# Private Inductive Types
# Proposing a language extension

Yves Bertot

April 7, 2013

**Abstract**

For the Coq system, we propose to add the possibility to declare an inductive type as private to the module where it is defined. The effect is to preserve the possibility to compute with a restricted set of functions, but to disallow uses of the more powerful pattern-matching constructs. Such a private type has fruitful applications in homotopy type theory.

## 1 Description of the behavior

We wish to propose a modification of the syntax and behavior of inductive type declarations, where the user can specify which computations are allowed on such a type. This modification is only a restriction of expressive power, so that such an extension does not directly endanger the consistency of the proof system.

### 1.1 Informal presentation

The extension works as follows. An inductive type `T` declared inside a module `M` can be declared as `Local`. Inside the module `M`, only one change is visible: the induction theorem `T_ind`, `T_rect`, `T_rec` are not generated; users may then decide to define their own version of these theorems. Aside from this absence of usual theorems, all usual manipulations about the inductive type are available inside the module. In particular, pattern-matching works as usual.

When the module `M` is closed and then used in other context, the type `T` is still visible to the user, together with any other functions that were defined inside the module, *but pattern-matching is no longer possible*. In practice, all functions defined with `T` as argument type must use the set of functions chosen by the user to perform any computation on this types values. An important aspect is that these functions can still be computed, so that a function applied to a collection of argument is convertible to the predicted value.

These are the only two changes: no access to the standard induction principles (everywhere) and no access to the pattern-matching construct (outside the defining module).

This extension caters to the need of providing functions that compute, so that `a: A` and `f a` is convertible to some `c` without exposing the whole structure of the type `A`. Modules already provide a good amount of information hiding, but this not sufficient, because if a module contains an axiom expressing `f a = c`, then `f a` and `c` are not convertible, which makes many proofs unwieldy.

### 1.2 Illustration

The following example provides an inductive type and only one function to define a value on this type.

```
Module collapsed_bool.
```

```
Local Inductive cbool : Type := ctrue | cfalse.

Definition cbool_rec1 {B : Type} (v1 v2 : cbool) (p : v1 = v2) (x : cbool) : B :=
  match x with c1 => v1 | c2 => v2 end.

End collapsed_bool.

Import collapsed_bool.
```

This module description adds only four objects in the environment: `cbool : Type`, `c1 : Type`, `c2 : Type` and `cbool_rec1`. Moreover, `cbool_rec1 u v p ctrue` is convertible to `u`. As a result, it is impossible to prove that `c1` and `c2` are different, because this requires defining a function with `cbool` as input type, but any such function has to obey the constraint that the values given to `ctrue` and `cfalse` have to be equal.

## 2 Implementation ideas

There is currently a simulation of this concept available on `git://github.com/HoTT/coq.git` in the branch `private_type`. However, this implementation does not respect all other functionalities provided by the Coq system (especially with respect to undoing). The implementation approach is to annotate all inductive definitions with an extra flag which is mutated when the module is closed. Then all other capabilities from the Coq system check this flag before allowing the construction of pattern-matching constructs. This implementation is not satisfactory because definitions are not supposed to be mutated, even when modules are closed. As a result, the undo mechanism does not handle these private types very well (in interactive sessions).

An alternative implementation idea would rely on an extra flag as above and limit the construction of pattern-matching constructs on private types to the cases where the private type's module path is a prefix of the module where the pattern-matching construct is being constructed. In both case, the restriction is enforced at type-checking time, not at computation time.

## 3 Uses in Homotopy Type Theory

In Homotopy Type Theory, equality is given a slightly different meaning than the one computer scientists are accustomed to. It makes sense to define types with several points and equalities between them, while retaining the capability to compute on these points (with full convertibility). Such types are called *higher inductive type*. To keep computation capabilities, it is important that the several equal points be described as constructors of an inductive type. But since one wants to add equality between these points, it is also important to prevent the user from proving that the various constructors are different. This is ensured by providing functions like `cbool_rec1` above. Once this is ensured, it is then possible to add the equalities between points by adding axioms.

The question of consistency of the system after using private types and adding axioms is still unresolved. It is not the operation of adding private types that endangers consistency, but the action of adding axioms describing the equalities. In many cases, the axioms would simply break consistency in the presence of regular inductive types and it remains to be proved that the restrictions imposed by private types are sufficient to recover consistency. However, there are higher inductive types with only one point (and extra equalities) where the extra axioms would not break consistency, even without privacy.

## 4 Uses in other domains

At the workshop, we would like to take the opportunity to discuss with the audience about other possible uses of such an extension.