Paradoxical Typecase

Stephanie Weirich
University of Pennsylvania
What this talk is not about

• Dependently typed Haskell
  – Equalities between kinds (k1 ~ k2)
  – *:*
  – Π-type

• Trellys
  – Foundations for new dependently-typed language
  – Mix “logical” language with “computation” language
  – \( \Gamma \vdash^\theta e : t \)
  – Type “t @ \theta” 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 \( t_1 = \text{List} \ t_2 \) implies \( t_1 = t_2 \)

• *Universal type injectivity*
  
  \( \forall \ a:*. \ t_1 = \forall \ a:*. \ t_2 \) implies that for all \( t \), \( t_1{\{t/a\}} = t_2{\{t/a\}} \)

• *Function type (codomain) injectivity*
  
  \( \prod x: t_1. \ t_2 = \prod x: t_1. \ t_2' \) implies that for all \( e_1 \), \( t_2{\{e/x\}} = t_2'{\{e/x\}} \)

• *Data constructor injectivity*  
  
  Just \( e = \text{Just} \ e' \) 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 $t_1$, $t_2$, such that
  $\Gamma, x:t_1 \vdash e : t_2$ where $\Gamma \vdash t = \prod x:t_1.t_2 : *$

• Need injectivity for preservation:
  – Say $(\lambda x. e) e' \rightarrow e \{e/x\}$ and $\Gamma \vdash (\lambda x. e) e' : t_2 \{e'/x\}$
  – Know $\Gamma \vdash \lambda x. e : \prod x:t_1.t_2$, and $\Gamma \vdash e' : t_1$
  – Want to prove $\Gamma, x:t_1 \vdash e : t_2$, to use substitution.
  – Inversion gives
    $\Gamma, x:t_1' \vdash e : t_2'$ where $\Gamma \vdash \prod x:t_1.t_2 = \prod x:t_1'.t_2' : *$
  – Injectivity gives $\Gamma \vdash t_1 = t_1' : *$ and $\Gamma, x:t_1' \vdash t_2 = t_2' : *$ to finish the case.
• 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

- Are you stuck in an infinite loop?
  - NO
  - YES

- Image of a paradoxical object

- Image of a woman shaving a man

- Image of a water-filled glass with a hole in the bottom
A logical paradox

\[ A \equiv \neg A \]

\((\lambda x \rightarrow x \ x) \ (\lambda x \rightarrow x \ x)\)
$A \cong \neg A$ 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)
\[
A \cong A \rightarrow A
\]

```haskell
data Void

data A = MkA { unA :: A \rightarrow A }

delta :: A \rightarrow A
delta x = (unA x) x

omega :: A
omega = delta (MkA delta)
```
(Strictly) positivity recursive types...

...but, what about recursive kinds?

data T :: (* \to Void) \to *

- T goes between (* \to Void) and *
- A typecase goes between * and (* \to Void)
- Not just T:
  - \forall : (* \to *) \to *
  - \prod : (* \to *) \to *
  - \sum : (* \to *) \to *
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)
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!!
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 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
Essence of Agda paradox

\[ J : : \ast \to (\ast \to \ast) \]

\[ J \ a = \text{tcase} \ a \ \text{of} \]
\[ \quad (I \ b) \to b \]
\[ \quad _ \to \lambda \ x \to \text{Void} \]

\[ C : : \ast \to \ast \]

\[ C \ a = \text{tcase} \ (J \ a \ a) \ \text{of} \]
\[ \quad \text{Void} \to \text{Unit} \]
\[ \quad _ \to \text{Void} \]

Observe: \[ J \ (I \ C) \ (I \ C) \Rightarrow C \ (I \ C) \Rightarrow \]
\[ \Rightarrow J \ (I \ C) \ (I \ C) \]
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?
  – \( \Pi : (\text{Set0} -> \text{Set0}) -> \text{Set1} \)
  – data I : (\text{Set0} -> \text{Set0}) -> \text{Set0}
  – \( f :: \text{Vec a n} == \text{Vec b n} -> a == b \)
  – \( f:: \text{iso a = head (iso (cons a nil))} \)