Inconsistency of classical logic in type theory

Herman Geuvers

November 2007

1 Introduction

In this note, we show the inconsistency of a strong version of classical logic in the type theory of Coq. More precisely, we show that from the assumption $\forall A{:}\mathsf{Prop}\{A\} + \{\neg A\}$, we can derive $\bot$. The type $\forall A{:}.\mathsf{Prop}\{A\} + \{\neg A\}$ (in Coq notation: `(A:Prop){A}+{~A}`) is more powerful than the 'ordinary classical axiom $\forall A{:}.\mathsf{Prop} A \vee \neg A$ (in Coq notation: `(A:Prop)A \/ ~A`), because it allows to define a function to a Set type by cases. This is precisely one of the crucial steps in the proof of the inconsistency: the construction of two functions $\mathsf{b2P} : \mathsf{bool}{\to}\mathsf{Prop}$ and $\mathsf{P2b} : \mathsf{Prop}{\to}\mathsf{bool}$ that form a retract from Prop to bool: $\forall A{:}\mathsf{Prop}.(\mathsf{b2P}(\mathsf{P2b}A)) \leftrightarrow A$.

The fact that we can derive an inconsistency from a retract from Prop to bool has already been noticed by Coquand in [2], in a slightly different form. There the inconsistency is established by showing how the (inconsistent) type system λU^- can be embedded if we have such a retract. Here we show the inconsistency directly, by adapting Hurkens' version of Girard's paradox (see [4]). The result is presented below by giving the Coq code of the inconsistency proof. To make the whole a bit more understandable, we first present the Lego code of the Hurkens inconsistency proof in λU^-.

2 Hurkens' proof of the inconsistency of λU^-

In [4], a very short version of Girard's paradox is presented, showing the inconsistency of type theory where $\mathsf{Type} : \mathsf{Type}$. As a matter of fact, the inconsistency can already be derived in λU^-, which corresponds to higher order predicate logic over polymorphic domains. (In terms of Pure Type Systems, the system with sorts $\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'$, axioms $\mathsf{Prop} : \mathsf{Type} : \mathsf{Type}'$ and the rules (for forming Π-types and λ-abstractions) $(\mathsf{Prop}, \mathsf{Prop})$, $(\mathsf{Type}, \mathsf{Prop})$, $(\mathsf{Type}, \mathsf{Type})$ and $(\mathsf{Type}', \mathsf{Type})$.)

Having seen the proof of Hurkens, Randy Pollack and the author wanted to formalize the proof in Lego, also to see whether a fixed point combinator could be obtained from it. This was done in the summer of 1994 and the Lego code is presented below. Of course, this proof development cannot be done in standard Lego (or Coq), as they do not allow $\mathsf{Type} : \mathsf{Type}$ (nor do they allow Type to be polymorphic). However, if one uses `TypeInType()` in Lego, one tells the type checker not to verify the universe conditions, hence allowing Type : Type. Then the following is a correct proof development in Lego.

Note for those familiar with Coq: `{i:U->Type}` denotes the Π-abstraction, written in Coq as `(i:U->Type)`. Also `{A|Type}` denotes a Π-abstraction, but now with `A` being an implicit argument, to be filled in by the type checker. In Lego, application binds stronger than abstraction, allowing to write `[z:V]r (z r) a`, where one would have to write `[z:V](r (z r) a)` in Coq. The `Refine` tactic corresponds – roughly – to the `Apply` tactic in Coq. The `;` symbol ends a Lego-command, just like the `.` in Coq.

```
[V = {A|Type}((A->Type)->(A->Type))->A->Type];
[U = V->Type];
[sb [A|Type][r:(A->Type)->(A->Type)][a:A] = [z:V]r (z r) a : U];
[le [i:U->Type][x:U] =
        x ([A|Type][r:(A->Type)->(A->Type)][a:A]i (sb r a)) : Type];
[induct [i:U->Type] = {x:U}(le i x)->i x : Type];
[WF = [z:V]induct (z le) : U];
[B:Type];
[I [x:U] = ({i:U->Type}(le i x)->i (sb le x))->B : Type];

Goal {i:U->Type}(induct i)->i WF;
intros i y;
Refine y WF ([x:U]y (sb le x));
Save omega;

Goal induct I;
Intros x p q;
Refine q I p ([i:U->Type]q ([y:U]i (sb le y)));
Save lemma;

Goal  ({i:U->Type}(induct i)->i WF)->B;
intros x;
Refine x I lemma ([i:U->Type]x ([y:U]i (sb le y)));
Save lemma2;

Goal B;
Refine lemma2 omega;
Save paradox;
```

This is a direct encoding of the proof of [4]. If one adds a declaration `F : B->B` and replaces the line `Refine q I p ([i:U->Type]q ([y:U]i (sb le y)));` by `Refine F(q I p ([i:U->Type]q ([y:U]i (sb le y))));`, one obtains also a proof of `B`, which now is a *looping combinator*. This is a list of terms `Y_1`, `Y_2`, ..., such that `Y_1 F` is convertible with `F(Y_2 F)`, `Y_2 F` is convertible with `F(Y_3 F)`, etcetera. See [3] for a description of looping combinators in type theory. If one looks at these looping combinators in a *Curry-style* way, i.e. ignoring all the types, both in the λ-abstractions and in the polymorphic applications, they are all *fixed-point combinators*. So, if we ignore the types, we find that `Y F` is convertible with `F(Y F)`.

3 Adapting Hurkens' paradox to an inconsistency of classical logic in Coq

The above paradox can be adapted to Coq, by adding (a strong form of) the classical axiom and by replacing Type with bool. In the original paradox, $A \to \mathsf{Type}$ is used as the type of predicates over A, and then $\Pi A{:}\mathsf{Type}(A \to \mathsf{Type})$ can not itself be of type Type (unless of course we allo Type : Type). However, if we replace Type with bool, we have – due to the impredicativity of Set – that $\Pi A{:}\mathsf{Set}(A \to \mathsf{bool}) : \mathsf{Set}$. To get an inconsistency, we now only need to be able to reflect Prop inside bool, which can be done if we assume the strong form of classical logic:

$$cl : \forall A{:}\mathsf{Prop}.\{A\} + \{\ A\}$$

This allows us to define mappings P2b and b2P from Prop to bool and back that form a retract, i.e. we can prove

$$\forall A{:}\mathsf{Prop}.(\mathsf{b2P}(\mathsf{P2b}A)) \leftrightarrow A.$$

The proof of inconsistency of `cl : (A:Prop){A} + {~A}` in Coq, now is as follows.

```
Require Bool.
Axiom cl : (A:Prop){A} + {~A}.

Definition P2b : Prop -> bool := [A:Prop]
        Cases (cl A) of
                (left _) => true |
                (right _) => false
        end.
Definition b2P : bool -> Prop := [b:bool]
        Cases b of
                true  => True |
                false  => False
        end.

Lemma p2p : (A:Prop)(b2P(P2b A))<->A.
Intro A;Split.
Unfold b2P P2b.
Elim (cl A).
Auto.
Intros; Contradiction.
Unfold b2P P2b.
```

```
Intro; Elim (cl A); Auto.
Qed.

Syntactic Definition p2p_1 := (proj1 ??(p2p ?)).
Definition retract : (A:Prop)(A->(b2P(P2b A)))
        := [A:Prop](proj2 ?? (p2p A)).
Definition V : Set
        := (A:Set)((A->bool)->(A->bool))->A->bool.
Definition U : Set
        := V->bool.
Definition sb : (A:Set)((A->bool)->A->bool)->A->V->bool
        := [A:Set;r:(A->bool)->(A->bool);a:A][z:V](r (z ? r) a).
Syntactic Definition Sb := (sb ?).
Definition le : (U->bool)->U->bool
        := [i:U->bool][x:U]
                (x ([A:Set][r:(A->bool)->(A->bool)][a:A](i (Sb r a)))).
Definition induct : (U->bool)->bool
        := [i:U->bool](P2b((x:U)(b2P(le i x)) -> (b2P(i x)))).
Definition WF  : V->bool
        := [z:V](induct (z ? le)).
Variable B:Prop.
Definition I : U->bool
        := [x:U] (P2b(
                ((i:U->bool)(b2P(le i x)) ->(b2P(i (Sb le x)))) ->B)).

Lemma omega: (i:U->bool)(b2P(induct i))-> (b2P(i WF)).
Intros i y.
Unfold induct in y.
Generalize (p2p_1 y).
Intros.
Apply H.
Unfold le WF induct.
Apply retract.
Intros x H0.
Apply H.
Exact H0.
Qed.

Lemma lemma:(b2P(induct I)).
Unfold induct.
Apply retract.
Intros x p.
Unfold I.
Apply retract.
Intro q.
Generalize (q I p).
```

```
Intro H.
Unfold I in H.
Generalize (p2p_1 H).
Intro H0.
Apply H0.
Intro i; Exact (q ([y:U](i (Sb le y)))).
Qed.

Lemma lemma2: ( (i:U->bool)(b2P(induct i)) -> (b2P(i WF)) ) -> B.
Intro x.
Generalize (x I lemma).
Intro H.
Unfold I WF in H.
Generalize (p2p_1 H).
Intro H1.
Apply H1.
Intro i.
Generalize (x ([y:U](i (Sb le y)))).
Intros H0 H3.
Apply H0.
Exact H3.
Qed.

Lemma par : B.
Exact (lemma2 omega).
Qed.
```

3.1 Inconsistency in $\lambda P2$

Coquand has shown in [2] that the following context Γ_0 is inconsistent in the Calculus of Constructions.

$$\Gamma_0 := A : \mathsf{Prop}, \epsilon : A \to \mathsf{Prop}, E : \mathsf{Prop}- > A, h : \Pi A{:}\mathsf{Prop}.(\epsilon(EA)) \leftrightarrow A$$

This was proved by embedding the (inconsistent) type system λU^- in this context. In [2], Coquand also raised the question whether a direct (simpler) proof of the inconsistency of Γ_0 could be given. This can indeed be done if we adapt our proof of inconsistency of $\forall A{:}\mathsf{Prop}.\{A\}+\{\neg A\}$ as follows. Replace the definitions `b2P`, `P2b` and the proof `P2P` in the above piece of Coq code by

```
Variable boo:Prop.
Variable b2P:boo->Prop.
Variable P2b:Prop->boo.
Variable P2P : (A:Prop)(b2P(P2b A))<->A.
```

and replace, in the rest of the proof, bool by `boo` and Set by Prop. In this way we find a proof of inconsistency of the context Γ_0 in the type system $\lambda P2$ (second order dependent type theory).

3.2 A slight weakening of the inconsistency proof

The inconsistency proof uses the inductive booleans, with dependent elimination over it and an inductive sum type with dependent elimination over it. The dependent elimination over the sum type can be weakened. We can restrict it to elimination over *polymorphic dependent predicates*. The standard dependent elimination rule yields

$$\begin{array}{lcl}\mathsf{sumbool\_ind} & : & \forall A,B{:}\mathsf{Prop}.\forall P{:}\{A\}{+}\{B\}{\to}\mathsf{Prop}.\\ & & (\forall a{:}A.P(\mathsf{left}\,ABa))\\ & & {\to}(\forall b{:}B.P(\mathsf{right}\,ABb))\\ & & {\to}\forall s{:}\{A\}{+}\{B\}.(Ps)\end{array}$$

This can be restricted to polymorphic predicates over sum types (i.e. $P : \Pi X,Y{:}\mathsf{Prop}.\{X\}{+}\{Y\}{\to}\mathsf{Prop}$). Then we have the following.

$$\begin{array}{lcl}\mathsf{OR\_ind} & : & \forall P : \Pi X,Y{:}\mathsf{Prop}.\{X\}{+}\{Y\}{\to}\mathsf{Prop}.\forall A,B{:}\mathsf{Prop}.\\ & & (\forall a{:}A.PAB(\mathsf{left}\,ABa))\\ & & {\to}(\forall b{:}B.PAB(\mathsf{right}\,ABb))\\ & & {\to}(\forall s{:}\{A\}{+}\{B\}.(PABs)).\end{array}$$

Using this restricted form of dependent elimination over sum types, we can still prove the incocnsistency, as is shown by the Coq code below. To be sure that we don't use the dependent elimination, we have declared a new sum type axiomatically. We only show the code until we have defined the mappings `P2b : Prop -> bool` and `b2P : bool -> Prop` and we have proved that they form a retract:

```
Lemma P2P : (A:Prop)(b2P(P2b A)) <-> A.
```

The rest of the inconsistency proof is the same as in the previous case.

```
Variable OR : Prop -> Prop -> Set.
Variable ORleft : (A,B:Prop)(A-> (OR A B)).
Variable ORright : (A,B:Prop)(B-> (OR A B)).

Variable OR_rec : (A,B:Prop; P:Set)
         (A->P) -> (B->P) -> (OR A B) -> P.

Variable OR_ind
     : (P:(X,Y:Prop)(OR X Y)->Prop)(A,B:Prop)
        ((a:A)(P ?? (ORleft A B a)))
        -> ((b:B)(P ?? (ORright A B b)))
        -> (s:(OR A B))(P ?? s).

Axiom eqor1 : (A,B:Prop; P:Set)
        (f1:(a:A)P) (f2:(b:B)P) (a:A)
        (OR_rec ??? f1 f2 (ORleft ?? a)) = (f1 a).

Axiom eqor2 : (A,B:Prop; P:Set)
        (f1:(a:A)P) (f2:(b:B)P) (b:B)
        (OR_rec ??? f1 f2 (ORright ?? b)) = (f2 b).

Axiom cl : (A:Prop)(OR A (~ A)).

Definition P2b : Prop -> bool := [A:Prop]
        (OR_rec A (~A) bool ([x:A]true) ([y:~A]false) (cl A)).

Definition b2P : bool -> Prop := [b:bool]
        Cases b of
                true  => True |
                false  => False
        end.

Lemma P2P : (A:Prop)(b2P(P2b A)) <-> A.
Proof.
Intro A;Split.
Unfold b2P P2b.
Apply (OR_ind
([X,Y:Prop;o:(OR X Y)](if (OR_rec X Y bool [_:X]true [_:(Y)]false o)
    then True
    else False)
   ->A)).
Intros.
Auto.
Intros.
Rewrite eqor2 in H.
Contradiction.
Unfold b2P P2b.
Intro.
Apply (OR_ind
([X,Y:Prop;o:(OR X Y)](if (OR_rec X Y bool [_:X]true [_:(Y)]false o)
    then True
    else False))).
Intro a; Rewrite eqor1.
Auto.
Intro b; Rewrite eqor2.
Elim b.Auto.
Qed.
```

References

[1] Barthe G. and Sørensen M. 2000, ‘Domain-free Pure Type Systems’, *Journal of Functional Programming* **10**, 417–452. Preliminary version in S. Adian and A. Nerode, editors, Proceedings of LFCS’97, LNCS 1234, pp 9-20.

[2] Coquand, T. 1989, An introduction to type theory, notes of the FPCL summer school, Glasgow 1989.

[3] Coquand, T. and Herbelin, H. 1994, A-translation and looping combinators in Pure Type Systems, *Journal of Functional Programming* 4(1).

[4] Hurkens A. 1995, A simplification of Girard’s paradox, in Dezani-Ciancaglini, M. Plotkin, G., eds, *Second International Conference on Typed Lambda Calculi and Applications*, TLCA’95, Vol. 902 of LNCS, pp. 266–278.