Canonical Structures for the working Coq user

Assia Mahboubi, Enrico Tassi

Canonical Structures for the working Coq user

Assia Mahboubi, Enrico Tassi

Inria

Abstract. This paper provides a gentle introduction to the art of programming type inference with the mechanism of Canonical Structures. Programmable type inference has been one of the key ingredients for the successful formalization of the Odd Order Theorem using the Coq proof assistant. The paper concludes comparing the language of Canonical Structures to the one of Type Classes and Unification Hints.

1 Introduction

One of the key ingredients to the concision, and intelligibility, of a mathematical text is the use of notational conventions and even sometimes the abuse thereof. These notational conventions are usually shaped by decades of practice by the specialists of a given mathematical community. If some conventions may vary according to the author's taste, most tend to stabilize into a well-established common practice. A trained reader can hence easily infer from the context of a typeset mathematical formula he is reading all the information that is not explicit in the formula but that is nonetheless necessary to the precise description of the mathematical objects at stake.

Formalizing a page of mathematics using a proof assistant requires the description of objects and proofs at a level of detail that is few orders of magnitude higher than the one at which a human reader would understand this description. This paper is about the techniques that can be used to reproduce at the formal level the ease authors of mathematics have to omit some part of the information they would need to provide, because it can be inferred. In the context of a large scale project like the formal proof of the Odd Order Theorem, which involves a large and broad panel of algebraic theories that should be both developed and combined, a faithful imitation of these practices becomes of crucial importance. Without them, the user of the proof assistant is soon overwhelmed by the long-windedness of the mathematical statements at stake.

What makes the meaning of mathematical conventions unambiguous and predictable is the fact that the human process of guessing the information that is not there can be described by an algorithm, which is usually very simple. In the Coq proof assistant, information which is not provided by the user is encoded in types, and the type inference algorithm is programmed to allow the user to omit information that can be inferred. Combining programmable type inference with user notations is a key design pattern of the libraries developed by the Mathematical Components team [18]. This combination is at the core

of the overloading of notations [8, 2], but also of the hierarchy of algebraic theories [5, 4, 3], as well as of various forms of automatic proof search and quotation [7].

The language used throughout the Mathematical Components libraries to program type inference is the one of Canonical Structures. Although this feature has been implemented in Coq from early versions of the system, it is much underrated by most of the community of users. As of today Canonical Structures actually remain poorly documented, to the notable exception of the work of Gonthier, Ziliani, Nanevski and Dreyer [10], where the authors detail the many technical issues one has to master in order to program proficiently proof search algorithms using Canonical Structures. This paper aims at providing a gentler introduction to Canonical Structures for the Coq user. Therefore we do not expose new contributions here but rather try to make a clear description of the methodologies employed to build the Mathematical Components libraries, so that they hopefully become accessible in practice to a wider audience. For this purpose, all the examples we draw in what follows are written in standard Coq version 8.4, without relying on any of the Mathematical Components library nor on the SSReflect shell extension [9] these libraries have been developed with. We expect the reader to be familiar with the vernacular language of Coq. The running example can be download at `http://ssr.msr-inria.inria.fr/doc/cs4wcu.v`

2 Canonical Structures

Saïbi [14] introduced both the Canonical Structure and the coercion mechanisms for the Coq system almost 15 years ago [14]. They have been available since version 6.1 of Coq system. However, while coercions were given visibility by publications in international venues [13], the only detailed description of Canonical Structures can be found in Saïbi's Ph.D. dissertation, that is written in French. The introduction of these mechanisms was motivated by the formalization of category theory due to Saïbi and Huet [12], in order to enhance the support offered by the Coq system for the formalization of algebraic concepts. The two mechanisms were seen as dual features: coercions were used to map a rich structure into a simpler one by forgetting parts of it and canonical structures where used to enrich a structure, in order for instance to restore the content discarded by the earlier insertion of a coercion.

In their formalization [12], Saïbi and Huet use Canonical Structures to overload function symbols [14, section 4.7, page 82] and to relate an algebraic theory with its instances. This technique has been re-used in a similar way for more recent large scale formalizations like the Fundamental Theorem of Algebra [6]. In his dissertation (see the example [14, 8.11.2, page 155]), Saïbi already spots that the type inference algorithm can iterate the Canonical Structure mechanism, but he seems to fail to grasp its potential and hence does not advertise this possibility. As a consequence, the documentation of Canonical Structures in the Coq manual (see [17], chapter 2.7.17) presents this mechanism as a non

iterative one, which can only be used to link abstract theories with the instances explicitly declared by the user.

In 2005, during the first year of the project toward the formalization of the Odd Order Theorem, Gonthier starts to make systematic use of Canonical Structures. In particular, he understands that the declaration of canonical instances can trigger an iterative process that lets one *program* type inference in a way that is reminiscent of Prolog. Combining this remark with the expressiveness of the type system of Coq it becomes quite natural to encode proof search into type inference.

The essence of the Canonical Structures mechanism is to extend the *unification* algorithm of the Coq system with a database of hints. Type inference compares types by calling the unification algorithm, that in turn looks into this database for solutions to problems that could not be solved otherwise. The user fills in the database by using a specific vernacular command to register the canonical solutions of his choice to some unification problems. Note that querying this database at unification time does not extend the trusted code base of the proof assistant. Just like the implicit arguments mechanism, this machinery is part of the proof engine and aims at decreasing the amount of type information provided by the user in order to describe a complete and well-formed Coq term.

In the current implementation the Canonical Structures database only stores solutions to unification problems of a very specific shape, that is the unification of a term with the projection of an unknown instance of a certain record type. This situation is typical of the issues faced when modeling algebraic structures with dependent record types. The following toy example illustrates how Canonical Structures have been used by their original authors. Suppose for instance that we have declared such a record type to define a naive interface for abelian (commutative) groups:

```
Structure abGrp : Type := AbGrp {
 carrier : Type;                zero : carrier;
 opp : carrier → carrier;       add : carrier → carrier → carrier;
 add_assoc : associative add;   add_comm : commutative add;
 zero_idl : left_id zero add;   add_oppl : left_inverse zero opp add }.
```

Here `carrier` is a projection extracting the type of the objects from a commutative group; similarly `zero` extracts the identity element, `add` the binary operation, ... We can prove the following theorem, valid for any instance of the structure:

```
Lemma subr0 : ∀ (aG : abGrp) (x : carrier aG), add aG x (opp aG zero) = x
```

Now suppose that we have constructed an instance of this interface which equips the type `Z` of integers with a structure of commutative group:

```
Definition Z_abGrp := AbGrp Z Z0 Z1 Zopp Zadd ....
```

Despite this effort, there is no way to use lemma `subr0` to simplify an expression of the form `(Zadd z (Zopp Z0))`, with `(z : Z)`, since Coq's unification algorithm is not aware of the content of our library. More precisely Coq does not know

that "Z forms a commutative group with Zadd, Zopp and Z0". The issue manifests itself as the difficult problem of unifying the type Z with (carrier ?), where ? represents the unknown commutative group. But if we declare Z_abGrp as the canonical commutative group over Z using the following command

```
Canonical Structure Z_abGrp : abGrp.
```

the unification algorithm is able to solve that problem by filling the hole with Z_abGrp.

In the rest of the paper we describe how Canonical Structures works, why these seemingly atomic hints are applied iteratively and how to build an algebraic hierarchy exploiting this fact.

3 Type inference and unification

The rich type theory of the Coq system confers to type inference the power of computing values and proofs. Due to this richness, the type inference algorithm of Coq is way more involved than the ones of mainstream programming languages, even in presence of a type class mechanism à la Haskell [19]. Yet only a fragment of this algorithm plays a role in understanding how Canonical Structures work. The purpose of this section is to give a picture of the relevant fragment of Coq's type inference algorithm, obviously without ambitioning to provide a complete or fully formal exposition of type inference. Let us start by defining the common and simplified syntax of Coq terms and types we will be working with:

$$t ::= t\ t \mid \pi_n \mid r \mid x \mid ?x$$

Since binders do not play a role here we omit them in this syntax and only consider applicative terms. On the contrary projections (resp. constructors) of record types are central to the canonical structures mechanism and deserve to be identified explicitly, by π_i (resp. r). We also need names (x) to represent declared or defined terms and also symbols ($?x$) for unification variables.

An algorithm takes as input some terms and an environment Γ that collects declarations ($t : T \in \Gamma$), definitions ($x := t \in \Gamma$) and unification variable assignments ($?x := t \in \Gamma$). The output of an algorithm is of the same kind, where the resulting environment Γ' is obtained from Γ by possibly assigning some unification variables. We express the definition of an algorithm in relational style, as inference rules. A rule defining $\mathcal{R}$ has the following shape:

$$\frac{(\Gamma, t_1)\ \mathcal{R}\ (\Gamma', t_2) \qquad (\Gamma', t_2)\ \mathcal{R}\ (\Gamma'', t_3)}{(\Gamma, t_1)\ \mathcal{R}\ (\Gamma'', t_3)}\ \text{rule name}$$

One should consider arguments on the left of the $\mathcal{R}$ symbol as the input of the $\mathcal{R}$ algorithm while the ones on the right as the output. The premises of the rule, representing calls to the same or other algorithms, are always performed in left to right order. In all what follows, for sake of brevity, we omit the Γ in the rules.

Type inference, denoted by ":", is defined by the following two rules:

$$\frac{t : T \in \Gamma}{t : T}\ env \qquad \frac{t_1 : \forall x : A, B \qquad t_2 : A' \qquad A' \sim A}{t_1\ t_2 : B[x/t_2]}\ app$$

Type inference recursively traverses a term imposing that all atoms are declared in our implicit environment Γ (*env* rule) and that the type expected by the head of an application coincides with the one of its actual argument (*app* rule). Type inference performs this type comparison by calling the unification algorithm, denoted by "$\sim$" and defined by the following rules:

$$\frac{}{t \sim t}\ eq \qquad \frac{}{?x \sim t}\ assign \qquad \frac{t_1 \sim t_2}{t\ t_1 \sim t\ t_2}\ fst\text{-}order \qquad \frac{t_2 \rhd t_2' \qquad t_1 \sim t_2'}{t_1 \sim t_2}\ red$$

Unification succeeds on syntactically identical terms thanks to the rule *eq*. Unification variables are assigned to terms by the rule *assign*, under the usual occurrence check and type compatibility conditions that we omit for sake of brevity. The algorithm applies the *fst-order* rule whenever the head symbols of the two compared terms are identical. The *red* rule replaces the term t_2 by its reduced form t_2' before continuing the unification. The *fst-order* rule has precedence over the *red* rule. It goes without saying that the unification algorithm implemented by Coq features many more rules than this simplified version, including the symmetric rules of *red* and *assign*. But we omit these extra rules which play no role for the topic of this tutorial. The following set of five rules defines the reduction algorithm, denoted "$\rhd$":

$$\frac{x := t \in \Gamma}{x \rhd t}\ unfold \qquad \frac{?x := t \in \Gamma}{?x \rhd t}\ subst \qquad \frac{t \rhd r\ t_1\ \ldots\ t_n}{\pi_i\ t \rhd t_i}\ proj$$

$$\frac{t_1 \rhd t_1'}{t_1\ t \rhd t_1'\ t}\ hd\text{-}red \qquad \frac{t_1 \rhd t_1' \qquad t_1' \rhd t_1''}{t_1 \rhd t_1''}\ trans$$

Reduction can unfold global constants, by the *unfold* rule, and substitute assigned unification variables, by the *subst* rule, as well as reduce projections applied to record constructors, by the *proj* rule. Reduction is closed transitively (*trans* rule) and with respect to applicative contexts (*hd-red* rule).

4 Basic overloading

In our first example we describe the infrastructure which creates an infix notation `==` that can be overloaded for several instances of binary comparisons. The "right" comparison function is chosen looking at the type of the compared objects. The user is required to declare a specific comparison function for each type of interest. The way of declaring such a function for a type is to build a dependent pair packaging together the type and the function and declare this pair as canonical.

Unfortunately this simple idea does not scale up properly: one often needs to attach to a type a bunch of operations (and properties) like in the abelian group example of section 2 and eventually reuse and extend the same set of operations

later on, for example when defining a field. Hence the general pattern is to define a special package type, called a *class*, to describe the set of operations of interest. The user then builds an instance of the class by providing all the operations, and packages that instance together with the type for which these operations are the canonical ones.

Going back to our example, we begin by defining the `class` of objects that can be compared using `==`. This class just contains the comparison operation, named `cmp`, and is parametrized over the type `T` of the objects to be compared. The class is modeled using a `Record` whose constructor is named `Class`. We then define the package one has to build in order to use the `==` notation on a specific type. This is again modeled using a record that packages together a type, called `obj`, and an instance of the `class` just defined on the type `obj`. The commands `Structure` and `Record` are actually synonyms in Coq, but we consistently use `Structure` to define this last type of packages, given that their instances can be made canonical using the `Canonical Structure` command.[1]

We consistently use modules as name spaces, so that the short names like `class` get a qualifying prefix `EQ.` once the name space definition is finished. For sake of clarity, even if we comment the code of an open name space, we use the qualified versions of the names like `EQ.type` or `EQ.class`.

```
Module EQ.
  Record class (T : Type) := Class { cmp : T → T → Prop }.
  Structure type := Pack { obj : Type; class_of : class obj }.
  Definition op (e : type) : obj e → obj e → Prop :=
    let 'Pack _ (Class the_cmp) := e in the_cmp.
  Check op. (* ∀e : EQ.type, EQ.obj e → EQ.obj e → Prop *)
  Arguments op {e} x y : simpl never.
  Arguments Class {T} cmp.
```

The constant `(EQ.obj : EQ.type → Type)` is a projection of the `EQ.type` record. In order to access the comparison operator present in the nested `class` record, we define the `EQ.op` projection, whose type is displayed in the above code. Note that the first argument `(e : EQ.type)` of `op` is declared as an implicit one by the `Arguments` command.

In a `theory` name space we declare the set of notations and establish the bunch of properties that are shared by all the instances of the `EQ.type` structure, here a single infix notation `==` for the `EQ.op` operator. Since the first argument of `EQ.op` is implicit, this notation actually hides a hole standing for an unknown record of type `EQ.type` from which the comparison operator is extracted.

```
  Module theory.
  Notation "x == y" := (op x y) (at level 70).
  Check ∀(e : type) (a b : obj e), a == b.
  End theory.
End EQ.
Import EQ.theory.
```

[1] A more appropriate name would be `Canonical Instance` or simply `Canonical`.

```
Fail Check 3 == 3.
(* Error: The term "3" has type "nat"
   while it is expected to have type "EQ.obj ?1". *)
```

The `Check` command inside the `EQ.theory` name space verifies that we can use the infix notation to develop the theory for objects in the `EQ.type` named `e`.

After closing the `EQ.theory` module, type checking the expression `(3 == 3)` fails. To understand the error message we write the critical part of the execution of the type inference algorithm, boxing the unsolvable unification problem. We denote by τ the type of the `EQ.op` constant:

$$\tau := \forall e : \texttt{EQ.type}, \texttt{EQ.obj}\ e \to (\texttt{EQ.obj}\ e \to \texttt{Prop}) \in \Gamma$$

$$\dfrac{\dfrac{\dfrac{\texttt{EQ.op} : \tau \in \Gamma}{\texttt{EQ.op} : \tau} \quad ?e : ?te \quad \dfrac{}{?te \sim \texttt{EQ.type}}}{\texttt{EQ.op}\ ?e : \tau[e/?e]} \quad 3 : \texttt{nat} \quad \boxed{\texttt{nat} \sim \texttt{EQ.obj}\ ?e}}{(\texttt{EQ.op}\ ?e)\ 3 : \texttt{EQ.obj}\ ?e \to \texttt{Prop}}$$

We can see that the unification problem involves a projection of an unknown instance $?e$ of the structure, and that the algorithm cannot invent which such instance would project by `EQ.obj` on type `nat`. Coq will only manage to synthesize a closed term from the input `(3 == 3)` after some more work from the user. We first need to define an instance `nat_EQty` of the structure which contains the comparison operator `nat_eq` that we want to use for objects of type `nat`.

```
Definition nat_eq (x y : nat) := nat_compare x y = Eq.
Definition nat_EQcl : EQ.class nat := EQ.Class nat_eq.
Canonical Structure nat_EQty : EQ.type := EQ.Pack nat nat_EQcl.
Check 3 == 3. (* Works! *)
Eval compute in 3 == 4. (* Evaluates to Lt = Eq, indeed 3 ≠ 4. *)
```

We have moreover turned this definition into a canonical instance, via the `Canonical Structure` command: this extends the $\sim$ algorithm with the rule:

$$\dfrac{\texttt{nat} \sim \texttt{EQ.obj nat\_EQty} \qquad ?x \sim \texttt{nat\_EQty}}{\texttt{nat} \sim \texttt{EQ.obj}\ ?x} \tag{1}$$

This rule has two premises: the first one verifies that the `obj` component of the `nat_EQty` structure is `nat`, while the second one hints the solution for $?x$.

The following execution shows that the first premise is trivially satisfied:

$$\dfrac{\dfrac{\dfrac{\texttt{nat\_EQty} := \texttt{EQ.Pack nat nat\_EQcl} \in \Gamma}{\texttt{nat\_EQty} \rhd \texttt{EQ.Pack nat nat\_EQcl}}\ \mathit{unfold}}{\texttt{EQ.obj nat\_EQty} \rhd \texttt{nat}}\ \mathit{proj} \qquad \dfrac{}{\texttt{nat} \sim \texttt{nat}}\ \mathit{eq}}{\texttt{nat} \sim \texttt{EQ.obj nat\_EQty}}\ \mathit{red}$$

This reduction also shows that, once `nat_EQty` is assigned to $?x$, thanks to the *subst* rule the unification problem $\texttt{nat} \sim \texttt{EQ.obj}\ ?x$ is solved by simply reducing the right hand side to `nat`.

Actually, the hint generated by the `Canonical Structure` command is the following one, where the right hand side of the first premise is obtained by reducing the term (`EQ.obj nat_EQty`) nto head normal form. This reduction is performed once and for all when the instance `nat_EQty` is made canonical.

$$\frac{\texttt{nat} \sim \texttt{nat} \qquad ?x \sim \texttt{nat\_EQty}}{\texttt{nat} \sim \texttt{EQ.obj}\ ?x} \tag{2}$$

Note that the first premise is not always trivial as this one. In the next example it plays the crucial role of iterating Canonical Structures resolution.

The last line of the Coq script shows that type inference has indeed found the right comparison function and can compute that `3` is different from `4`.

The next step is to be able to use the `==` on compound objects, like a pair of natural numbers. Hence we declare a *family* of hints providing a canonical method to compare pairs of objects when they both live in equality structures.

```
Fail Check ∀(e : EQ.type) (a b : EQ.obj e), (a,b) == (a,b).
(* Error: The term "(a, b)" has type "(EQ.obj e * EQ.obj e)"
   while it is expected to have type "EQ.obj ?15". *)
Definition pair_eq (e1 e2 : EQ.type) (x y : EQ.obj e1 * EQ.obj e2) :=
  fst x == fst y /\ snd x == snd y.
Definition pair_EQcl (e1 e2 : EQ.type) := EQ.Class (pair_eq e1 e2).
Canonical Structure pair_EQty (e1 e2 : EQ.type) : EQ.type :=
  EQ.Pack (EQ.obj e1 * EQ.obj e2) (pair_EQcl e1 e2).
```

We use the infix `*` to denote the type of pairs and we declare the obvious equality function over pairs as canonical. Note that `pair_EQty` has two parameters, `e1` and `e2`, of the same type. The following hint is added to the unification algorithm:

$$\frac{t_1 * t_2 \sim \texttt{EQ.obj}\ ?y * \texttt{EQ.obj}\ ?z \qquad ?x \sim \texttt{pair\_EQty}\ ?y\ ?z}{t_1 * t_2 \sim \texttt{EQ.obj}\ ?x} \tag{3}$$

Note that t_1 and t_2 are arbitrary terms, and hence the new rule is applicable whenever the unification problem involves the `EQ.obj` projection and the type constructor `*`. The unification variables $?y$ and $?z$ are fresh.

The following reduction justifies the shape of the first premise.

$$\cfrac{\cfrac{\cfrac{\texttt{pair\_EQty} := \texttt{EQ.Pack}\ \ldots \in \Gamma}{\texttt{pair\_EQty} \rhd \texttt{EQ.Pack}\ \ldots}\ \textit{unfold}}{\texttt{pair\_EQty}\ ?y\ ?z \rhd \texttt{EQ.Pack}\ (\texttt{EQ.obj}\ ?y * \texttt{EQ.obj}\ ?z)\ldots}\ \textit{hd-red}}{\texttt{EQ.obj}\ (\texttt{pair\_EQty}\ ?y\ ?z) \rhd \texttt{EQ.obj}\ ?y * \texttt{EQ.obj}\ ?z}\ \textit{proj}$$

It goes without saying that the first premise of (3) composes with the *fst-order* rule, given that both sides have the same head constant `*`. One could thus reformulate the hint in the following way, where the recursive calls on smaller, but similar, unification problems is evident:

$$\frac{t_1 \sim \texttt{EQ.obj}\ ?y \qquad t_2 \sim \texttt{EQ.obj}\ ?z \qquad ?x \sim \texttt{pair\_EQty}\ ?y\ ?z}{t_1 * t_2 \sim \texttt{EQ.obj}\ ?x} \tag{4}$$

This hint is correct by induction: if unification finds a value for $?y$ so that (`EQ.obj` $?y$) unifies with t_1 and a value for $?z$ so that (`EQ.obj` $?z$) unifies with t_2, then (`pair_EQty` $?y$ $?z$) is a solution for the problem, because it reduces to (`EQ.obj` $?y$ $*$ `EQ.obj` $?z$) that unifies with ($t_1 * t_2$) by the *fst-order* rule.

It is now clear that type inference can make sense of a comparison by the `==` infix notation of any nested pairs of objects, as soon as their types are equipped with registered equality structures, and that this could be made to work for other type constructors like `list`, `option`, etc...

5 Inheritance

In this section we show how to organize structures depending on one others by implementing inheritance. Let us start assuming we have defined a structure of types equipped with an order operator in a module named `LE`, exactly as we did for `EQ`, with an infix notation `<=`. As we equipped `nat` with an `EQ.type` canonical structure `nat_EQty` we also equip it with an `LE.type` canonical structure `nat_LEty`. Now we can mix the two operators `==` and `<=` on objects of type `nat`. This however is not sufficient for the development of the abstract theory shared by types that are instances of both structures:

```
Check 2 <= 3 /\ 2 == 2. (* Works! *)
Fail Check ∀(e : EQ.type) (x y : EQ.obj e), x <= y → y <= x → x == y.
(* Error: The term "x" has type "EQ.obj e"
   while it is expected to have type "LE.obj ?32". *)
```

In this failing example, the type of `x` and `y` is extracted from the equality structure, which provides no mean of guessing a related order structure. Using `LE.type` and `LE.obj` instead of `EQ.type` and `EQ.obj` would yield a similar error.

We need to craft a structure equipped with both comparison and order, and also impose that the two operations can be combined is a sensible way. We begin expressing this compatibility property. The packaging of this single property into a record is overkilling, but we aim at exposing the general schema here:

```
Module LEQ.
  Record mixin (e : EQ.type) (le : EQ.obj e → EQ.obj e → Prop) :=
    Mixin { compat : ∀(x y : EQ.obj e), (le x y /\ le y x) ↔ x == y }.
```

To express the property we need a type `e` of objects being comparable with `==` and an additional operation named `le`. The `mixin` is the only extra component we need to build the new class starting from the one of `EQ` and the one of `LE`.

```
  Record class T := Class {
    EQ_class : EQ.class T;
    LE_class : LE.class T;
    extra : mixin (EQ.Pack T EQ_class) (LE.cmp T LE_class) }.
  Structure type := _Pack { obj : Type; class_of : class obj }.
  Arguments Mixin {e le} _.
  Arguments Class {T} _ _ _.
```

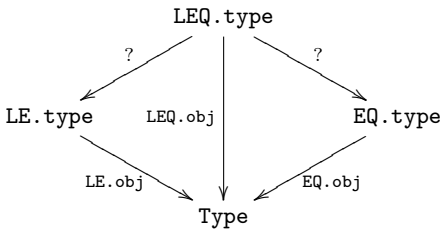
Some choices are arbitrary in this definition. For example one could use a symmetric mixin taking in input an `LE.type` and an operation `eq`. In general one parametrizes the mixin so to ease the writing of its content.

The reason why the `_Pack` constructor has been named this way becomes clear in the next section, where we implement a smarter constructor `Pack` to ease the declaration of canonical instances.

Unfortunately we have not yet fulfilled our goals completely:

```
Module theory.
Fail Check ∀(leq : type) (n m : obj leq), n <= m → n <= m → n == m.
(* Error: The term "n" has type "LEQ.obj leq"
   while it is expected to have type "LE.obj ?44". *)
```

The error message is better explained in the following graph where the nodes are types and the edges represent the inheritance relation. The two edges from `LEQ.type` to `EQ.type` and `LE.type` are missing: Coq does not know that an instance of the interface `LEQ.type` is also an instance of `LE.type` and `EQ.type`.



Again the issue manifests as an hard unification problem, that luckily falls in the domain of Canonical Structures. We only need to provide the following hints:

```
Definition to_EQ (leq : type) : EQ.type :=
  EQ.Pack (obj leq) (EQ_class _ (class_of leq)).
Definition to_LE (leq : type) : LE.type :=
  LE.Pack (obj leq) (LE_class _ (class_of leq)).
Canonical Structure to_EQ.
Canonical Structure to_LE.
```

The hint generated by the last command is the following one:

$$\frac{\texttt{LEQ.obj}\ t \sim \texttt{LEQ.obj}\ ?y \qquad ?x \sim \texttt{LEQ.to\_LE}\ ?y}{\texttt{LEQ.obj}\ t \sim \texttt{LE.obj}\ ?x} \tag{5}$$

The following reduction justifies the shape of the first premise and the correctness of the hinted solution:

$$\dfrac{\dfrac{\dfrac{\texttt{LEQ.to\_LE} := \texttt{LE.Pack}\ \ldots \in \Gamma}{\texttt{LEQ.to\_LE} \rhd \texttt{LE.Pack}\ \ldots}\ \textit{unfold}}{\texttt{LEQ.to\_LE}\ ?y \rhd \texttt{LE.Pack}\ (\texttt{LEQ.obj}\ ?y)\ldots}\ \textit{hd-red}}{\texttt{LE.obj}\ (\texttt{LEQ.to\_LE}\ ?y) \rhd \texttt{LEQ.obj}\ ?y}\ \textit{proj}$$

Now we can state a lemma using both == and <= on an abstract type leq. In this simple case the proof is just the content of the mixin.

```
  Lemma lele_eq (leq : type) (x y : obj leq) : x <= y → y <= x → x == y
  Proof. ... Qed.
  Arguments lele_eq {leq} x y _ _.
  End theory.
End LEQ.
```

Even if objects of type nat can be compared with both == and <=, we did not prove that the corresponding operations are compatible. All we have to do is to build an instance of LEQ.type over nat and declare it as canonical.

```
Import LEQ.theory.
Example test1 (n m : nat) : n <= m → m <= n → n == m.
Proof. Fail apply (lele_eq n m). Abort.
(* Error: The term "n" has type "nat"
   while it is expected to have type "LEQ.obj ?48". *)
Lemma nat_LEQ_compat (n m : nat) : n <= m → m <= n → n == m.
Proof. ... Qed.
Definition nat_LEQmx := LEQ.Mixin nat_LEQ_compat.
Canonical Structure nat_LEQty : LEQ.type :=
  LEQ._Pack nat (LEQ.Class nat_EQcl nat_LEcl nat_LEQmx).
```

6 Proof search

Just like in section 4, we can program a generic instance of equality-and-order structure for pairs of equality-and-order instances. Then the look-up for canonical instances can be iterated in a similar way, but this time for the equality, order and equality-and-order instances at once:

```
Lemma pair_LEQ_compat (l1 l2 : LEQ.type) (n m : LEQ.obj l1 * LEQ.obj l2):
  n <= m → m <= n → n == m.
Proof. ... Qed.
Definition pair_LEQmx (l1 l2 : LEQ.type) :=
  LEQ.Mixin (pair_LEQ_compat l1 l2).
Canonical Structure pair_LEQty (l1 l2 : LEQ.type) : LEQ.type :=
  LEQ._Pack (LEQ.obj l1 * LEQ.obj l2)
    (LEQ.Class
      (EQ.class_of (pair_EQty (to_EQ l1) (to_EQ l2)))
      (LE.class_of (pair_LEty (to_LE l1) (to_LE l2)))
      (pair_LEQmx l1 l2)).
Example test2 (n m : (nat * nat) * nat) : n <= m → m <= n → n == m.
Proof. now apply (lele_eq n m). Qed.
```

The proof of the toy example test2 illustrates that the look-up for an instance of equality-and-order structure, which justifies the use of lemma lele_eq, truly amounts to a proof search mechanism. Indeed, neither have we ever programmed

explicitly the operators equipping type `(nat * nat) * nat`, not have we proved the requirements of lemma `lele_eq` by hand on that instance: both the programs and the proofs have been synthesized from generic patterns.

7 Declaring instances made easier

The declaration of the canonical structure `pair_LEQty` and `nat_LEQty` is still unsatisfactory: it is not only very verbose, but also very redundant. Indeed we had to provide by hand many components the system is in principle able to infer. In particular the `EQ.type` and `LE.type` structures for `nat` and the pair type are provided explicitly by hand even if we have registered them as canonical. In this section we show how to program a packager that given the type and the mixin that characterizes the `LEQ.type` we are building, infers all the remaining fields.

The main difficulty is that the information registered in the Canonical Structures database can be retrieved only when a precise problem is posed to the unification algorithm. We hence start with the description of some generic Swiss knife which allows to pose arbitrary problems to the unification algorithm and to extract the resulting information.

The first problem to overcome is that unification (and its Canonical Structures database) is used by type inference to process *types*, while in many occasions we want to enforce the unification of *terms*.[2] However we can inject values into types, even artificially, using the dependent types of Coq. To this purpose we define and use a `phantom` type. This construction is also used in the programming language context to trick the type system into enforcing extra invariants [11].

```
Module infrastructure.
  Inductive phantom {T : Type} (t : T) : Type := Phantom.
  Definition unify {T1 T2} (t1 : T1) (t2 : T2) (s : option string) :=
    phantom t1 → phantom t2.
  Definition id {T} {t : T} (x : phantom t) := x.
  Notation "[find v | t1 ∼ t2 ] rest" :=
    (fun v (_ : unify t1 t2 None) => rest) ...
  Notation "[find v | t1 ∼ t2 | msg ] rest" :=
    (fun v (_ : unify t1 t2 (Some msg)) => rest) ...
  Notation "'Error: t msg" := (unify _ t (Some msg)) ...
End infrastructure.
```

Note that in order to improve the error messages, and to facilitate debugging, the `unify` function optionally holds a string representing an error message. The notation "`[find v | t1 ∼ t2 | msg ] rest`" should be read as: "find `v` such that `t1` unifies with `t2` or fail with `msg`", then continue with `rest`.

The second problem is that, once terms are lifted to the types level, we still need to be able to call the unification procedure *at the right moment*. The infrastructure we want to build has to be generic, i.e. be expressed for an arbitrary

[2] It is avoidable in this context, but we prefer to present this mechanism in its full generality given its ubiquity in the Mathematical Components library.

type `T` and mixin `m`, but there is no canonical structure for an arbitrary type `T`. We need to run the unification procedure only when `T` is provided. To this extent the terms to be unified are stored as `unify t1 t2` that is defined as `phantom t1 → phantom t2`.[3] Each argument of type `unify` will be later instantiated with an identity function, whose type `phantom t → phantom t` (for some `t`) is required to match `phantom t1 → phantom t2`, and hence forces the unification of `t1` with `t2`.

With this infrastructure we can define a packager to build an inhabitant `(LEQ._Pack T (LEQ.Class ce co m))` of the `LEQ.type` record given the type for the objects `T` and the mixin `m0`.

```
Import infrastructure.
Definition packager T e0 le0 (m0 : LEQ.mixin e0 le0) :=
    [find e  | EQ.obj e ~ T       | "is not an EQ.type" ]
    [find o  | LE.obj o ~ T       | "is not an LE.type" ]
    [find ce | EQ.class_of e ~ ce ]
    [find co | LE.class_of o ~ co ]
    [find m  | m ~ m0             | "is not the right mixin" ]
  LEQ._Pack T (LEQ.Class ce co m).
Notation Pack T m := (packager T _ _ m _ id _ id _ id _ id).
```

The parameters `e0` and `le0` are needed only for the argument `m0` to be well-typed. Now remark that this packager eventually builds an instance of the `LEQ.type` structure from the input type `T` plus three components `ce`, `co` and `m` that are not calculated from the input mixin `m0` by a Coq function. The component `ce` (resp. `co`) is the class field of the record `e` (resp. `o`), which is itself an `EQ.type` (resp. `LE.type`) that is inferred as the canonical instance over `T`. Finally `m` is forced to be equal to the input mixin `m0`. The `Pack` notation is just a macro to provide the packager with enough unknown values `_` and enough identity functions for the unification problems to be triggered *when the notation is used and* `T` *is provided*. Canonical instance declarations can then be shortened as follows:

```
Canonical Structure nat_LEQty := Eval hnf in Pack nat nat_LEQmx.
Canonical Structure pair_LEQty (l1 l2 : LEQ.type) :=
  Eval hnf in Pack (LEQ.obj l1 * LEQ.obj l2) (pair_LEQmx l1 l2).
```

When we declare the first canonical instance, type inference gives a type to `(Pack nat nat_LEQmx)`. All the `_` part of the `Pack` notation are inferred by the unification algorithm that in turn finds a value for `e`, then `o`, then `ce`, then `co` and finally `m`. The "`Eval hnf in`" command reduces away the abstractions in the body of the packager. Error reporting is also quite accurate:

```
Fail Canonical Structure e := Eval hnf in Pack bool nat_LEQmx.
(* ... 'Error: bool "is not an EQ.type" *)
Fail Canonical Structure e := Eval hnf in Pack (nat * nat) nat_LEQmx.
(* ... 'Error: nat_LEQmx "is not the right mixin" *)
```

Note that, in this simple case, the packager could be simplified by using explicitly the `e0` parameter of the `m0` mixin for the value of the `e` component.

[3] On the contrary using `t1 = t2` would force their type to be immediately unifiable.

8 Conclusions and Related Works

As a conclusion we compare three similar mechanisms that have been implemented independently in proof assistants based on the Calculus of Inductive Constructions: Coq's Type Classes [15], which have also been used to develop hierarchies of algebraic structures [16]; Coq's Canonical Structures [14], which we have presented in this tutorial; and Matita's Unification Hints [1], that can be seen as a generalization of Canonical Structures.

	Type Classes	Canonical Structures	Unification Hints
1^{st} class	Yes	Yes	Yes
local instance	Yes	Almost	Yes
search engine	ad hoc	unification	unification
priority/overlap	explicit	encodable	explicit
backtracking	native	encodable	/
package/inheritance	unbundled/trivial	bundled/easy	bundled/easy
narrowing	/	No	Yes

1^{st} **class**: With all these mechanisms, interfaces are record types, hence first class objects. This is fundamental to build new types and instances on the fly in the middle of proofs — one cannot for instance declare a module in the middle of a proof. This is also crucial to develop a generic theory by quantifying on all the instances of such a structures to express statements like "the first order theory of an algebraically closed field has the quantifier elimination property" [3].

Local instance: Even if new structure instances can be built in the middle of a proof, Canonical Structures require all the building blocks to be globally defined. In other words there is no support for hinting the unification using a construction one obtains in the middle of a proof. On the contrary Type Classes take into account local instances available in the proof context. Actually one can program unification to postpone all unification problems that fail because of a missing Canonical Structure in the following way:

```
Canonical Structure failsafe t f : EQ.type := EQ.Pack t (EQ.Class f).
Set Printing All.
Check (true == true). (* @EQ.op (failsafe bool ?9) true true : Prop *)
Fail Lemma test : true == true.
(* Cannot infer an internal placeholder of type "bool → bool → Prop *)
```

The `failsafe` Canonical Structure has a special status, given that its `obj` component `t` is a parameter. Canonical Structures of this kind are called "default" and are used only if no other canonical structure can be applied. With this hint type inference can give a type to `(true == true)` even if there is no `EQ.type` declared as canonical for the type `bool`. Unfortunately Coq 8.4 handles unresolved unification variables like `?9` in a non uniform way. For example a statement of a theorem cannot contain unresolved variables.

Search engine: This is certainly the major limitation of all three mechanisms, that, to our knowledge, are equally undocumented for that respect. To understand failures one may need to dig into the sources of the systems.

Backtracking and priorities: Backtracking is a native mechanism of Type Classes' search engine. This mechanism offers vernacular commands to assign weighted priorities to the instances declared by the user. Instances can overlap and the search engine may try all possibilities. Instances of Canonical Structures cannot overlap. However, one can alias a term by means of a definition and encode priorities into a chain of aliases (See [10], section 2.3). Instances of Unification Hints can overlap, and they are tried in an order provided explicitly by the user. The language for the declaration of Unification Hints has a syntax very close to the one we used in this paper to explain hint (4) in section 4 and a notion of cut à la Prolog would clearly apply but is not currently implemented.

Package/Inheritance: Different mechanisms impose different styles to declare the interfaces. Type Classes enforce the unbundled approach, where the values used to search instances have to be parameters of the structure. Canonical Structures and Unification Hints enforce a bundled (also called packed) approach, where the values used to search for instances have to be fields of the structure. Implementing inheritance relation between interfaces is possible using all three mechanisms. The tricky case is the one of multiple inheritance like our `LEQ` of section 5, where a structure has to inherit from two a priori unrelated structures that nonetheless provide operators and specifications on the same type. This problem is trivial with unbundled structures, as the common domain type constraint can be expressed syntactically. With structures expressed in bundled style the best solution known is the one of packed-classes [5] we used in section 5, that provides a general solution to single and multiple inheritance. This solution is used consistently in the whole web of interfaces defined in the Mathematical Components libraries.

Narrowing: Type Classes do not extend unification, so this point does not apply. Canonical Structures extend unification only when dealing with problems involving a record projection. Unification Hints let the user extend unification more freely. For example a unification hint could enable the unification algorithm to solve the problem "$?x$ `* (S n)` $\sim$ `0`" by assigning $?x$ to `0`.

A very natural question is if it is possible to mix Canonical Structures and Type Classes in Coq. The answer is yes. For example one can build an `EQ.type` instance using the Type Classes search engine as follows:

```
Existing Class EQ.class.
Canonical Structure failsafe T {c : EQ.class T} : EQ.type := EQ.Pack T c.
Instance bool_EQcl : EQ.class bool := EQ.Class bool bool_cmp.
Check true == true. (* Works! ?9 assigned to bool_EQcl *)
```

The converse is also possible.

```
Class eq_class {A} : Type := Class { cmp : A → A → Prop }.
Notation "x === y" := (cmp x y) (at level 70).
Instance find_CS (e : EQ.type) : eq_class := Class (EQ.obj e) (@EQ.op e).
Set Printing All.
Check 0 === 0. (* @cmp nat (find_CS nat_EQty) O O : Prop *)
```

Acknowledgments. We are grateful to Frédéric Chyzak, Georges Gonthier, Laurence Rideau and Matthieu Sozeau for proofreading this text.

References

[1] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. Hints in unification. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009.

[2] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In *TPHOLs*, volume 5170 of *LNCS*, pages 86–101, 2008.

[3] C. Cohen. *Formalized algebraic numbers: construction and first order theory.* PhD thesis, École polytechnique, 2012.

[4] F. Garillot. *Generic Proof Tools and Finite Group Theory.* PhD thesis, École polytechnique, 2011.

[5] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *TPHOLs*, volume 5674 of *LNCS*, pages 327–342. Springer, 2009.

[6] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in coq. *J. Symb. Comput.*, 34(4):271–286, 2002.

[7] G. Gonthier. Point-free, set-free concrete linear algebra. In *ITP*, volume 6898 of *LNCS*, pages 103–118. Springer, 2011.

[8] G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A Modular Formalisation of Finite Group Theory. In *TPHOLs*, pages 86–101, 2007.

[9] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2012.

[10] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP*, pages 163–175. ACM, 2011.

[11] R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming, Cornerstones of Computing*, pages 245–262, 2003.

[12] G. P. Huet and A. Saïbi. Constructive category theory. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, pages 239–276. The MIT Press, 2000.

[13] A. Saïbi. Typing algorithm in type theory with inheritance. In P. Lee, F. Henglein, and N. D. Jones, editors, *POPL*, pages 292–301. ACM Press, 1997.

[14] A. Saïbi. *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types: application à la Théorie des Catégories.* PhD thesis, Université Paris VI, 1999.

[15] M. Sozeau and N. Oury. First-Class Type Classes. In *Theorem Proving in Higher Order Logics, 21th International Conference*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, August 2008.

[16] B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.

[17] The Coq Development Team. *The Coq proof assistant, version 8.4.* http://coq.inria.fr.

[18] The Mathematical Component Team. A Formalization of the Odd Order Theorem using the Coq proof assistant, September 2012. http://www.msr-inria.inria.fr/Projects/math-components/feit-thompson.

[19] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of POPL*, pages 60–76. ACM, 1989.