The Consistency of Large Elimination

Robin Adams

September 17, 2004

In the current implementation of Plastic, there are three ways in which inductive types may be built:

1. As objects of type **Type**. For example, the command
   `Inductive [N : Type]`
   `Constructors [zero : N]`
   `[succ : (x : N) N]`
   produces the declarations
   `[N : Type]`
   `[zero : N]`
   `[succ : N -> N]`
   `[E_N : (C : N -> Type)C zero -> ((n : N) C n -> C (succ n)) ->`
   `(n : N) C n]`
   along with the computation rules

$$\begin{aligned} \texttt{E\_N C a f zero} &\to \texttt{a} \\ \texttt{E\_N C a f (succ n)} &\to \texttt{f n} \end{aligned}$$

2. As objects of type **Prop**[1]. For example, the command
   `Inductive [le : (x,y:El N)El Prop]`
   `Relation`
   `Constructors [leref : (n:El N) Prf (le n n)]`
   `[lesucc : (m,n:El N)(H : Prf (le m n))Prf (le m (succ n))]`
   produces the declarations
   `[le : N -> N -> Prop]`
   `[leref : (n:N) Prf (le n n)]`
   `[lesucc : (m,n:N) Prf (le m n) -> Prf (le m (succ n))]`
   `[E_le : (C : N -> N -> Prop) ((n:N) Prf (C n n)) -> ((m,n:N) Prf`
   `(C m n) -> Prf (C m (succ n))) -> (m,n:N) Prf (le m n) -> Prf`
   `(C m n)]` together with the computation rules

$$\begin{aligned} \texttt{E\_le C H H}' \texttt{ n n (leref n)} &\to \texttt{H n} \\ \texttt{E\_le C H H}' \texttt{ m (succ n)(lesucc m n K)} &\to \texttt{H}' \texttt{ m n (E\_le C H H}' \texttt{ m n K)} \end{aligned}$$

[1] We assume that **Prop** : **Type** and Prf : **Prop** → **Type** have already been declared.

(Thus, we define the less-than-or-equal-to relation on N to be the least binary relation on N such that $n \leq n$ and if $m \leq n$ then $m \leq n+1$.)

3. An object of type **Prop** with large elimination. For example, the command
```
Inductive [le :  (x,y:El N)El Prop]
Relation_LE
Constructors [leref :  (n:El N) Prf (le n n)]
[lesucc :  (m,n:El N)(H : Prf (le m n))Prf (le m (succ n))]
```
produces the declarations
```
[le :  N -> N -> Prop]
[leref :  (n:N) Prf (le n n)]
[lesucc :  (m,n:N) Prf (le m n) -> Prf (le m (succ n))]
[E_le :  (C : N -> N -> Type) ((n:N) C n n) -> ((m,n:N) C m n ->
C m (succ n)) -> (m,n:N) Prf (le m n) -> C m n)]
```
together with the computation rules

$$\begin{aligned} \texttt{E\_le C H H'\ n n (leref n)} &\rightarrow \texttt{H n} \\ \texttt{E\_le C H H'\ m (succ n)(lesucc m n K)} &\rightarrow \texttt{H' m n (E\_le C H H' m n K)} \end{aligned}$$

(The only difference between this and 2 is that **Type** appears in the kind of *E_le* rather than **Prop**, and correspondingly some occurrences of Prf disappear.)

Using `Relation_LE`, we are able to inductively define relations in such a way that we can not only prove propositions by induction over the relation, but also define objects of any type in **Type** by recursion; for example, we could define an object $f(m,n)$ for all natural numbers m, n such that $m \leq n$, by recursion over the proof that $m \leq n$. Using the definition of `le` given above, this is equivalent to recursion over the difference $n - m$.

In the Plastic documentation, it is asked whether inconsistencies can be produced using `Relation_LE`.

Free use of `Relation_LE` certainly can produce inconsistencies, as we shall show.

1 Girard's Paradox

In "Inconsistency of Classical Logic in Type Theory", Geuvers presents the following Lego script, written by Randy Pollack, building Girard's paradox within Lego with a universe `Type` which is its own type:

```
[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 script can easily be turned into a Plastic script. The file "girard2.lf" is more or less a line-by-line translation of this script, with the following changes:

- **Prop** replaces **Type** throughout.
- Whenever the Lego script builds a Π-type in **Type** by quantifying over **Type**, we build an inductive relation in **Prop** with `Relation_LE`.

The script uses `Relation_LE` twice, as well as `Relation` twice. Each of these latter two relations could have been defined using UTT's ∀ constructor, as well as the ability to form Π-types in **Type**. Thus, we have

Theorem 1 *The theory formed by adding the following two types to UTT is inconsistent:*

```
Inductive [V : El Prop]
Relation_LE
Constructors [VI : (Z : (A : El Prop)(R : (P : (x :  El (Prf A)) El
Prop) (x :  El (Prf A)) El Prop) (a :  El (Prf A)) El Prop) Prf V];

Inductive [U : El Prop]
Relation_LE
Constructors [UI : (F : (Z : El (Prf V)) El Prop) Prf U];
```

2 Inconsistent Relations

We are also able to prove several single uses of `Relation_LE` inconsistent, as follows:

Theorem 2 *The system formed by adding the following type to UTT is inconsistent:* `Inductive [A : Type][B : (x : El A) Type][pi : El Prop] Relation_LE Constructors [lam : (b : (x : El A) El (B x)) Prf Pi]`

Proof Given these Π-types, we can build the two types in Theorem 1 as follows:

$$\texttt{V} \equiv \Pi A : \mathbf{Prop}.\Pi R : \mathrm{Prf}(\Pi P : \mathrm{Prf}(\Pi x : \mathrm{Prf}\, A.\mathbf{Prop}).\, \mathrm{Prf}(\Pi x : A.\mathbf{Prop})).\Pi a : \mathrm{Prf}\, A.\mathbf{Prop}$$

$$\texttt{U} \equiv \Pi Z : \mathrm{Prf}\, V.\mathbf{Prop}$$

Theorem 3 *The system formed by adding the following type to UTT is inconsistent:* `Inductive [propinprop : El Prop] Relation_LE Constructors [down : (P : El Prop) propinprop]`

(We are essentially building a copy of **Prop** inside **Prop**, giving us all the power of the inconsistent axiom $* : *$.)

Proof Given `propinprop`, we can easily define

$$\texttt{up} : \mathrm{Prf}\,\texttt{propinprop} \to \mathbf{Prop}$$

thus:

$$\texttt{up} \equiv \texttt{E\_propinprop}\mathbf{Prop}([P : \texttt{El}\mathbf{Prop}]P)$$

We then have

$$\texttt{up}\ \texttt{down}P \to P \qquad (P : \mathbf{Prop})$$

We can now define the types of Theorem 1 thus:

$$\texttt{V} \equiv \forall A : \mathbf{Prop}.((A \Rightarrow \texttt{propinprop}) \Rightarrow (A \Rightarrow \texttt{propinprop})) \Rightarrow \forall a : \mathrm{Prf}\, A.\texttt{propinprop}$$

$$\texttt{U} \equiv \texttt{V} \Rightarrow \texttt{propinprop}$$

This is essentially the same as was done in the paper "Inconsistency of Classical Logic ...", where a copy of **Prop** was built within Set, over which large elimination is possible.

Theorem 4 *The system formed by adding the following type to UTT is inconsistent:* `Inductive [A : Type][P : (x : El A) El Prop][Ex : El Prop] Relation_LE Constructors [ExI : (a : El A)(x : El (Prf (P a))) Prf Ex]`

Proof Take any propositon $\top$: **Prop** for which there is a proof H : Prf $\top$. We can define `propinprop` in Theorem 3 thus:

$$\begin{aligned} \texttt{propinprop} &\equiv \texttt{Ex}\mathbf{Prop}[p : \mathbf{Prop}]\top \\ \texttt{down} &\equiv [P : \mathbf{Prop}]\texttt{ExI}PH \\ \texttt{up} &\equiv \texttt{E\_Ex}\mathbf{Prop}([p : \mathbf{Prop}]\top)\mathbf{Prop}([p : \mathbf{Prop}][x : \text{Prf } \top]p) \end{aligned}$$

It is an interesting question which uses of `Relation_LE` are consistent and which are not. I strongly suspect that the definition of the empty proposition (that with no constructors) is consistent, for it declares no new reduction rules.

I have been looking into extending Goguen's Typed Operational Semantics (TOS) to the types built with `Relation_LE` to see whether any such types can be proven consistent that way. This has led me to consider the definition of TOS for UTT as built within a weak (lambda-free) logical framework, as opposed to LF. I believe this work may well prove valuable in its own right.