Notes on axiomatising Hurkens's Paradox

Arnaud Spiwack

July 15, 2015

Abstract

An axiomatisation of Hurkens's paradox in dependent type theory is given without assuming any impredicative feature of said type theory.

Hurkens's paradox [8] is a very economic, though rather hard to understand, paradox of the U^- impredicative type theory, described in Section 1.1, whose main characteristic is to feature to nested impredicative sorts. Its terseness makes it the weapon of choice to derive inconsistencies from logical principle or experimental language features of your favourite proof assistant. Or, rather, embedding U^- in some way is the weapon of choice, Hurkens's paradox serves as a way to turn this into a proof of false.

It may sound like a futile game to play: if you are the ideal mathematician you will never implement inconsistent feature in your proof assistant. Unfortunately, you are not, and deriving contradiction will happen to you from time to time. Having a tool for that may turn out to be of tremendous help. As a bonus, the inconsistency of U^- can serve to derive potentially useful principles, such as the fact that if the principle of excluded middle holds in an impredicative sort, then types in that sort have the proof irrelevance property (see Section 2.4).

The downside in all that is that there does not seem to be a good way to express, within dependent type theory, the existence of an impredicative sort. Coquand [3] gave a sufficient condition, albeit much stronger, to derive contradictions in a generic way. His proof was based on Girard's [6] paradox rather than Hurkens's one (which came out ten years later). Geuvers [5] later gave a proof based more directly on Hurkens's one and relying on a single impredicative sort, but this proof wasn't very generic. The result was that Hurkens's proof was included *twice* in the distribution of the Coq proof assistant [9]: Geuvers's proof, and a variant due to Hugo Herbelin to prove slightly different results.

This situation is certainly unsatisfactory, as adapting Hurkens's proof for every little variation around the same theme is significantly more work than describing an encoding of U^-. It prevents good people from finding perfectly good proof of contradictions: it isn't fair to assume that everyone is an expert in Hurkens's proof.

As it happens, however, there is a perfectly good axiomatisation of U^- in your favourite dependently typed proof assistant (in actuality, a sufficient

subsystem). And the corresponding proof of contradiction is, *mutatis mutandis*, Geuvers's, where conversion rules are replaced by equalities.

1 Axiomatic Hurkens's paradox

The trick, so to speak, of the axiomatix presentation of U^- is generally attributed to Martin-Löf: a universe is given by an type U:**Type** describing the types in the universe, and an decoding function El:U→**Type** describing, for each type in U the elements of that types. Sorts are to be encoded as such universes. System U^- has two of these, commonly called *large* and *small*, together with rules to combine them. Each of these rules take the form of a product formation rule (see Barendregt's presentations of *pure type systems*, formerly known as *generalised type systems* [1][2, Section 5.2]). Instead of the usual presentation where there is a single dependent product with a number of formation rules, we will have a distinct dependent product – with its own introduction rule (λ-abstraction) and elimination rule (application) – for each of the formation rule. For each pair λ-abstraction & application, there may be a β-equivalence rule, modelled as an equality; only the β-equivalence rules which are effectively used in the proof are axiomatised.

1.1 Axiomatic U^-

The full axiomatic presentation appears below, in Coq syntax. It is also part of Coq's distribution and can be found, at the time these notes are being written, in the file theories/Logic/Hurkens.v.

Large universe

```
Variable U1 : Type.
Variable El1 : U1 → Type.
```

The large universe U1 is closed by dependent products over types in U1. The definition of dependent product and λ-abstraction are defined using the function space of the dependent type theory. Notations are defined for dependent product, λ-abstraction and application. As usual, an arrow notation is used when the dependent product has a constant range.

```
Variable Forall1 : forall u:U1, (El1 u → U1) → U1.
  Notation "'∀₁' x : A , B" := (Forall1 A (fun x ⇒ B)).
  Notation "A '⟶₁' B" := (Forall1 A (fun _ ⇒ B)).
Variable lam1 : forall u B, (forall x:El1 u, El1 (B x)) → El1 (∀₁ x:u, B x).
  Notation "'λ₁' x , u" := (lam1 _ _ (fun x ⇒ u)).
Variable app1 : forall u B (f:El1 (∀₁ x:u, B x)) (x:El1 u), El1 (B x).
  Notation "f '·₁' x" := (app1 _ _ f x).
Variable beta1 : forall u B (f:forall x:El1 u, El1 (B x)) x,
                 (λ₁ y, f y) ·₁ x = f x.
```

The large universe U1 is made impredicative by a dependent product with large domain. The standard presentation would use a sort U2, of which U1 is a member; the dependent product would then have, as a domain, some

T:U2. This would be unnecessary complexity as U2 is so restricted that the only interesting type in it would be U1. So, instead, we simply restrict the domain of the product to be U1.

```
Variable ForallU1 : (U1→U1) → U1.
  Notation "'∀₂' A , F" := (ForallU1 (fun A ⇒ F)).
Variable lamU1 : forall F, (forall A:U1, El1 (F A)) → El1 (∀₂ A, F A).
  Notation "'λ₂' x , u" := (lamU1 _ (fun x ⇒ u)).
Variable appU1 : forall F (f:El1(∀₂ A,F A)) (A:U1), El1 (F A).
  Notation "f '·₁' [ A ]" := (appU1 _ f A).
Variable betaU1 : forall F (f:forall A:U1, El1 (F A)) A,
                  (λ₂ x, f x) ·₁ [ A ] = f A.
```

Small universe The small universe U0 is an element of the larger one. Therefore we need an u0:U1 and U0 is taken to be El1 u0 rather than a variable.

```
Variable u0 : U1.
Notation U0 := (El1 u0).
Variable El0 : U0 → Type.
```

The small universe U0 is closed by dependent products in U0. The definitions are symmetric to the corresponding ones of U1. Notice, however, the lack of β-rule, which is unnecessary to derive a contradiction.

```
Variable Forall0 : forall u:U0, (El0 u → U0) → U0.
  Notation "'∀₀' x : A , B" := (Forall0 A (fun x ⇒ B)).
  Notation "A '⟶₀' B" := (Forall0 A (fun _ ⇒ B)).
Variable lam0 : forall u B, (forall x:El0 u, El0 (B x)) → El0 (∀₀ x:u, B x).
  Notation "'λ₀' x , u" := (lam0 _ _ (fun x ⇒ u)).
Variable app0 : forall u B (f:El0 (∀₀ x:u, B x)) (x:El0 u), El0 (B x).
  Notation "f '·₀' x" := (app0 _ _ f x).
```

The small universe U0 is made impredicative by a dependent whose range is in U1. Contrary to the impredicative product of U1, the range cannot be restricted to be only U0. Here again, the β-rule is not needed.

```
Variable ForallU0 : forall u:U1, (El1 u→U0) → U0.
  Notation "'∀₀¹' A : U , F" := (ForallU0 U (fun A ⇒ F)).
Variable lamU0 : forall U F, (forall A:El1 U, El0 (F A)) → El0 (∀₀¹ A:U, F A).
  Notation "'λ₀¹' x , u" := (lamU0 _ _ (fun x ⇒ u)).
Variable appU0 : forall U F (f:El0(∀₀¹ A:U,F A)) (A:El1 U), El0 (F A).
  Notation "f '·₀' [ A ]" := (appU0 _ _ f A).
```

1.2 Proof of contradiction

From there, we can proceed to use Hurkens's argument to derive a contradiction. Let's be precise: we shall prove that every type in U0 is inhabited. It will only be an actual contradiction if U0 contains the empty type. For this purpose, let's assume a type in U0, we will then prove it is inhabited.

```
Variable F:U0.
```

The proof will require simplifying β-redexes. We provide tactics to that effect.

```
Ltac simplify :=
  (repeat rewrite ?beta1, ?betaU1);
  lazy beta.

Ltac simplify_in h :=
  (repeat rewrite ?beta1, ?betaU1 in h);
  lazy beta in h.
```

These tactics are rather brute-force, in that they will β-reduce as much as possible without any particular strategy. On the other hand, they, crucially, don't unfold Coq definitions so that we can give them hints by manually unfolding the appropriate terms to be simplified. Allowing the simplification tactics to unfold Coq definitions turns out to be intractable.

It is traditional to regard U1 as the type of datatypes and U0 as the type of proposition. This view is justified by the fact that U0 is not equipped with β-conversion rules. In the proof, following Geuvers [5], data is explicitly given, while propositions are proved with tactics. Here are the data definitions (I'm playing a bit loose here, since I consider propositions to be data, they are according to the above definition at least):

```
Definition V : U1 := ∀₂ A, ((A ⟶₁ u0) ⟶₁ A ⟶₁ u0) ⟶₁ A ⟶₁ u0.
Definition U : U1 := V ⟶₁ u0.

Definition sb (z:El1 V) : El1 V := λ₂ A, λ₁ r, λ₁ a, r ·₁ (z·₁[A]·₁r) ·₁ a.

Definition le (i:El1 (U⟶₁u0)) (x:El1 U) : U0 :=
  x ·₁ (λ₂ A, λ₁ r, λ₁ a, i ·₁ (λ₁ v, (sb v) ·₁ [A] ·₁ r ·₁ a)).
Definition le' : El1 ((U⟶₁u0) ⟶₁ U ⟶₁ u0) := λ₁ i, λ₁ x, le i x.
Definition induct (i:El1 (U⟶₁u0)) : U0 :=
  ∀₀¹ x:U, le i x ⟶₀ i ·₁ x.

Definition WF : El1 U := λ₁ z, (induct (z·₁[U] ·₁ le')).
Definition I (x:El1 U) : U0 :=
  (∀₀¹ i:U⟶₁u0, le i x ⟶₀ i ·₁ (λ₁ v, (sb v) ·₁ [U] ·₁ le' ·₁ x)) ⟶₀ F
.
```

The proofs follow Geuvers [5] as well. The main difference is that we must explicitly call to simplify where conversion was used implicitly and that standard Coq tactics calls to the *intro* and *apply* tactics are generally replaced by tactics of the form *refine* (λ_0 x, _) and *refine* (h·$_0$_) respectively.

Lemma Omega : El0 ($\forall_0{}^1$ i:U$\longrightarrow_1$u0, induct i $\longrightarrow_0$ i $\cdot_1$ WF).
Proof.
refine ($\lambda_0{}^1$ i, λ_0 y, _).
refine (y$\cdot_0$[_]$\cdot_0$_).
unfold le,WF,induct. simplify.
refine ($\lambda_0{}^1$ x, λ_0 h0, _). simplify.
refine (y$\cdot_0$[_]$\cdot_0$_).
unfold le. simplify. *unfold* sb *at* 1. simplify. *unfold* le' *at* 1. simplify.
exact h0.
Qed.

Lemma lemma1 : El0 (induct (λ_1 u, l u)).
Proof.
unfold induct.
refine ($\lambda_0{}^1$ x, λ_0 p, _). simplify.
refine (λ_0 q,_).
assert (El0 (l (λ_1 v, (sb v)$\cdot_1$[U]$\cdot_1$le'$\cdot_1$x))) *as* h.
{ *generalize* (q$\cdot_0$[λ_1 u, l u]$\cdot_0$p). simplify.
intros q'. *exact* q'. }
refine (h$\cdot_0$_).
refine ($\lambda_0{}^1$ i,_).
refine (λ_0 h', _).
generalize (q$\cdot_0$[λ_1 y, i $\cdot_1$ (λ_1 v, (sb v)$\cdot_1$[U] $\cdot_1$ le' $\cdot_1$ y)]). simplify.
intros q'.
refine (q'$\cdot_0$_). *clear* q'.
unfold le *at* 1 **in** h'. simplify_in h'.
unfold sb *at* 1 **in** h'. simplify_in h'.
unfold le' *at* 1 **in** h'. simplify_in h'.
exact h'.
Qed.

Lemma lemma2 : El0 (($\forall_0{}^1$i:U$\longrightarrow_1$u0, induct i $\longrightarrow_0$ i$\cdot_1$WF) $\longrightarrow_0$ F).
Proof.
refine (λ_0 x, _).
assert (El0 (l WF)) *as* h.
{ *generalize* (x$\cdot_0$[λ_1 u, l u]$\cdot_0$lemma1). simplify.
intros q.
exact q. }
refine (h$\cdot_0$_). *clear* h.
refine ($\lambda_0{}^1$ i, λ_0 h0, _).
generalize (x$\cdot_0$[λ_1 y, i$\cdot_1$(λ_1 v, (sb v)$\cdot_1$[U]$\cdot_1$le'$\cdot_1$y)]). simplify.
intros q.
refine (q$\cdot_0$_). *clear* q.
unfold le **in** h0. simplify_in h0.
unfold WF **in** h0. simplify_in h0.
exact h0.
Qed.

```
Theorem paradox : El0 F.
Proof.
  exact (lemma2·0Omega).
Qed.
```

The takeaway insight is that because the paradox does not actually make use of the reduction rules in propositions of U0, using equality to model conversion in these propositions doesn't raise any obstacle to the completion of the proof.

Nothing in this proof is particularly specific to Coq: it could be done in any variant of Martin-Löf type theory, provided that an identity type is available. Of course, the support of Coq for rewriting significantly helps, if your favourite proof assistant doesn't have a similar feature it may be painful to port this generic paradox.

2 Applications

In this section we will see a few instances of the generic axiomatisation of Hurkens's proof can help derive contradictions. They come from the file theories/Logic/Hurkens.v of the Coq distribution (version 8.5).

2.1 Sorts

A common implementation of universes is to use a sort of the dependent type theory for a universe of U^-. In that case. El is just the identity.

```
Variable U := Type.
Let El := fun X ⇒ X.
```

For universes defined this way, small products and their λ-abstraction, application and β-rule are defined straightforwardly (eq_refl is Coq's witness of reflexivity of equality).

```
Let Forall (A:U) (B:A → U) : U := forall x:A, B x
Let lam u B (f:forall x:A,B x) := f
Let app u B (f:forall x:A,B x) (x:A) := f x
Let beta u B f x : f x = f x := eq_refl
```

2.2 Impredicative sort

Impredicativity, for a sort U, can also be characterised to some degree. The idea is that there must be a bigger sort U' which can be projected onto U. See, for example, the bracketing construction in [7]. This projection could be implemented, for instance, for Coq's impredicative **Prop** sort as **fun** X:**Type** ⇒ **forall** P:**Prop**, (X→P)→P.

The signature of Section 2.1 is extended with the constraint that U' is bigger than U and a projection.

```
Let U' := Type.
Let U:U' := Type.
Variable proj : U' → U.
```

With the following laws.

```
Hypothesis proj_unit : forall (A:U'), A → proj A.
Hypothesis proj_counit : forall (F:U→U), proj (forall A,F A) → (forall A,F A).
Hypothesis proj_coherent : forall (F:U → U) (f:forall x:U, F x) (x:U),
                           proj_counit _ (proj_unit _ f) x = f x.
```

The proj_unit law expresses that if proj generally diminishes the ability to distinguish between elements of A:U2, it does not lose elements. We don't have a way back from proj A to A in general, but proj forms a monad. The proj_unit law expresses a small variation on this latter remark.

These properties are sufficient to show that U is closed by large product. The β-rule, omitted, is easily derived from proj_coherent.

```
Let ForallU (F:U→U) : U := proj (forall A, F A).
Let lamU1 F (f:forall A:U, F A) : proj(forall A:U, F A):= proj_unit _ f
Let appU1 F (f:proj(forall A:U, F A)) (A:U) : F A := proj_counit _ f x.
```

We can exploit Coq's universe polymorphism (form version 8.5) to turn this section into a generic definition of impredicative sort. Indeed, under the polymorphic interpretation **Type** represents an arbitrary type, including the impredicative sort **Prop**, which is indeed impredicative in the above sense.

2.3 Generalising Geuvers's proof

Geuvers [5] proves that an impredicative sort U1 cannot be a retract of an U0:U1. His proof is made for U1 = **Prop**, but we can instantiate the proof of Section 1 to obtain the same result for any sort which is impredicative sort in the sense of Section 2.2.

```
Let U2 := Type.
Let U1:U2 := Type.
Variable U0:U1.
```

Where U1 is impredicative over U2 as in Section 2.2. The retraction is given by the following functions. Only a weak form of retraction is needed were types in U1 which are "logically equivalent" are considered equal.

```
Variable proj0 : U0 → U1.
Variable inj0 : U1 → U0.
Hypothesis inj0_unit : forall (b:U1), b → proj0 (inj0 b).
Hypothesis inj0_counit : forall (b:U1), proj0 (inj0 b) → b.
```

From this (weak) retraction we can define El0 and corresponding products for U0 despite the fact that U0 is not necessarily a sort.

```
Let El0 (u:U0) := proj0 u
Let Forall0 (u:U0) (B:proj0 u → U0) : U0 := inj0 (forall x:proj0 u, proj0 (B x))
Let Lambda0 u B (f:forall x:proj0 u, proj0 (B x))
  : proj0 (inj0 (forall x:proj0 u, proj0 (B x))) := inj0_unit _ f.
Let app0 forall u B (f:proj0 (inj0 (forall x:proj0 u, proj0 (B x)))) (x:proj0 u)
  : proj0 (B x) := inj0_counit _ f x
```

Large products are define much the same:

```
Let Forall0 (u:U1) (B:u → U0) : U0 := inj0 (forall x:u, proj0 (B x))
Let Lambda0 u B (f:forall x:u, proj0 (B x))
  : proj0 (inj0 (forall x:u, proj0 (B x))) := inj0_unit _ f.
Let app0 forall u B (f:proj0 (inj0 (forall x:u, proj0 (B x)))) (x:u)
  : proj0 (B x) := inj0_counit _ f x
```

From this, the paradox is set up, so we can deduce that every proposition of P:U0 is "inhabited" in that El0 P = proj0 P is inhabited, and therefore, that every proposition of F:U1 is inhabited since inj0 F:U0 is "inhabited" in the sense of U0, *i.e.* proj0 (inj0 F) is inhabited, then inj0_counit concludes.

Since **Prop** is an instance of the signature of Section 2.2, we prove, like Geuvers, that **Prop** is not a retract of a proposition P:**Prop**.

2.4 Excluded middle and proof irrelevance

Geuvers proof, from Section 2.3, helps proving a result, by Coquand [4], that excluded middle, in an impredicative sort makes it proof irrelevant, *i.e.* every type in that sort have at most one element. This proof appear in the Coq distribution in the file theories/Logic/ClassicalFact.v, presumably written by Hugo Herbelin. It uses Geuvers result and was mostly unmodified with the new proof of said result. With the characterisation of Section 2.2, this could be done in an arbitrary impredicative sort, but the Coq proof is done only for the impredicative sort **Prop**, and we will present it that way for simplicity.

The basic idea is that excluded middle:

```
Variable em: forall A:Prop, A∨¬A.
```

turns the **Prop** sort into a boolean universe with only two elements. So assuming a proposition with two *distinct* values

```
Variable U0:Prop.
Variables t f : U0.
Hypothesis not_eq_t_f : t ≠ f.
```

we can reflect **Prop** into U0 proposition as in Section 2.3. Where True is reflected as t and False as f, as the names suggest.

This is formalised as a retraction given by:

```
Let inj0 (A:Prop) : U0 := or_ind A (¬A) U0 (fun _ ⇒ t) (fun _ ⇒ f) (em A).
Let proj0 (x:U0) : Prop := t = x.
```

Where or_ind:forall A B P : **Prop**, (A → P) → (B → P) → A ∨ B → P is the elimination principle of disjunction.

We are left to prove the unit and counit laws of inj0 and proj0 to satisfy the premisses of the paradox in Section 2.3. The unit law is direct:

```
Lemma inj0_unit : forall A:Prop, A → proj0 (inj0 A).
Proof.
 intros A x. unfold proj0, inj0.
 destruct (em A) as [h | h].
 + reflexivity.
 + contradiction.
Qed.
```

The counit law is the step that makes a crucial use of the not_eq_t_f hypothesis:

```
Lemma inj0_counit : forall A:Prop, proj0 (inj0 A) → A.
Proof.
 intros A h. unfold proj0, inj0 in *.
 destruct (em A) as [l | l].
 + apply l.
 + absurd (t=f).
  * apply not_eq_t_f.
  * apply h.
Qed.
```

Section 2.3 then yields a contradiction. Since U0 is arbitrary we have: **forall** (A:**Prop**) (x y:A), ¬¬(x=y). A last application of the excluded middle yields the expected result:

```
forall (A:Prop) (x y:A), x=y
```

2.5 Variants of Prop

A (monadic) modality on **Prop** is given by a mapping:

```
Variable M : Prop → Prop.
```

Together with the following laws:

```
Hypothesis unit : forall A:Prop, A → M A.
Hypothesis join : forall A:Prop, M (M A) → M A.
Hypothesis incr : forall A B:Prop, (A→B) → M A → M B.
```

Such a modality is automatically equipped with a distribution property over arbitrary conjunctions:

```
Lemma strength: forall A (P:A→Prop), M(forall x:A,P x) → forall x:A,M(P x).
Proof.
 eauto.
Qed.
```

With a modality we can define the type of modal propositions, where the unit law is actually an equivalence (modalities are closure operators, by the join law, so the type of modal propositions is the image of M up to logical equivalence).

```
Definition MProp := { P:Prop | M P → P }.
```

Despite not being a sort, MProp can be seen as a subtype of **Prop** and, therefore, as a universe in the sense of Section 1.1.

```
Definition El (P:MProp) : Prop := proj1_sig P.
```

Because of strength, the MProp universe is closed by products of arbitrary types. The **Program** keyword makes it possible to populate MProp by giving the proposition P (*first projection*) explicitly and discharging the proof that M P → P to tactics.

```
Program Definition Forall (A:Type) (F:A→MProp) : MProp :=
  forall x:A, El (F x).
Next Obligation.
  intros A F h x.
  apply strength with (x:=x) in h.
  destruct (F x). cbn in ∗.
  eauto.
Qed.
```

Definitions of U^- products, small and large, for MProp follow immediately:

```
Let Forall1 (u:MProp) (F:El u → MProp) : MProp := Forall (El u) F.
Let ForallU1 (F:MProp→MProp) : MProp := Forall MProp F.
```

Because El (Forall A F) = **forall** x:A, F, introduction, elimination and β-rules for the products are immediate.

Just like in Section 2.3, a retraction of MProp into a modal proposition can be used to trigger Hurkens's paradox. This is an example of instance of Hurkens's paradox where neither of the universes are sorts of the system.

```
Variable U0:MProp.
Variable proj0 : U0 → MProp.
Variable inj0 : MProp → U0.
Hypothesis inj0_unit : forall (A:MProp), El A → El (proj0 (inj0 A)).
Hypothesis inj0_counit : forall (A:MProp), El (proj0 (inj0 A)) → El A.
```

Following the the proof of Section 2.3, we conclude from this context that every modal proposition is inhabited. This is not necessarily a contradiction, as falsity need not be modal. For instance the trivial modality, whose only modal proposition in True.

```
Definition M (A:Prop) : Prop := True
```

A more interesting modality is, for a given X:

```
Definition M (A:Prop) : Prop := A∨X
```

for such a modality exhibiting a retraction into a modal proposition only prove ¬X: it is always the case that the smallest modal proposition is M False.

2.6 Weak excluded middle and proof irrelevance

In this section we will be concerned with the double-negation modality, whose modal propositions are also called *negative propositions*:

```
Definition M (A:Prop) : Prop := ¬¬A
```

and will use the paradox from Section 2.5, to prove that the weak principle of excluded middle

```
Hypothesis wem : forall A:Prop, ¬¬A ∨ ¬A.
```

entails a weak form of proof irrelevance. This is a new proof I added to theories/Logic/ClassicalFact.v and is available from version 8.5.

Looking closely at wem it becomes clear that it claims decidability of exactly the negative propositions.

```
Remark wem' : forall A:MProp, El A ∨ ¬El A.
```

The proof, therefore, proceeds just like the proof of Section 2.4. We begin by postulating a proposition with two proofs.

```
Variable U0:Prop.
Variables t f : U0.
Hypothesis not_eq_t_f : t ≠ f.
```

Notice that U0 is negative, since U0 has a proof, in particular ¬¬U0→U0 holds. So we only need to construct a retraction into U0. The retraction is given by inj0 and proj0 which are, *mutatis mutandis* the same as in Section 2.4: double negations have to be inserted for propositions which need to be negative, and proofs of negativity have to be provided when building negative propositions.

```
Let inj0 (A:MProp) : U0 :=
  or_ind (El A) (¬El A) U0 (fun _ ⇒ t) (fun _ ⇒ f) (wem' A).
Let proj0 (x:U0) : MProp :=
  exist (fun P⇒¬¬P → P) (¬¬(t = x)) (fun h x ⇒ h (fun k ⇒ k x)).
```

The unit and counit laws follow and we eventually derive a contradiction. That is, since U0:**Prop** is arbitrary a proof that:

```
forall (A:Prop) (x y:A), ¬¬(x=y)
```

Contrary to to the case of (strong) excluded middle, we cannot eliminate this last double-negative. So proof irrelevance doesn't follow from weak excluded middle. However, this section proves that weak excluded middle is incompatible with any sort of proof relevance principle. In particular, in Coq lingo, weak excluded middle cannot hold in impredicative Set, that is an impredicative sort with strong elimination.

3 Conclusion

The axiomatisation of Hurkens's paradox presented in Section 1 is very versatile. It can be used, mostly, to prove that some combination of logical

principles are incompatible, but also to detect bugs in a dependent-type-theory implementation. Which is a completely fair and healthy activity if you ask this author.

It is, certainly, an improvement over a situation where each paradox would need a careful redesign of Hurkens's proof to fit the specific premises. In practice it meant that paradoxes were not derived, because the brave paradox-finder didn't have the energy or expertise to translate Hurkens's paradox.

As per the axiomatisation itself. It has the pleasant property of requiring only a subset of U^- where the "proofs" of "propositions" don't require β-rules or any kind of equality rule. So something was learned. Adapting the proof to the axiomatisation doesn't present any new difficulty, except from controlling rewriting a little. It wasn't discovered before solely by the virtue of nobody looking. The reader who enjoyed this axiomatisation can celebrate the bout of optimism which made me look the right way, and the night I lost over it.

Bibliography

[1] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[2] Henk Barendregt. Lambda calculi with types. *Handbook of logic in computer science*, 1992.

[3] Thierry Coquand. An analysis of Girard's paradox. Technical report, 1986.

[4] Thierry Coquand. Metamathematical investigations of a calculus of constructions. Technical report, INRIA, 1989.

[5] Herman Geuvers. Inconsistency of classical logic in type theory. 2007.

[6] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État, Paris 7, 1972.

[7] Hugo Herbelin and Arnaud Spiwack. The Rooster and the Syntactic Bracket. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 169—-187, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] Antonius Hurkens. A simplification of Girard's paradox. *Typed Lambda Calculi and Applications*, pages 266–278, 1995.

[9] The Coq development team. The Coq Proof Assistant. http://coq.inria.fr/.