Paradoxical Typecase

Stephanie Weirich

University of Pennsylvania

What this talk is not about

- Dependently typed Haskell
  - Equalities between kinds (k1 ~ k2)
  - *:*
  - ∏-type

- Trellys
  - Foundations for new dependently-typed language
  - Mix "logical" language with "computation" language
  - $\Gamma \vdash^{\theta} e : t$
  - Type "t @ θ" integrates values between languages

A Paradox

Type injectivity is necessary for preservation, but leads to inconsistency

What is wrong with injectivity?

- *Data type injectivity*

  List t1 = List t2 implies t1 = t2

- *Universal type injectivity*

  ∀a:*. t1 = ∀a:*.t2 implies that for all t, t1{t/a} = t2{t/a}

- *Function type (codomain) injectivity*

  ∏x:t1. t2 = ∏x:t1. t2' implies that for all e1, t2{ e/x} = t2'{e/x}

- *Data constructor injectivity*

  Just e = Just e' implies e = e'

Only one available in Coq and Agda

Type injectivity is important

- Inversion in the presence of type conversion:

  If $\Gamma \vdash \lambda x.\ e : t$ then there is some t1, t2, such that
  $\Gamma, x{:}t1 \vdash e : t2$ where $\Gamma \vdash t = \prod x{:}t1.t2 : *$

- Need injectivity for preservation:
  - Say $(\lambda x.\ e)\ e' \ \text{->}\ e\{ e/ x\}$ and $\Gamma \vdash (\lambda x.\ e)\ e' : t2\ \{e'/x\}$
  - Know $\Gamma \vdash \lambda x.\ e : \prod x{:}t1.t2$, and $\Gamma \vdash e' : t1$
  - Want to prove $\Gamma, x{:}t1 \vdash e : t2$, to use substitution.
  - Inversion gives
    $\Gamma, x{:}t1' \vdash e : t2'$ where $\Gamma \vdash \prod x{:}t1.t2 = \prod x{:}t1'.t2' : *$
  - Injectivity gives $\Gamma \vdash t1 = t1' : *$ and $\Gamma, x{:}t1' \vdash t2 = t2' : *$ to finish the case.

Dire warnings

- From Agda manual:

Automatic injectivity of type constructors has been disabled (by default). To enable it, use the flag –injective-type-constructors, either on the command line or in an OPTIONS pragma. Note that this flag makes Agda anti-classical and ***possibly inconsistent****:*

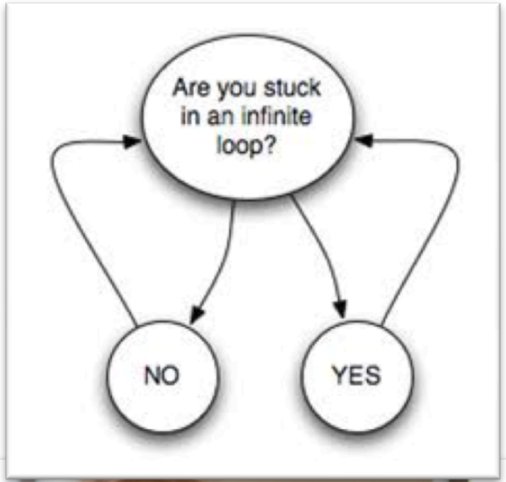
Agda with excluded middle is inconsistent

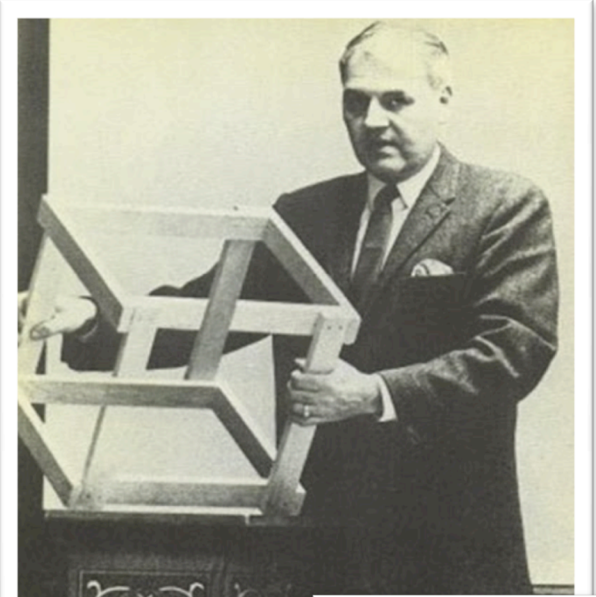
http://thread.gmane.org/gmane.comp.lang.agda/1367

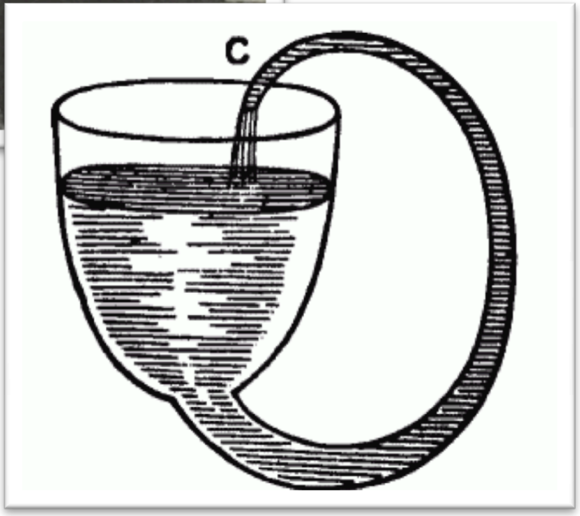
- From Coq FAQ:

...Injectivity of constructors is restricted to predicative types. If injectivity on large inductive types were not restricted, we would be allowed to derive an inconsistency (e.g. following the lines of Burali-Forti paradox). The ***question remains open whether injectivity is consistent*** *on some large inductive types not expressive enough to encode known paradoxes (such as type I above)....*

Logical Paradoxes







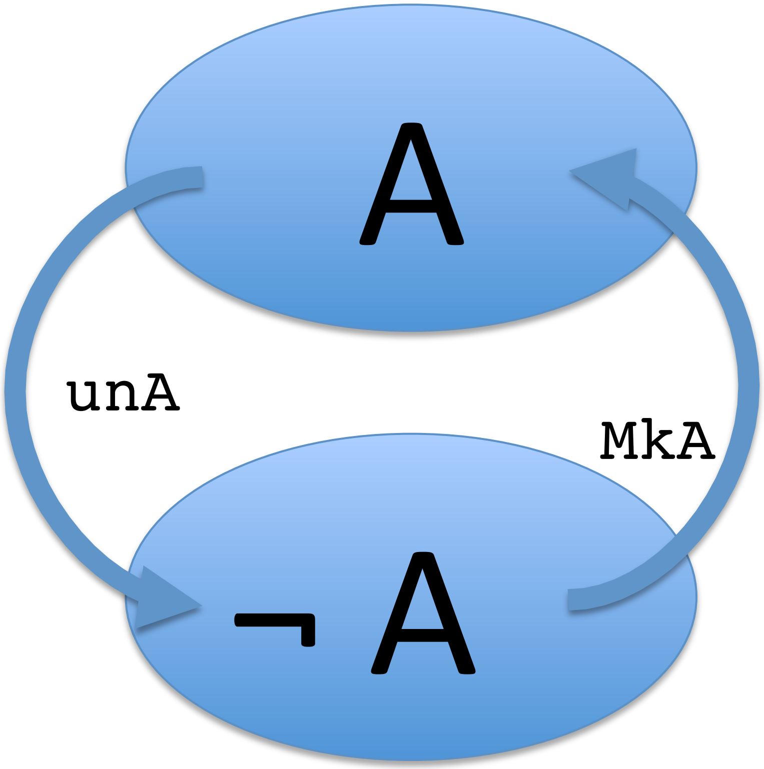
A logical paradox

$$A \cong \neg A$$

`(λx -> x x) (λx -> x x)`

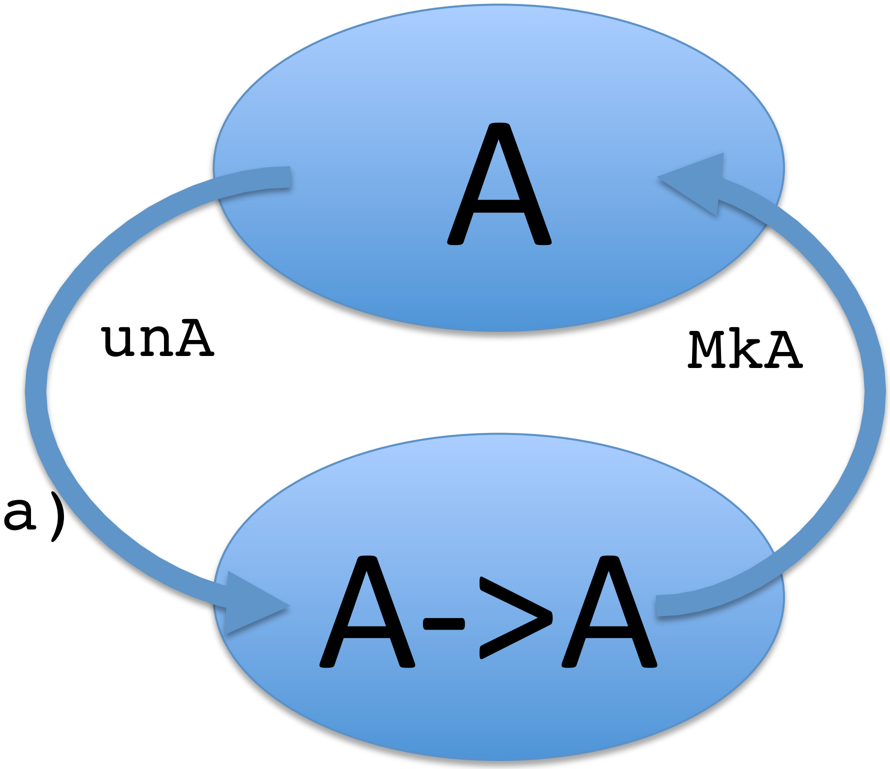
A ≅ ¬ A in Haskell

```
data Void -- uninhabited type
data A = MkA { unA :: A -> Void }

delta :: A -> A
delta x = (unA x) x

omega :: Void
omega = delta (MkA delta)
```



A ≅ A -> A

```
data Void
data A = MkA { unA :: A -> A }

delta :: A -> A
delta x = (unA x) x

omega :: A
omega = delta (MkA delta)
```



Easy (?) to avoid

(Strictly) positivity recursive types…

…but, what about recursive kinds?

```
data T :: (* -> Void) -> *
```

- `T` goes between `(* -> Void) and *`
- A `typecase` goes between * and `(* -> Void)`
- Not just `T`:
  - `∀:(* -> *) -> *`
  - `∏:(* -> *) -> *`
  - `∑:(* -> *) -> *`

In Haskell type language?!

```
{-# LANGUAGE DataKinds, KindSignatures,
  TypeFamilies #-}


data Void
data T (c :: * -> Void)


type family Delta (t :: *) :: Void
type instance Delta (T c) = c (T c)
```

~~`type Omega = Delta (T Delta)`~~

Last line doesn’t quite typecheck, whew!!

Expression level loop

```
data Void
data T (c :: * -> Void)
```

```
data R (t :: *) = MkR { unR :: t -> Void }

delta :: R R -> Void
delta x = unR x x

omega :: Void
omega = delta (MkR delta)
```

Doesn’t quite typecheck, whew!!

Expression level loop

```
data Void
data T (c :: * -> Void)
```

```
data R (t :: *) = MkR { unR :: t -> Void }

delta :: R (T R) -> Void
delta x = unR x x

omega :: Void
omega = delta (MkR delta)
```

Doesn’t quite typecheck,
need R (T R) ~ T R

Type families

```
data Void
data T (c :: * -> *)

type family Delta (t :: *) :: *
type instance Delta (T c) = c (T c)

data R (t :: *) = MkR { unR :: Delta t -> Void }

delta :: R (T R) -> Void
delta x = unR x x

omega :: Void
omega = delta (MkR delta)
```

Example from Oleg Kiselyov

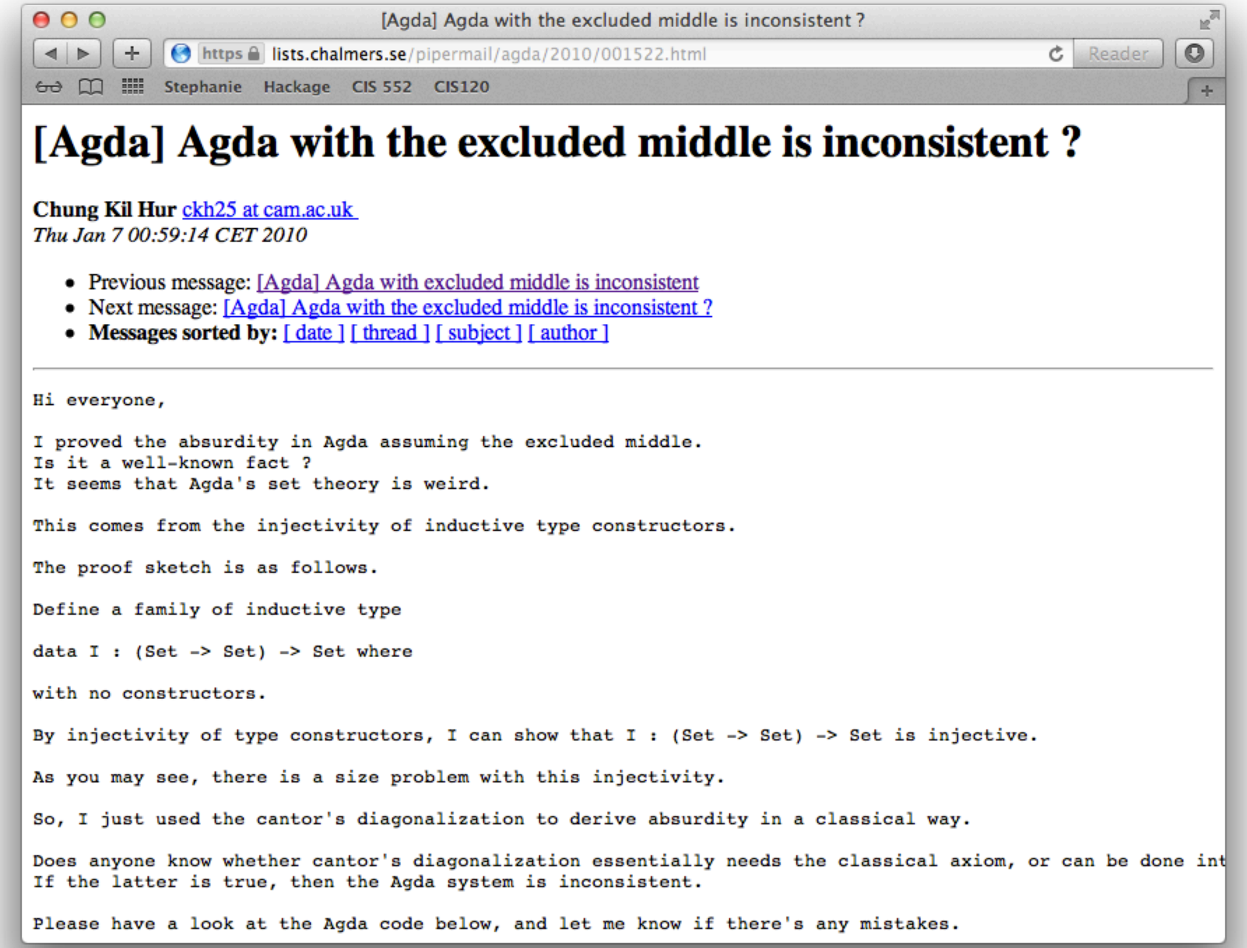
Can we just eliminate typecase?

typecase = GADTs + injectivity

```
data Void
data T (c :: * -> *)

~~type family Delta (t :: *) :: *~~
~~type instance Delta (T c) = c (T c)~~

data R (t :: *) =
    forall c:*->*. (t ~ T c) => MkR ( c (T c) -> Void )
unR :: R (T c) -> c (T c) -> Void
unR (MkR x) = x

delta :: R (T R) -> Void
delta x = unR x x

omega :: Void
omega = delta (MkR delta)
```

Need injectivity here
x :: (T c ~ T c’) => c’ (T c’) -> Void
Coerce to:: c (T c) -> Void

typecase = LEM + injectivity



[Agda] Agda with the excluded middle is inconsistent ?

Chung Kil Hur ckh25 at cam.ac.uk
Thu Jan 7 00:59:14 CET 2010

- Previous message: [Agda] Agda with excluded middle is inconsistent
- Next message: [Agda] Agda with the excluded middle is inconsistent ?
- **Messages sorted by:** [date] [thread] [subject] [author]

Hi everyone,

I proved the absurdity in Agda assuming the excluded middle.
Is it a well-known fact ?
It seems that Agda's set theory is weird.

This comes from the injectivity of inductive type constructors.

The proof sketch is as follows.

Define a family of inductive type

data I : (Set -> Set) -> Set where

with no constructors.

By injectivity of type constructors, I can show that I : (Set -> Set) -> Set is injective.

As you may see, there is a size problem with this injectivity.

So, I just used the cantor's diagonalization to derive absurdity in a classical way.

Does anyone know whether cantor's diagonalization essentially needs the classical axiom, or can be done int
If the latter is true, then the Agda system is inconsistent.

Please have a look at the Agda code below, and let me know if there's any mistakes.

Agda example

```
postulate exmid : ∀ (A : Set1) -> A + (A -> Void)
postulate Iinj :∀ x y -> I y ≡ I x -> y ≡ x

J : Set -> (Set -> Set)
J a with exmid (∑ x:Set. I x ≡ a)
  J a | inl (x, _) = x
  J a | inr b = λ x → Void

IJIeqI : ∀ x -> I (J (I x)) ≡ I x
IJIeqI = …

J_srj : ∀ (x : Set -> Set) -> ∑ a:Set. x ≡ J a
J_srj x = (I x, pf) where
     pf : x ≡ J (I x)
     pf = Iinj IJIeqI
```

```
J a = tcase a of
   (I b) -> b
   _     ->
     λ x -> Void
```

Essence of Agda paradox

```
J :: * -> (* -> *)
J a = tcase a of
       (I b) -> b
       _     -> λ x -> Void
C :: * -> *
C a = tcase (J a a) of
        Void -> Unit
        _    -> Void
```

Observe: `J (I C) (I C) => C (I C) =>`

`=> J (I C) (I C)`

What next?

- Disallow typecase?
  - Mendler-style eliminator for types?
- Disallow LEM (and equality?)
  - Nice to be compatible with classical reasoning
  - Propositional equality core component of dependent types
- Disallow injectivity? For quantified types *and* datatypes?
  - Current strategy by Agda & Coq
  - Sometimes useful in user code, but not often
  - …but seems strange given necessity for preservation
- Find a weaker statement of injectivity? LEM?
- Predicativity?
  - ∏ : (Set0 -> Set0) -> Set1
  - data I : (Set0 -> Set0) -> Set0
  - f ::Vec a n == Vec b n -> a == b
  - f:: iso a = head (iso (cons a nil))