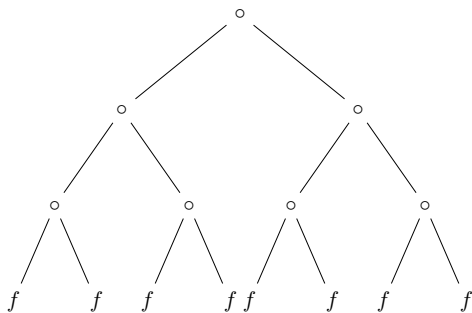
$$((f \circ f) \circ (f \circ f)) \circ ((f \circ f) \circ (f \circ f))$$

which can be associated to the tree in Figure 4.

In this case we can remark that the tree has a height h of 3 and that each inner node has equal children, which therefore, have the same solution. In Elpi we can improve the performance by doing the search only once for each sub-tree and have an exponential reduction of recursive calls in terms of the height of the tree. The Elpi's rule for this task can be:

Listing 4.6.1: Ad-hoc rule for `compose_inj`

```
1 :after "0"
2 tc-Inj A A RA RA {{compose lp:LF lp:LF}} Sol :- !,
```

Figure 4: Tree with height $h = 3$

```

3  tc-Inj A A RA RA LF SolLF,
4  Sol = {{compose_inj lp:RA lp:RA lp:RA lp:LF lp:LF lp:SolLF lp:SolLF}}.

```

This rule is applied if the first two arguments, the second two and the left and right functions applied to `compose` are the same. At line 3, we check if the function `LF` is injective and if so, we can return the solution `compose_inj` whose two last arguments for the left and right children are the same and stored in the variable `SolLF`.

A further improvement we can apply to this function may be to reduce the size of the proof term. This can be done in `Coq` via memory sharing so that the kernel receives a smaller term to typecheck.

Listing 4.6.2: Ad rule for `compose_inj` with memory sharing

```

1  :after "0"
2  tc-Inj A A RA RA {{@compose lp:A lp:A lp:A lp:LC lp:LC}} Sol :- !,
3  tc-Inj A A RA RA LC Sol1,
4  Sol = {{
5    let sol := lp:Sol1 in
6    let lc := lp:LC in
7    @compose_inj lp:A lp:A lp:A lp:RA lp:RA lp:RA lc lc sol sol
8  }}.

```

In the rule above, we are reducing exponentially the size of the resulting proof term since the solution `sol` and the type of the left child `lc` are typechecked only once at each inner node of the tree.

This is just a proof of concept but shows the power of custom rules.

5 Validation of results and synthetic benchmark

The aim of this project is the development of a type-class solver for `Coq`, which is able to reproduce and improve the search engine of `Coq`. In order to identify out some of the software bug of our code, we took the library `stdpp` [20]. `stdpp` is a library containing definitions and lemmas for data structures such as lists, finite maps, finite sets and finite multisets. In `stdpp`, all these data structures are encoded via a very massive usage of type classes/instances. In this way, a lot of goals can be solved automatically via the type-class solver of `Coq`. We count 113 type classes and 1,083 instances. This library is therefore particularly adapted to validate our results.

We have found some bugs in our implementation thanks to `stdpp`, such as the unification problems explained in Section 4.5, the importance of ordering of instance in `Elpi`'s database and so on. We also had the opportunity to have several exchanges with Robbert Krebber, the main developer and maintainer of `stdpp`, to get concrete issues of `Coq`'s type-class solver, some of which have been listed in Section 4.4.

`Elpi`'s type-class solver solves 10,466 of type-class goals which is about the 73.54% of the total number of goals in `stdpp`, with 2,390,004 recursive calls. The remaining goals not solved by `Elpi`'s solvers are mostly related to unification problems.

These results are very promising, since we can see how in a short time this project has reached a good rate of coverage of `stdpp`. These results come together with a synthetic benchmark which confirms, once again, the competitiveness of our solver.

The synthetic benchmark considered is based on goal associated to the `Inj` type class. The goals are

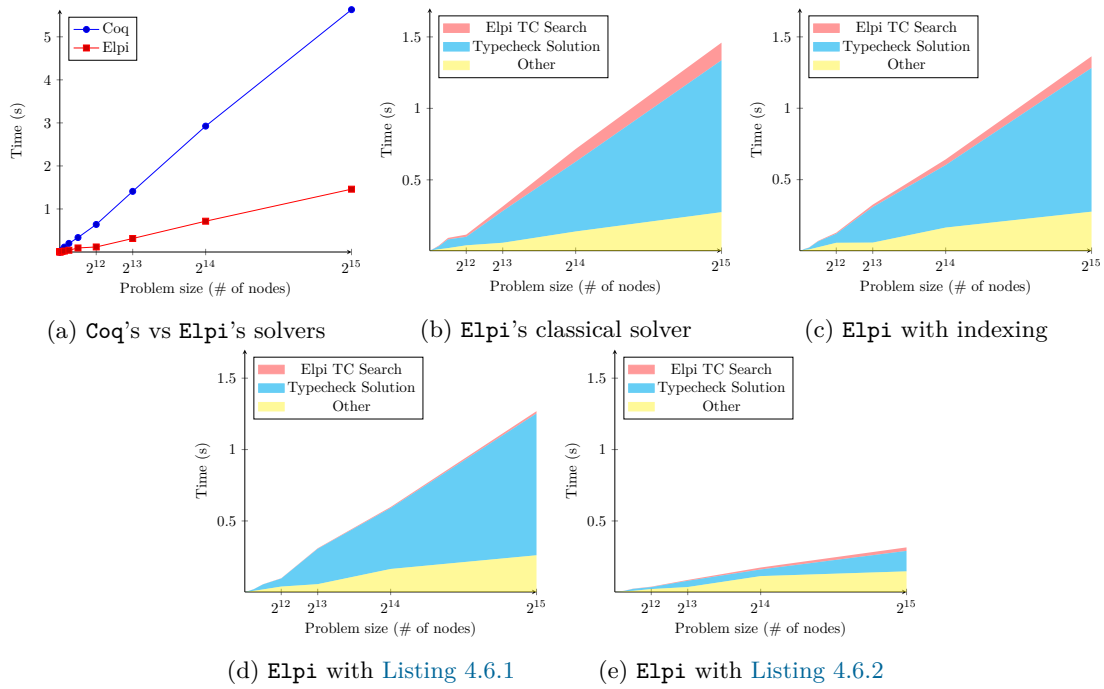


Figure 5: Synthetic of composition of functions

complete binary tree of composition of functions. An example of such a tree is in Figure 4. We start with the goal

$$\text{Inj } (f \circ f)$$

which is a tree of height 1. The next goal will be

$$\text{Inj } ((f \circ f) \circ (f \circ f))$$

and so on. We have compared the performance of the two type-class solvers for trees of height going from 1 to 14. For each tree, if its height is n , then there are $\sum_1^n 2^N = 2^{n+1} - 1$ sub-goals to solve: the root generates two sub-goals A and B ; A and B generate both two sub-goals and so on. The tree has 2^n leaves.

We have tested this problem with four different variations of Elpi's solver. In particular:

- in Figure 5a, we compare Elpi's and Coq's performance
- in Figure 5b, we use the classical Elpi's solver, with no handwritten rules
- in Figure 5c, we have added the indexing on the 5th argument of Inj
- in Figure 5d, we have added the ad-hoc rule shown in Listing 4.6.1
- in Figure 5e, we have added the ad-hoc rule shown in Listing 4.6.2

In the plots, the x -axis represents the number of sub-goals to be solved on a logarithmic scale⁷, while the y -axis represents the time taken to solve a goal in seconds. The Elpi's time has been divided into 3 components: the time taken to do the instance search (the red area), the time taken to typecheck the solution (the blue area). The yellow area, called *other*, represents the time spent by Coq-elpi that is not attributable to our type-class solver. This time depends on several causes, but the most important factor is the translation of the Coq's terms into their Elpi's representation (see Section 2.2). This translation is done twice, the first time is to receive the Coq's term as input, and the second time to transmit the result of Elpi to Coq.

Nevertheless, from Figure 5a, we see that the classical Elpi's solver is about 3.85 times faster than Coq for trees of height 14, while on small trees, Elpi is already faster for trees of height 5.

At first, we have tried to apply the indexing algorithm of Elpi. This improvement, should only affect the time of the red area, and, actually, the red area is about 1.5 smaller than that of the classical solver. We should keep in mind that in our benchmark, there are 15 instances of Inj, which should not cause many conflicts during the term hashing.

The rule in Listing 4.6.1 as the indexing is intended to reduce the search time and, therefore, so only the red area is affected. We can see this by comparing the size of the red regions in Figure 5b and Figure 5d. In fact, the red area is barely negligible because instead of exploring the whole tree,

⁷Recall that 2^{15} is the number of sub-goals to be solved in a tree of height 14

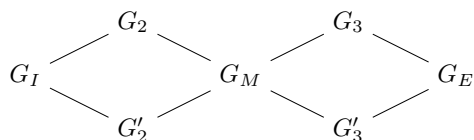


Figure 6: Diamond problem example

we break all its symmetries and therefore the exponential search in terms of the height h of the tree is halved at each node. This means that the number of recursive calls is linear in term of h : in this way the complexity class of the search changes from exponential to linear.

The last and more interesting result is shown in Figure 5e. In this plot we use the memory sharing taken from Listing 4.6.2. We see that all the regions are smaller compared to the plots in Figure 5. The red area represent the same efficiency of the recursive search as the one used in Figure 5d. Furthermore, the solution returned to Coq is much smaller, since the proofs of Inj for the left and right children of each node are typechecked once for each level of the tree instead of twice.

6 Future work

Some interesting improvements to the code have been left behind due to a lack of time, but we can mention some of them in the following sub-sections, since we believe they can give higher performances and usability for our solver and more in general to Elpi.

6.1 Memoisation and tabled search

The DFS search algorithm used to explore the clauses in logic programming languages allows one to entirely explore a tree of solutions to solve a goal. Because of its implementation, the DFS procedure asks the user to correctly place the instances in the database to tweak the search in a more efficient way. However, especially in libraries specialized in the representation of mathematical structures, there soon exists a big hierarchy of type-class inheritance. This hierarchy, even if the instances are ordered as best as possible, may contain a lot of different paths that allow to solve a goal. This is the case for the so-called diamond problem.

In Figure 6, there are four different paths from the initial G_I to the sink G_E , where we try to solve twice the same goal G_M . We can imagine having even more complex towers of diamonds leading to a sink with an exponential blow-up with respect to the height of the tower. In most of the cases this cost a loss of time.

Another problem the DFS algorithm may fall into, is the possibility to loop indefinitely on the same goal. We have sketched an example of this kind in Listing 1.5.1, where the solver keeps generating sub-goals on the form (Num _).

To solve both problems, some research has been done. The idea is to use a particular kind of memoisation, called tabled resolution[14]. In this way, during the search a memory is kept updated during all the exploration of the search tree. The memory is made of a *positive cache* (PC) and a *negative cache* (NC). The first contains the set of pairs $\Gamma_i \vdash G_i$ that have been solved and to each of them the solution is associated. The second contains the pairs $\Gamma_j \vdash G_j$ on which a solution has not yet been found.

Now, during the resolution of a goal $\Gamma \vdash G$, the algorithm, before exploring the database of instances, should look at the cache where three situations can happen:

- $\Gamma \vdash G \in \text{PC}$, then, since a solution \mathcal{S} has already been found for that goal, the search is not done and \mathcal{S} is returned back;
- $\Gamma \vdash G \in \text{NC}$, then this goal has already been found but not solution has not been found, therefore the resolution for G is stopped
- $\Gamma \vdash G \notin \text{PC} \cup \text{NC}$, then the classic DFS search is applied. If a solution \mathcal{S} is found on G , then $\Gamma \vdash G : \mathcal{S}$ is added to PC, otherwise, $\Gamma \vdash G$ is added to NC

If a goal belonging to NC contains a flexible variable, and during the resolution of another goal this flexible variable is fixed, then it is woken up since, now, it may be possible to solve G into PC. Thanks to tabling:

- the particular ad-hoc rules in [Listing 4.6.1](#) and [Listing 4.6.2](#) will no longer be necessary, since the cache would detect two equal sub-goals and give the second the same solution as the first without searching
- the cycle produced in [Listing 1.5.1](#) by the goal $(\text{Num } _)$ is stopped after the first recursive call since after that, $\vdash \text{Num}?n$ is in the NC
- in [Figure 6](#), if the resolution of G_I pass through the path

$$(\Gamma_I \vdash G_I, \Gamma_2 \vdash G_2, \Gamma_M \vdash G_M, \Gamma_3 \vdash G_3, \Gamma_E \vdash G_E)$$

if, in a future iteration, the algorithm tries to take the path $\Gamma_I \vdash G_I, \Gamma_{2'} \vdash G_{2'}, \Gamma_M \vdash G_M$, no more search will be done since the $\Gamma_M \vdash G_M$ has already found the same couple in the previous search

This search via tabling will be built in the Elpi runtime and, therefore, will be available not only for the resolution of type-class goals but also to any other Elpi program, since the principle of memoization can be easily generalized in logic programs.

6.2 Indexing algorithm on pattern

As explained in [Section 3.1.3](#), Elpi comes with an indexing algorithm useful to increase performance during the search. However, especially while working with Coq's terms, we remark that a lot of bits used for the encoding are spent on pieces of information that are not interesting. The Coq's function `compose` is translated into the Elpi's term:

```
app [global (const «compose»), X0, X1, X2]
```

This encoding is valid for all the application of Coq's functions. We can see that in those cases, the hash function would encode the sub-terms `app`, `global` and `const`. Although it would be more interesting to modify the indexing procedure so that, instead of encoding every sub-term of a term T , the indexing is done over a pattern [4] indicated by the user that should match the term received in input.

For instance, the user could be allowed to use a syntax of the form:

```
:index ( _ _ _ _ (3 : app [global (const X) | _]) _ )
pred tc-Inj o:term, o:term, o:term, o:term, o:term, o:term.
```

saying that we encode the 5th argument of the `tc-Inj` predicate in a the following way: instead of encoding all sub-terms from position 0 to the given depth, we encode only the sub-terms marked with `X` with depth 3. The use of patterns needs discrimination trees, by which terms are represented in a tree of prefixes, as rapidly shown in [Figure 3](#).

6.3 Overlapping instances detection

This final sub-section of potential improvements to the Elpi type-class prototype is linked to instance overlapping[19]. We say that an instance I_1 overlaps an instance I_2 if to any goal on which we can apply I_2 we can always apply I_1 . This may happen when, for all arguments B_i at position i of the instance I_2 , the corresponding argument A_i of the instance I_1 , is such that $B_i \subseteq A_i$. For example, if we work with sets, we may want to create and first instance I_\emptyset for empty sets and a second I_S for general sets. In this case any time I_\emptyset can be used, so I_S . The goal of the overlapping instances detection should be such that the user is free to choose if a type class should allow or not this kind of overlapping (for example, `Haskell`, by default, does not allow this option).

Avoiding instance overlapping means that the user should be warned after instance compilation if this instance overlaps or is overlapped by other instances. The overlapping problem often happens while importing different files. For example, let's consider the following three files:

- `file1.v` containing the definition of the type class `Add` and the implementation of `addNat`
- `file2.v` containing the implementation of `addProd`
- `file3.v` containing the instance `addProd1` which is defined as the instance `addProd` in `file2.v`

We suppose that `file2.v` and `file3.v` both import `file1.v` (note that we do not have the instance for the addition over booleans). Now let's imagine `file4.v` to import the three files defined before and let $G = \text{Add } ((\text{bool} * \text{bool}) * (\text{bool} * \text{bool}))$ the goal to be solved in `file4.v`. G has, of course, no solution, but to have this certification by the type-class solver, we have to do the following search:

```
Goal G -> apply addProd -> sub-goals = (bool * bool), (bool * bool)
      sub-goall1 (bool * bool) -> apply addProd -> sub-goals = bool, bool
```

```

sub-goal1 bool -> no solution, so backtrack
sub-goal1 (bool * bool) -> apply addProd' -> sub-goals = bool, bool
sub-goal1 bool -> no solution, so backtrack

Goal G -> apply addProd' -> sub-goals = (bool * bool), (bool * bool)
sub-goal1 (bool * bool) -> apply addProd -> sub-goals = bool, bool
sub-goal1 bool -> no solution, so backtrack
sub-goal1 (bool * bool) -> apply addProd' -> sub-goals = bool, bool
sub-goal1 bool -> no solution, so backtrack

```

In the previous example, we see that a lot of useless searching is done while trying to solve G , since at any step of the search, when the sub-goal has the shape $(_ * _)$, the number of possible instances to be applied is 2: `addProd` and `addProd'`. This doubles the number of possible paths at any node of the exploration tree, producing an exponential number of paths to be explored. Worst scenarios are possible, since all the instances `addProd` ^{n} , ..., `addProd'` ^{n} could be imported in the same file, and in this case the complexity of the search is n^h where h is the height of the tree and n the number of overlapping instances imported.

Note that detecting overlapping instances is a concept tightly linked to the deterministic search explained in [Section 4.4.2](#). In deterministic search, backtracks are forbidden for deterministic type classes during runtime, while with overlapping instance detection, backtracks are removed during compile time since a goal should unify to at most one clause. A starting point for this overlapping detection is provided in [8], where the detection of the determinacy of predicates is done by verifying mutual exclusion between clauses in the database and where modes and types are used to make this kind of analysis.

7 Conclusion

Type-class resolution is a powerful part of Coq's elaborator, and it is increasingly used by Coq's developers, who use it to automate proof search. The instances are sought by unifying their type and the goal of the current proof, but multiple factors, such as hint modes, the order in which instances are declared, the diamond problem, and so on, heavily impact the performance of the search. The actual Coq's type-class solver has raised some issues in the research and industrial domains, ranging from performance to a lack of control over the database. The implementation of our prototype wants to ease those issues by answering power users' needs. The results we have obtained are rather promising since we can already solve some difficult problems, and in some cases, our resolution of goals is faster than Coq's. Our expectation is to continue this project in a PhD to stabilize our prototype and, step by step, give a solution to the open problems we have mentioned previously. As said, the type-class solver constitutes just a part of the entire elaboration process, and we think it's possible to use Elpi and cover other components of elaboration such as canonical structure resolution, coercions and type inference. We hope this project will be welcomed by the scientific community and have some impact on the Coq infrastructure. Our final goal is for our solver to be integrated into Coq so that it can coexist with Coq's solver and potentially, one day, replace it.

Acknowledgements I'm very grateful to have been chosen to work on this project. Furthermore, I would really like to thank my advisor *Enrico Tassi* for the time he dedicated to me and for all the ideas, propositions, and cues he addressed me in the study and development of this prototype. I would also like to thank the team *stamp* where I did my internship. This internship allowed me to come into contact with the research world. I had the possibility to present our project to the team *Gallinette* at *Nantes Université* and to the Coq's workshop at the *ITP23* conference. Last, but not least, I would like to thank *Robbert Krebber* from *Radboud University Nijmegen* for suggesting a lot of features we added to our solver together with *Jarl G. Taxerås Flaten* and *Matthieu Sozeau* for the time he dedicated to illustrate me some concrete problems linked to type-class resolution.

8 References

- [1] Gopalan Nadathur and Dale Miller. "An Overview of Lambda-Prolog". In: June 1988, pp. 810–827.

-
- [2] P. Wadler and S. Blott. “How to Make Ad-Hoc Polymorphism Less Ad Hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76. ISBN: 0897912942. DOI: [10.1145/75277.75283](https://doi.org/10.1145/75277.75283). URL: <https://doi.org/10.1145/75277.75283>.
- [3] Dale Miller. “A logic programming language with lambda-abstraction, function variables, and simple unification”. In: *Extensions of Logic Programming*. Ed. by Peter Schroeder-Heister. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 253–281. ISBN: 978-3-540-46879-0.
- [4] William McCune. “Experiments with discrimination-tree indexing and path indexing for term retrieval”. In: *Journal of automated reasoning* 9.2 (1992), pp. 147–167.
- [5] Cordelia Hall et al. “Type classes in Haskell”. In: *Programming Languages and Systems — ESOP ’94*. Ed. by Donald Sannella. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 241–256. ISBN: 978-3-540-48376-2.
- [6] Jason Holdsworth. “The Nature of Breadth-First Search”. In: (Feb. 1999).
- [7] Matthieu Sozeau and Nicolas Oury. “First-Class Type Classes”. In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 278–293. ISBN: 978-3-540-71067-7.
- [8] Pedro López-García, Francisco Bueno, and Manuel V. Hermenegildo. “Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Analyses”. In: *New Generation Comput.* 28 (Apr. 2010), pp. 177–206. DOI: [10.1007/s00354-008-0085-1](https://doi.org/10.1007/s00354-008-0085-1).
- [9] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. DOI: [10.1017/CB09781139021326](https://doi.org/10.1017/CB09781139021326).
- [10] Assia Mahboubi and Enrico Tassi. “Canonical Structures for the Working Coq User”. In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 19–34. ISBN: 978-3-642-39634-2.
- [11] Leonardo de Moura et al. *Elaboration in Dependent Type Theory*. 2015. arXiv: [1505.04324](https://arxiv.org/abs/1505.04324) [cs.LG].
- [12] Beta Ziliani and Matthieu Sozeau. “A unification algorithm for Coq featuring universe polymorphism and overloading”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 179–191. DOI: [10.1145/2784731.2784751](https://doi.org/10.1145/2784731.2784751). URL: <https://doi.org/10.1145/2784731.2784751>.
- [13] Thom Fruehwirth. *Constraint Handling Rules - What Else?* 2017. arXiv: [1701.02668](https://arxiv.org/abs/1701.02668) [cs.PL].
- [14] Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. “Tabled Typeclass Resolution”. In: *CoRR* abs/2001.04301 (2020). arXiv: [2001.04301](https://arxiv.org/abs/2001.04301). URL: <https://arxiv.org/abs/2001.04301>.
- [15] *Coq builtins*. URL: <https://github.com/LPCIC/coq-elpi/blob/master/coq-builtin.elpi> (visited on 09/07/2023).
- [16] *CRM: Conversion rules*. URL: <https://coq.inria.fr/refman/language/core/conversion.html> (visited on 09/07/2023).
- [17] *CRM: Hint Modes*. URL: <https://coq.inria.fr/refman/proofs/automatic-tactics/auto.html#coq:cmd.Hint-Mode> (visited on 09/07/2023).
- [18] *Elpi builtins*. URL: <https://github.com/LPCIC/coq-elpi/blob/master/elpi-builtin.elpi> (visited on 09/07/2023).
- [19] *Haskell: Overlapping Instances*. URL: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/instances.html#overlapping-instances (visited on 09/07/2023).
- [20] *The stdpp library*. URL: <https://gitlab.mpi-sws.org/iris/stdpp> (visited on 09/07/2023).

A Appendix

A.1 Term with missing information vs. full term

In [Listing A.1.1](#), we are defining the `determinant` function for matrices using the mathematical notation and on the other hand, in [Listing A.1.2](#), we can see all the implicit pieces of information that have been added to the original term.

We can remark that the first term takes 2 lines of code, while the expanded version takes 50.

Listing A.1.1: Determinant: term with implicits

```
1 Definition determinant n (A : 'M_n) : R :=
2   Σ (σ : perm_of n) sgn(σ) * Π i, A i (σ i).
```

Listing A.1.2: Determinant: complete term

```
1 determinant =
2   fun (n : nat)
3     (A : matrix (GRing.SemiRing.sort (GRing_Ring__to__GRing_SemiRing R)) n n)
4     =>
5     @bigop.body
6       (GRing.Nmodule.sort (GRing_SemiRing__to__GRing_Nmodule
7         (GRing_Ring__to__GRing_SemiRing R)))
8       (Finite.sort
9         (perm_perm_type__canonical__fintype_Finite
10          (fintype_ordinal__canonical__fintype_Finite n)))
11       (@GRing.zero
12         (GRing_SemiRing__to__GRing_Nmodule (GRing_Ring__to__GRing_SemiRing R)))
13       (index_enum
14         (perm_perm_type__canonical__fintype_Finite
15          (fintype_ordinal__canonical__fintype_Finite n)))
16       (fun
17         s : @perm_of (fintype_ordinal__canonical__fintype_Finite n) (Phant
18           ↪ (ordinal n)) =>
19         @BigBody
20           (GRing.Nmodule.sort (GRing_SemiRing__to__GRing_Nmodule
21             (GRing_Ring__to__GRing_SemiRing R)))
22           (@perm_of (fintype_ordinal__canonical__fintype_Finite n)
23             (Phant (ordinal n))) s
24           (@GRing.add
25             (GRing_SemiRing__to__GRing_Nmodule (GRing_Ring__to__GRing_SemiRing
26               ↪ R)))
27           true
28           (@GRing.mul (GRing_Ring__to__GRing_SemiRing R)
29             (@GRing.exp (GRing_Ring__to__GRing_SemiRing R)
30               (@GRing.opp
31                 (join_GRing_Ring_between_GRing_SemiRing_and_GRing_Zmodule R)
32                 (GRing.one (GRing_Ring__to__GRing_SemiRing R)))
33                 (nat_of_bool
34                   (@odd_perm (fintype_ordinal__canonical__fintype_Finite n) s))))
35           (@bigop.body (GRing.SemiRing.sort (GRing_Ring__to__GRing_SemiRing
36             ↪ R))
37             (Finite.sort (fintype_ordinal__canonical__fintype_Finite n))
38             (GRing.one (GRing_Ring__to__GRing_SemiRing R))
39             (index_enum (fintype_ordinal__canonical__fintype_Finite n))
40             (fun i : Finite.sort (fintype_ordinal__canonical__fintype_Finite
41               ↪ n)
42               =>
```

```

39      @BigBody (GRing.SemiRing.sort (GRing_Ring__to__GRing_SemiRing
    ↪ R))
40      (Finite.sort (fintype_ordinal__canonical__fintype_Finite n))
    ↪ i
41      (@GRing.mul (GRing_Ring__to__GRing_SemiRing R)) true
42      (@fun_of_matrix
43      (GRing.SemiRing.sort (GRing_Ring__to__GRing_SemiRing R)) n
    ↪ n
44      A i
45      (@fun_of_perm.body
46      fintype_ordinal__canonical__fintype_Finite n) s i))))))

```

A.2 Instance insertion in Elpi database

Let I_1 and I_2 be two instances with priority 10 and let I_1 be defined before I_2 . When I_1 is added, it is placed after the hook named 10. The Elpi database after this insertion is the following:

```

...
:name "10" hook.
% Here the clause for  $I_1$ 
:name "11" hook.
...

```

Next, having priority 10, I_2 is placed in `tc.db` after the hook with name 10. The Elpi database will look as follows:

```

...
:name "10" hook.
% Here the clause for  $I_2$ 
% Here the clause for  $I_1$ 
:name "11" hook.
...

```

We can note that in the database, I_2 is placed before I_1 . This reflects the Coq's algorithm for instance insertion.

A.3 Goal reordering

In this section we want to provide some examples of how goal reordering (see [Section 4.4.1](#)) is done in Elpi.

Complex instance reordering To better show how the reordering of goals is done with respect to the constraints of an instance, we can take the instance `toSortInst` defined in [Listing A.3.1](#). Here, it is not so evident to immediately see if the constraints are well-ordered.

Listing A.3.1: A complex example of goal reordering

```

Class A (T1: Type) (T2 : Type).
Class B (T1: Type) (T2 : Type).
Class C (T1: Type) (T2 : Type) (T3 : Type) (T4 : Type).
Class D (T1: Type) (T2 : Type) (T3 : Type).

Hint Mode A + - : typeclass_instances.
Hint Mode B + - : typeclass_instances.
Hint Mode C - + - - : typeclass_instances.
Hint Mode D + + - : typeclass_instances.

Instance toSortInst (T1 T2 T3 T4 : Type)
  `(B T4 T1, C T1 T2 T3 T4, D T1 T3 T4) : A T1 T2.

```

The dependency graph corresponding to this instance is made up of four nodes, since there are three premises plus the goal $A \ T1 \ T2$. The node corresponding to the premise $B \ T4 \ T1$ has an entry arc labeled $T4$ and an exiting one labeled $T1$, since the first argument of B is in input and the second is in output mode. The other constraints are built in the same way. The special node for $A \ T1 \ T2$ has only exiting arcs. The partial graph corresponding to this instance is depicted in Figure 7a. The second step is to build a complete graph from it by linking all the exiting arcs with label l to all the entering arcs with label l . The complete graph is depicted in Figure 7b.

The sorting algorithm will start to remove the node A from the graph. In this way, all the arcs labeled with $T1$ and $T2$ are removed from it. This means that both arguments will be provided by the goal received in input. Note also that removing $T1$ from the node B to the node D removes a cycle.

Next, the node having no incoming edges is C , which is removed from the graph with the arcs labeled $T3$ and $T4$. Now, both the nodes B and D have no incoming edges, so they can be removed in any order. In our case, we choose B before D .

The final clause corresponding to `toSortInst` with sorted arguments is:

Listing A.3.2: Compiled clause for `toSortInst` with sorted premises

```
tc-A T1 T2 Sol1 {{toSortInst lp:T1 lp:T2 lp:T3 lp:T4 lp:SolB lp:SolC lp:SolD}} :-
  not (var T2), % T2 is in output, but should not be flexible since it is
              % an input of the type class C
tc-C T1 T2 T3 T4 SolC,
tc-B T4 T1 SolB,
tc-D T1 T3 T4 SolD.
```

Cyclic instance Let's take the instance `cyclicInstance` defined in Listing A.3.3. In this instance the two premises related to the type classes B and C have mutual dependencies.

The corresponding partial and complete graphs for this instance are depicted in Figure 8a and Figure 8b. The execution of the topological sort algorithm will remove the node A at first, but then the remaining sub-graph has a cycle where:

- B needs $T2$ as input which is provided by C
- C needs $T3$ which is provided by B

This instance can never be applied to any goal due to its mode and, therefore, Elpi's solver will throw a compilation error telling the instance `cyclicInst` cannot be compiled with respect to its modes.

Listing A.3.3: Instance with cyclic dependant constrains

```
Class A (T1: Type).
Class B (T1: Type) (T2 : Type) (T3 : Type).
Class C (T1: Type) (T2 : Type).

Hint Mode A + : typeclass_instances.
Hint Mode B + + - : typeclass_instances.
Hint Mode C + - : typeclass_instances.
```

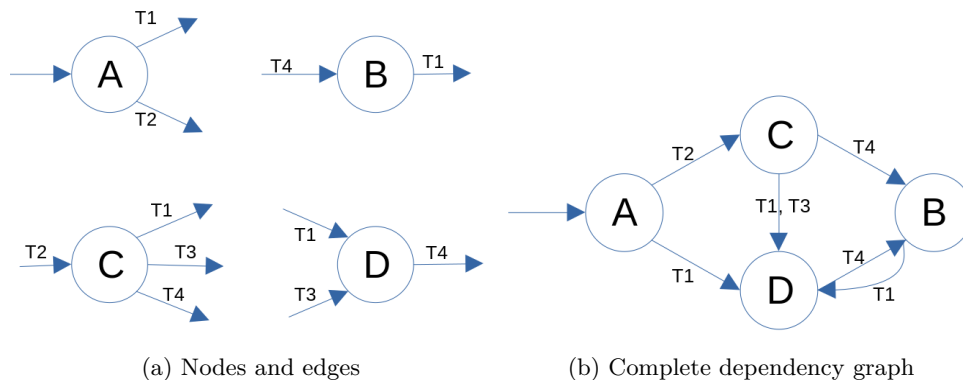


Figure 7: Graph for `toSortInst`

```
Instance cyclicInst (T1 T2 T3 : Type)
  `(B T1 T2 T3, C T3 T2) : A T1.
```

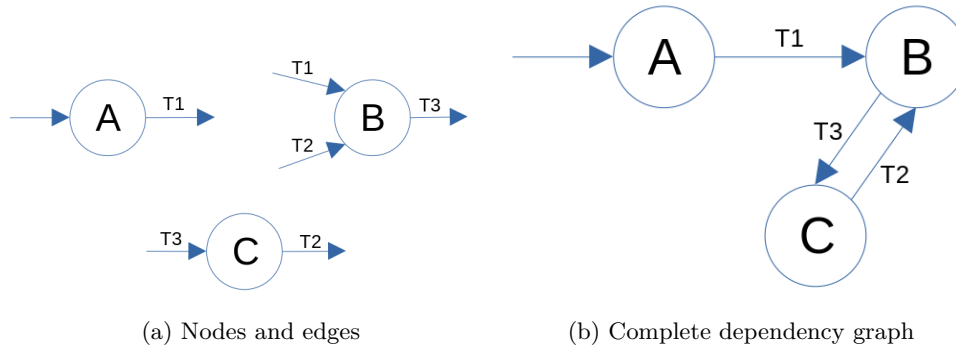


Figure 8: Graph for cyclicInst

Stuck instance In this final example, we take into account the instance `suckInst` defined in [Listing A.3.4](#). In this instance, the premise `B T2` needs a type `T2` in input, but there is no way to determine `T2`. The graph associated to `suckInst` is depicted in [Figure 9](#).

Listing A.3.4: Instance with not determinable argument

```
Class A (T1 : Type).
Class B (T1 : Type).

Hint Mode A + : typeclass_instances.
Hint Mode B + : typeclass_instances.

Instance stuckInst (T1 T2 : Type) `(B T2) : A T1.
```

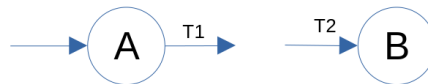


Figure 9: Graph for stuckInst

A.4 Some example

Goal in Elpi representation Let's consider the following goal:

```
Goal Inj eq eq (compose (fun x => (S (id x))) (fun x => (S (id x)))).
```

The corresponding term in Elpi is shown in [Listing A.4.1](#). In this code snippet, it is evident that doing indexing on the function `(compose (fun x => (S (id x))) (fun x => (S (id x))))` is not efficient since, in this case the indexing should be bigger than 20, since, to reach the sub-term representing the function `compose`, asks for an encoding of all the sub-terms before it.

Listing A.4.1: Goal of injective function

```
app [global (const «Inj»), X0, X1,
     app [global (indt «eq»), X2],
     app [global (indt «eq»), X3],
     app [global (const «compose»), X4, X5, X6,
          fun `x` X7 c0 \
            app [global (indc «S»),
                 app [global (const «id»), X8 c0, c0]],
```

```
fun `x` X9 c0 \  
  app [global (indc «S»),  
       app [global (const «id»), X10 c0, c0]]]
```