Notes on axiomatising Hurkens’s Paradox

Arnaud Spiwack

July 17, 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 is a very economic, though rather hard to understand, paradox of the U^- impredicative type theory, described in Section whose main characteristic is to feature 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 gave a sufficient condition, albeit much stronger, to derive contradictions in a generic way. His proof was based on Girard’s paradox rather than Hurkens’s one (which came out ten years later). Geuvers 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: 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 a 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* [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 ($\lambda$-abstraction) and elimination rule (application) – for each of the formation rule. For each pair $\lambda$-abstraction & application, there may be a $\beta$-equivalence rule, modelled as an equality; only the $\beta$-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**

\[
\text{Variable } U1 : Type. \\
\text{Variable } El1 : U1 → Type. \\
\]

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

\[
\text{Variable } \text{forall}1 : \forall u : U1, (El1 u → U1) → U1. \\
\text{Notation } "\forall!_1' x : A , B " := (\text{forall}1 A (fun x ⇒ B)). \\
\text{Notation } "A' → →_1' B" := (\text{forall}1 A (fun _ ⇒ B)). \\
\text{Variable } \text{lam}1 : \forall u B, (forall x:El1 u, El1 (B x)) → El1 (\forall!_1 x:u, B x). \\
\text{Notation } "\lambda!_1' x , u' := (\text{lam}1 _ _ (fun x ⇒ u)). \\
\text{Variable } \text{app}1 : \forall u B ((forall (\forall!_1 x:u, B x)) (x:El1 u), El1 (B x)). \\
\text{Notation } "f · _' _ := (\text{app}1 _ _ f x). \\
\text{Variable } \text{beta}1 : \forall u B (forall (\forall!_1 x:u, B x)) x, \\
(\lambda!_1 y, f y) ·_1 x = f x. \\
\]

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
The small universe $U_0$ is an element of the larger one. Therefore we need an $u_0:U_1$ and $U_0$ is taken to be $E_{11}u_0$ rather than a variable.

Small universe The small universe $U_0$ is closed by dependent products in $U_0$. The definitions are symmetric to the corresponding ones of $U_1$. Notice, however, the lack of $\beta$-rule, which is unnecessary to derive a contradiction.

The small universe $U_0$ is made impredicative by a dependent whose range is in $U_1$. Contrary to the impredicative product of $U_1$, the range cannot be restricted to be only $U_0$. Here again, the $\beta$-rule is not needed.

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 $U_0$ is inhabited. It will only be an actual contradiction if $U_0$ contains the empty type. For this purpose, let’s assume a type in $U_0$, we will then prove it is inhabited.
Variable \( F : U_0 \).

The proof will require simplifying \( \beta \)-redexes. We provide tactics to that effect.

\[
\text{Ltac simplify} := \quad \begin{array}{l}
\text{(repeat rewrite ?beta1, ?betaU1); lazy beta.}
\end{array}
\]

\[
\text{Ltac simplify_in h} := \quad \begin{array}{l}
\text{(repeat rewrite ?beta1, ?betaU1 in h); lazy beta in h.}
\end{array}
\]

These tactics are rather brute-force, in that they will \( \beta \)-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 \( U_1 \) as the type of datatypes and \( U_0 \) as the type of proposition. This view is justified by the fact that \( U_0 \) is not equipped with \( \beta \)-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):

\[
\begin{align*}
\text{Definition } V : U_1 & := \forall_2 A, ((A \to U_0) \to_1 A) \to_1 A \to_1 U_0. \\
\text{Definition } U : U_1 & := V \to_1 U_0. \\
\text{Definition } sb \ (z : \text{El1 } V) : \text{El1 } V & := \lambda_2 A, \lambda_1 r, \lambda_1 a, r \cdot_1 (z \cdot_1 [A] \cdot_1 r) \cdot_1 a. \\
\text{Definition } \text{le} \ ((i : \text{El1 } (U \to_1 U_0)) : U_0) & := x \cdot_1 (\lambda_2 A, \lambda_1 r, \lambda_1 a, i \cdot_1 (\lambda_1 v, (sb v) \cdot_1 [A] \cdot_1 r) \cdot_1 a)). \\
\text{Definition } \text{le'} : \text{El1 } ((U \to_1 U_0) \to_1 U \to_1 U_0) & := \lambda_1 i, \lambda_1 x, \text{le} i x. \\
\text{Definition } \text{induct} \ ((i : \text{El1 } (U \to_1 U_0)) : U_0) & := \forall_0^1 x : U, \text{le} i x \to_0 \text{le'} i x. \\
\text{Definition } \text{WF} : \text{El1 } U & := \lambda_1 z, (\text{induct} \ (z \cdot_1 [U] \cdot_1 \text{le'})). \\
\text{Definition } I \ (x : \text{El1 } U) : U_0 & := (\forall_0^1 x : U \to_1 U_0, \text{le} i x \to_0 I \cdot_1 (\lambda_1 v, (sb v) \cdot_1 [U] \cdot_1 \text{le'} \cdot_1 x)) \to_0 F.
\end{align*}
\]

The proofs follow Geuvers [5] as well. The main difference is that we must explicitly call to \textit{simplify} where conversion was used implicitly and that standard Coq tactics calls to the \textit{intro} and \textit{apply} tactics are generally replaced by tactics of the form \textit{refine} (\( \lambda_0 x \_ \)) and \textit{refine} (\( h_{0\_} \)) respectively.
Lemma Omega : El0 (∀i : u →₁ u₀, induct i →₀ i₁ WF).
Proof.
refine (λ₀¹ i, λ₀ y, _).
refine (y₀[1]₀...).
unfold le,WF,induct. simplify.
refine (λ₀¹ x, λ₀ h₀, _). simplify.
refine (y₀[1]₀...).
unfold le, simplify. unfold sb at 1. simplify. unfold le' at 1. simplify.
exact h₀.
Qed.

Lemma lemma1 : El0 (induct (λ₁ u, I u)).
Proof.
unfold induct.
refine (λ₀¹ x, λ₀ p, _). simplify.
refine (λ₀ q, _).
assert (El0 (I WF)) as h.
{ generalize (q₀[λ₁ u, I u]₀p). simplify.
  intros q'. exact q'.
}
refine (h₀[1]₀...).
refine (λ₀¹ l₀...).
refine (λ₀ h', _).
genralize (q₀[λ₁ y, i₁ (λ₁ v, (sb v)₁[U]_₁ le'₁ y)]). simplify.
intros q'.
refine (q'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 ((∀₀¹ i : u →₁ u₀, induct i →₀ i₁ WF →₀ F)).
Proof.
refine (λ₀¹ x, _).
assert (El0 (I WF)) as h.
{ generalize (x₀[λ₁ u, I u]₀[lemma1]). simplify.
  intros q.
  exact q.
}
refine (h₀[1]₀...). clear h.
refine (λ₀¹ i, λ₀ h₀, _).
genralize (x₀[λ₁ y, i₁ (λ₁ v, (sb v)₁[U]_₁ le'₁ y)]). simplify.
intros q.
refine (q₀...). 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_i, Omega).
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 \( \lambda \)-abstraction, application and \( \beta \)-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 $U' := \text{Type}$.
Let $U : U'$ := \text{Type}.

Variable $\text{proj} : U' \to U$.

With the following laws.

**Hypothesis** $\text{proj}_\text{unit} : \forall (A : U'), A \to \text{proj} A$.

**Hypothesis** $\text{proj}_\text{counit} : \forall (F : U \to U), \text{proj} (\forall (A : U), F A) \to (\forall (A : U), F A)$.

**Hypothesis** $\text{proj}_\text{coherent} : \forall (F : U \to U) (f : \forall (x : U), F x) (x : U),
\text{proj}_\text{counit} \_ (\text{proj}_\text{unit} \_ f) x = f x$.

The $\text{proj}_\text{unit}$ law expresses that if $\text{proj}$ generally diminishes the ability to distinguish between elements of $A : U'$, it does not lose elements. We don’t have a way back from $\text{proj} A$ to $A$ in general, but $\text{proj}$ forms a monad. The $\text{proj}_\text{unit}$ law expresses a small variation on this latter remark.

These properties are sufficient to show that $U$ is closed by large product. The $\beta$-rule, omitted, is easily derived from $\text{proj}_\text{coherent}$.

Let $\text{forall}_U (F : U \to U) : U := \text{proj} (\forall (A : U), F A)$.
Let $\text{lam}_U F (f : \forall (A : U), F A) : \text{proj} (\forall (A : U), F A) := \text{proj}_\text{unit} \_ f$
Let $\text{app}_U F (f : \forall (A : U), F A) (A : U) : F A := \text{proj}_\text{counit} \_ f x$.

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 $\text{Type}$ represents an arbitrary type, including the impredicative sort $\text{Prop}$, which is indeed impredicative in the above sense.

### 2.3 Generalising Geuvers’s proof

Geuvers [5] proves that an impredicative sort $U_1$ cannot be a retract of an $U_0 : U_1$. His proof is made for $U_1 = \text{Prop}$, 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 $U_2 := \text{Type}$.
Let $U_1 : U_2 := \text{Type}$.

Variable $U_0 : U_1$.

Where $U_1$ is impredicative over $U_2$ as in Section 2.2. The retraction is given by the following functions. Only a weak form of retraction is needed were types in $U_1$ which are “logically equivalent” are considered equal.

**Variable** $\text{proj}_0 : U_0 \to U_1$.
**Variable** $\text{inj}_0 : U_1 \to U_0$.

**Hypothesis** $\text{inj}_0\_\text{unit} : \forall (b : U_1), b \to \text{proj}_0 (\text{inj}_0 b)$.

**Hypothesis** $\text{inj}_0\_\text{counit} : \forall (b : U_1), \text{proj}_0 (\text{inj}_0 b) \to b$.

From this (weak) retraction we can define $E_0$ and corresponding products for $U_0$ despite the fact that $U_0$ is not necessarily a sort.
Where or_ind
Prop we can reflect
excluded middle, in an impredicative sort makes it proof irrelevant,
new proof of said result. With the characterisation of Section
distribution in the file
theories/Logic/ClassicalFact.v
type in that sort have at most one element. This proof appears in the Coq
Hugo Herbelin. It uses Geuvers result and was mostly unmodified with the
the impredicative sort
Prop be done in an arbitrary impredicative sort, but the Coq
proof is done only for
Geuvers proof, from Section

2
2
Prop

or_ind A : Prop, A V ¬A.

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
2
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 V B → P is the
elimination principle of disjunction.

2.4 Excluded middle and proof irrelevance

Geuvers proof, from Section

2
2
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 V ¬A.

Let inj0 (u : U0) := inj0 u
Let forall0 (u : U0) (B : inj0 u → U0) : inj0 (forall x inj0 u, inj0 (B x))
Let Lambda0 u B (f : forall x inj0 u, inj0 (B x))

: inj0 (forall x inj0 u, inj0 (B x)) := inj0_unit f.
Let app0 forall0 u B (f : inj0 (forall x inj0 u, inj0 (B x)))) (x inj0 u)

: inj0 (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, inj0 (B x))
Let Lambda0 u B (f : forall x inj0 u, inj0 (B x))

: inj0 (forall x inj0 u, inj0 (B x)) := inj0_unit f.
Let app0 forall0 u B (f : inj0 (forall x inj0 u, inj0 (B x)))) (x inj0 u)

: inj0 (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 inj0 P = inj0 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. inj0 (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.

\(\text{Let } \text{inj0 } (u : U0) := \text{proj0 } u \text{ \text{Prop}}\)
\(\text{Let } \forall u (u : U0) (B : \forall u \rightarrow U0) : U0 := \text{inj0 } (\forall x \forall u, \forall u (B x))\)
\(\text{Let } \lambda u B (f : \forall x \forall u, \forall u (B x)) : \forall u (\forall x \forall u, \forall u (B x)) := \text{inj0_unit } f.\)
\(\text{Let } \text{app0 } \forall u B (f : \forall x \forall u (\forall x \forall u (B x)))) (x \forall u) : \forall u B x := \text{inj0_counit } f x\)

\(\text{From this, the paradox is set up, so we can deduce that every proposition of } P : U0 \text{ is “inhabited” in that } \text{inj0 } P = \text{inj0 } P \text{ is inhabited, and therefore, that every proposition of } F : U1 \text{ is inhabited since } \text{inj0 } F : U0 \text{ is “inhabited” in the sense of } U0, \text{i.e. } \text{inj0 } (\text{inj0 } F) \text{ is inhabited, then } \text{inj0_counit } \text{concludes.}\)

\(\text{Since } \text{Prop} \text{ is an instance of the signature of Section } 2.2 \text{ we prove, like Geuvers, that } \text{Prop} \text{ is not a retract of a proposition } P : \text{Prop}.\)

2.4 Excluded middle and proof irrelevance

Geuvers proof, from Section

2
2
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 V ¬A.


We are left to prove the unit and counit laws of inj\(_0\) and proj\(_0\) to satisfy the premisses of the paradox in Section 2.3. The unit law is direct:

\textbf{Lemma} inj\(_0\)\textunderscore unit : \forall A : \text{Prop}, A \to \text{proj}_0 (\text{inj}_0 A).
\textbf{Proof.}
intros A x. unfold proj\(_0\), inj\(_0\).
destruct (em A) as [h h].
+ reflexivity.
+ contradiction.
\textbf{Qed.}

The counit law is the step that makes a crucial use of the not\_eq\_t\_f hypothesis:

\textbf{Lemma} inj\(_0\)\textunderscore counit : \forall A : \text{Prop}, \text{proj}_0 (\text{inj}_0 A) \to A.
\textbf{Proof.}
intros A h. unfold proj\(_0\), inj\(_0\) in \_.
destruct (em A) as [l l].
+ apply l.
+ absurd (t=f).
  * apply not\_eq\_t\_f.
  * apply h.
\textbf{Qed.}

Section 2.3 then yields a contradiction. Since \(U_0\) is arbitrary we have:\n\textbf{forall} (A : \text{Prop}) (x y : A), \neg \neg (x=y). A last application of the excluded middle yields the expected result:

\textbf{forall} (A : \text{Prop}) (x y : A), x=y

### 2.5 Variants of Prop

A (monadic) modality on \textit{Prop} is given by a mapping:

\textit{Variable} \(M : \text{Prop} \to \text{Prop}\).

Together with the following laws:

\textbf{Hypothesis} unit : \forall A : \text{Prop}, A \to M A.
\textbf{Hypothesis} join : \forall A : \text{Prop}, M (M A) \to M A.
\textbf{Hypothesis} incr : \forall A B : \text{Prop}, (A \to B) \to M A \to M B.

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

\textbf{Lemma} strength : \forall (P : A \to \text{Prop}), M(\forall x : A, P x) \to \forall x : A, M (P x).
\textbf{Proof.}
eauto.
\textbf{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** $M\text{Prop} := \{P : \text{Prop} \mid M P \rightarrow P\}$.

Despite not being a sort, $M\text{Prop}$ can be seen as a subtype of $\text{Prop}$ and, therefore, as a universe in the sense of Section 1.1.

**Definition** $\text{El}(P : M\text{Prop}) : \text{Prop} := \text{proj1} \_\text{sig} P$.

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

*Program Definition* Forall $(A : \text{Type})\ (F : A \rightarrow M\text{Prop}) : M\text{Prop} :=$

$\forall x : A, \text{El} (F x)$.

**Next Obligation.**

*intros* A F h x.

*apply* strength with ($x := x$) in h.

*destruct* ($F x$). *cbn* in $\ast$.

*eauto*.

Qed.

Definitions of $U^-$ products, small and large, for $M\text{Prop}$ follow immediately:

Let Forall1 $(u : M\text{Prop})\ (F : \text{El} u \rightarrow M\text{Prop}) : M\text{Prop} :=$ Forall ($\text{El} u$) F.

Let ForallU1 $(F : M\text{Prop} \rightarrow M\text{Prop}) : M\text{Prop} :=$ Forall $M\text{Prop} F$.

Because $\text{El} (\text{Forall} A F) = \forall x : A, F$, introduction, elimination and $\beta$-rules for the products are immediate.

Just like in Section 2.3, a retraction of $M\text{Prop}$ 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** $U0 : M\text{Prop}$.

**Variable** $\text{proj0} : U0 \rightarrow M\text{Prop}$.

**Variable** $\text{inj0} : M\text{Prop} \rightarrow U0$.

**Hypothesis** inj0\_unit : forall $(A : M\text{Prop})\ (\text{El} A \rightarrow \text{El} (\text{proj0} (\text{inj0} A)))$.

**Hypothesis** inj0\_counit : forall $(A : M\text{Prop})\ (\text{El} (\text{proj0} (\text{inj0} A)) \rightarrow \text{El} A)$.

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** $M (A : \text{Prop}) : \text{Prop} := \text{True}$.

A more interesting modality is, for a given $X$:

**Definition** $M (A : \text{Prop}) : \text{Prop} := A \lor X$

for such a modality exhibiting a retraction into a modal proposition only prove $\neg X$: it is always the case that the smallest modal proposition is $M \text{False}$.
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** \( M(A:Prop) : Prop \equiv \neg\neg A \)

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

**Hypothesis** \( \text{wem} : \forall A : \text{Prop}, \neg\neg A \lor \neg A. \)

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 \( \text{wem} \) it becomes clear that it claims decidability of exactly the negative propositions.

**Remark** \( \text{wem}’ : \forall A : \text{MProp}, \text{El} A \lor \neg\text{El} A. \)

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

**Variable** \( U_0 : \text{Prop}. \)

**Variables** \( t, f : U_0. \)

**Hypothesis** \( \text{not_eq_t_f} : t \neq f. \)

Notice that \( U_0 \) is negative, since \( U_0 \) has a proof, in particular \( \neg\neg U_0 \rightarrow U_0 \) holds. So we only need to construct a retraction into \( U_0 \). The retraction is given by \( \text{inj}_0 \) and \( \text{proj}_0 \) 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.

\[
\text{Let inj}_0 (A : \text{MProp}) : U_0 := \text{or_ind (El A) (\neg\neg A) U_0 (fun _ \Rightarrow t) (fun _ \Rightarrow f) (wem’ A)}.
\]

\[
\text{Let proj}_0 (x : U_0) : \text{MProp} := \text{exist (fun P \Rightarrow \neg\neg P \rightarrow P) (\neg\neg (t = x)) (fun h x \Rightarrow h (fun k \Rightarrow k x))}.
\]

The unit and counit laws follow and we eventually derive a contradiction. That is, since \( U_0 : \text{Prop} \) is arbitrary a proof that:

\[
\text{forall (A : Prop) (x y : A), \neg\neg(x = y)}
\]

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 11 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 $\beta$-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/.