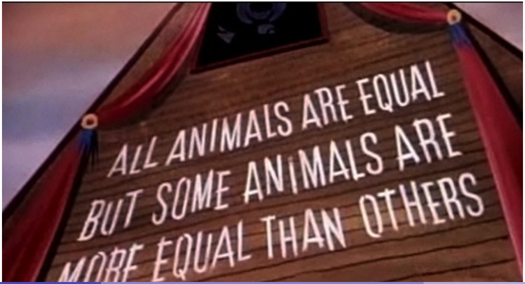
A Short History of Equality

Thorsten Altenkirch

School of Computer Science
University of Nottingham

June 25, 2011



Agda is cool!

```
data Vec (A : Set) : ℕ → Set where
  []    : Vec A zero
  _::_  : {n : ℕ} → A → Vec A n → Vec A (suc n)

data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)

_!!_ : ∀ {A n} → Vec A n → Fin n → A
[]        !! ()
(x :: xs) !! zero    = x
(x :: xs) !! (suc i) = xs !! i
```

Ulf Norell

Safe lookup in Agda.

Theorem proving in Agda

$$
\begin{aligned}
&\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathit{zero} \quad + n = n \\
&\mathit{suc}\ m + n = \mathit{suc}\ (m + n)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{assoc} : \{i\ j\ k : \mathbb{N}\} \to i + (j + k) \equiv (i + j) + k \\
&\mathit{assoc}\ \mathit{zero} \quad j\ k = \mathit{refl} \\
&\mathit{assoc}\ (\mathit{suc}\ i)\ j\ k = \mathit{cong}\ \mathit{suc}\ (\mathit{assoc}\ i\ j\ k)
\end{aligned}
$$

- Exploit Curry-Howard.
- Think of proofs as programs.
- Termination checker to achieve logical soundness.

Basic ingredients of Type Theory

Per Martin-Löf

Π-types $(x : A) \to B\ x$ or $\{x : A\} \to B\ x$

- Generalize function types ($A \to B$).
- Represent universal quantification
- Alternative syntax: $\Pi\ [x : A]\ B\ x$

Σ-types $\Sigma\ [x : A]\ B\ x$

- Generalize product types
- Represent existential quantification
- Usually curried away or replaced by datatypes

Equality types $a \equiv b$ (for $a\ b : A$)

- No simply typed correspondence
- Represent propositional equality
- Implicitly used in dependent datatypes (like *Vec* or *Fin*)

Equality to define inductive families

$$
\begin{array}{l}
\textbf{data } \mathit{Fin} : \mathbb{N} \to \mathit{Set} \textbf{ where} \\
\quad \mathit{zero} : \{\, n : \mathbb{N} \,\} \to \mathit{Fin}\ (\mathit{suc}\ n) \\
\quad \mathit{suc} \ \ : \{\, n : \mathbb{N} \,\} \to \mathit{Fin}\ n \to \mathit{Fin}\ (\mathit{suc}\ n)
\end{array}
$$

Fin is the initial algebra of the following functor:

$$
\begin{array}{l}
\mathit{TFin} : (\mathbb{N} \to \mathit{Set}) \to \mathbb{N} \to \mathit{Set} \\
\mathit{TFin}\ X\ n = (\Sigma\ [\, m : \mathbb{N}]\ (\mathit{suc}\ m \equiv n)) \\
\qquad\qquad\quad \uplus\ (\Sigma\ [\, m : \mathbb{N}]\ (\mathit{suc}\ m \equiv n) \times X\ m)
\end{array}
$$

Equality types

$$\textbf{data } \_\equiv\_ \; \{A : Set\} : A \to A \to Set \textbf{ where}$$
$$\quad refl : (a : A) \to a \equiv a$$

Proof: $\_\equiv\_$ is an equivalence relation (using pattern matching):

$$sym : \{A : Set\} \; (a\, b : A) \to a \equiv b \to b \equiv a$$
$$sym\ a\ .a\ (refl\ .a) = refl\ a$$
$$trans : \{A : Set\} \; (a\, b\, c : A) \to a \equiv b \to b \equiv c \to a \equiv c$$
$$trans\ a\ .a\ b\ (refl\ .a)\ q = q$$

J : the eliminator

$$
\begin{aligned}
& J : \{A : Set\} \\
& \quad (M : (a\,b : A) \to a \equiv b \to Set) \\
& \quad \to ((a : A) \to M\;a\;a\;(refl\;a)) \\
& \quad \to (a\,b : A)\;(p : a \equiv b) \to M\;a\;b\;p \\
& J\;M\;m\;a\;.a\;(refl\;.a) = m\;a
\end{aligned}
$$

- Think of induction on equality proofs
- Alternative to pattern matching
- Combinator instead of a scheme.

sym and *trans* from *J*

We can derive *sym* and *trans* from *J* alone:

$$
\begin{aligned}
&sym : \{A : Set\}\ (a\ b : A) \to a \equiv b \to b \equiv a\\
&sym = J\ (\lambda\ a'\ b'\ \_ \to b' \equiv a')\\
&\qquad\quad (\lambda\ a' \to refl\ a')
\end{aligned}
$$

$$
\begin{aligned}
&trans : \{A : Set\}\ (a\ b\ c : A) \to a \equiv b \to b \equiv c \to a \equiv c\\
&trans\ a\ b\ c = J\ (\lambda\ a'\ b'\ \_ \to b' \equiv c \to a' \equiv c)\\
&\qquad\qquad\qquad (\lambda\ a' \to \lambda\ q' \to q')\\
&\qquad\qquad\qquad a\ b
\end{aligned}
$$

Uniqueness of Identity Proofs

- Can all pattern matching programs derived using J?

$$\begin{aligned}&uip : \{A : Set\}\ (a\ b : A)\ (p\ q : a \equiv b) \to p \equiv q\\ &uip\ .b\ b\ (refl\ .b)\ (refl\ .b) = refl\ (refl\ b)\end{aligned}$$

- Attempts to prove *uip* fail.
- We cannot use J to eliminate proofs of the type $a \equiv a$.

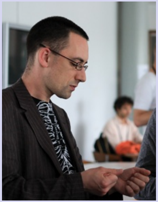
A 2nd eliminator K

$$
\begin{aligned}
&K : \{A : Set\} \\
&\quad (M : (a : A) \to a \equiv a \to Set) \\
&\quad \to ((a : A) \to M\ a\ (refl\ a)) \\
&\quad \to (a : A)\ (p : a \equiv a) \to M\ a\ p \\
&K\ M\ m\ a\ (refl\ .a) = m\ a
\end{aligned}
$$

using K and J we can derive *uip*:

$$
\begin{aligned}
&uip : \{A : Set\}\ (a\ b : A)\ (p\ q : a \equiv b) \to p \equiv q \\
&uip = J\ (\lambda\ a'\ b'\ p' \to (q' : a' \equiv b') \to p' \equiv q') \\
&\qquad\qquad (K\ (\lambda\ a''\ q'' \to refl\ a'' \equiv q'') \\
&\qquad\qquad\quad (\lambda\ a'' \to refl\ (refl\ a'')))
\end{aligned}
$$

Conor's PhD

Conor McBride (1999): 

J and *K* and the eliminators for other datatypes are enough to implement pattern matching.

But:

Do we really need K?

Groupoids

- While we cannot show that all equality proofs are equal using only J.
- We can show some equations between equality proofs.
- Equality proofs from a **groupoid**.
- A **groupoid** is a category where every morphism has an inverse (i.e. is an isomorphism).
- As categories generalize monoids and preorders ...
- ...groupoids generalize groups and equivalence relations

Equality forms a groupoid

Only using J we can prove:

$$
\begin{aligned}
&\mathit{lneutr} : \mathit{trans}\ \mathit{refl}\ p \equiv p \\
&\mathit{rneutr} : \mathit{trans}\ p\ \mathit{refl} \equiv p \\
&\mathit{assoc} : \mathit{trans}\ (\mathit{trans}\ p\ q)\ r \equiv \mathit{trans}\ p\ (\mathit{trans}\ q\ r) \\
&\mathit{linv} : \mathit{trans}\ (\mathit{sym}\ p)\ p \equiv \mathit{refl} \\
&\mathit{rinv} : \mathit{trans}\ p\ (\mathit{sym}\ p) \equiv \mathit{refl}
\end{aligned}
$$

Hofmann/Streicher

Hofmann/Streicher 94  

Groupoids form a model of Type Theory in which *uip* doesn't hold.
Hence *uip* is not derivable from *J* only.

Incompleteness?

- We can view the lack of *uip* as an incompleteness of Martin-Löf's original formulation of equality types.
- This can easily be fixed by adding K.
- There is another incompleteness of equality types.
- Which is easier to show.
- But harder to fix!

Consider the functions

$$f : \mathbb{N} \to \mathbb{N}$$
$$f = \lambda\ n \to n + 0$$
$$g : \mathbb{N} \to \mathbb{N}$$
$$g = \lambda\ n \to n$$

We can show

$$exteq : (n : \mathbb{N}) \to f\ n \equiv g\ n$$
$$exteq\ n = add0lem\ n$$

but we cannot show

$$eq : f \equiv g$$

because if such a proof exists.
Then there is one in normal from (*refl*).
And f and g would have to be convertible (same normal form).
However, $n + 0$ and n are not convertible.

Extensionality

This shows that the principle:

$$\begin{array}{l} ext : \{A\ B : Set\}\ (f\ g : A \to B) \\ \quad \to ((a : A) \to f\ a \equiv g\ a) \to f \equiv g \end{array}$$

is not provable in Type Theory.

Data vs codata

- Data (like $\mathbb{N}$) is defined by the way it is constructed.
- Codata (like functions) is defined by the way it is eliminated.
- Data is based on a producer contract, the producer only uses the allowed constructors.
- Codata is based on a consumer contract, the consumer only uses the allowed eliminators.
- The producer contract justifies elimination principles (like induction) for data.
- The consumer contract justifies coelimination principles (like coinduction and extensionality) for codata.

The Leibniz principle

- Any two objects should be either distinguishable (without using equality) or equal.
- Since all we can do with a function is to apply it, two extensionally equivalent functions should be equal.

Axiom?

- Why don't we add *ext* as an axiom?
- Disadvantage: this induces non-canonical elements in other types.

$$\begin{aligned} &\mathit{strange} : \mathbb{N} \\ &\mathit{strange} = \mathit{subst}\ (\lambda\ \_ \rightarrow \mathbb{N})\ (\mathit{ext}\ f\ g\ \mathit{exteq})\ 0 \end{aligned}$$

- Adding axioms destroys the computational structure of Type Theory.

Setoids?

- A set with an equivalence relation is called a setoid.
- We can define the setoid of functions with extensional equality.
- We define operations on setoids instead of sets.
- Disadvantages:
  - Each time we have to prove that any operation preserves extensional equality even though we know this is always true.
  - We have to decide which sets we turn into setoids and which we leave as sets. This leads potentially to many copies of a given operation.
- Why not working in the Type Theory generated by setoids?

Extensional Equality in Intensional Type Theory

- Indeed, this was the idea which lead to my LICS 99 paper.
- However, Setoids are not a model of Type Theory because certain equalities don't hold.
- E.g. the Beck-Chevallier condition fails

  $$(\Pi x : A.Bx)[\delta] = \Pi x : A[\delta].(Bx)[\delta]$$

  because both sides produce different equality proofs.
- We can address this by introducing a type **Prop** with the property that all proofs of a proposition are *convertible*.
- While this is a non-standard Type Theory, it is possible to implement such a theory.
- However, nobody ever implemented a Type Theory based on my LICS99 paper.

Observational Type Theory

- Jointly with Conor McBride and Wouter Swierstra we developed a more syntactic approach to the setoid model:
  Observational Type Theory, now (PLPV 08)
- Equality is defined by recursion over the types (following the setoid model).
- We also define

$$subst : \{A : Set\}\ \{B : A \to Set\}\ \{a\ b : A\} \to a \equiv b \to B\ a \to B\ b$$

  by recursion over B.
- Other constants, in particular

$$cong : \{A\ B : Set\}\ (f : A \to B)\ \{a\ b : A\} \to a \equiv b \to f\ a \equiv f\ b$$

  are added as axioms.
- We have irreducible terms in equality types.
  But not in other types (like $\mathbb{N}$).
- This is the basis for the ongoing implementation of **Epigram 2.**

Equality of types

- When should two types be provably equal?
- All operations in Type Theory preserve isomorphisms.
- Unlike Set Theory, e.g. $\{0,1\} \simeq \{1,2\}$ but $\{0,1\} \cup \{0,1\} \not\simeq \{0,1\} \cup \{1,2\}$.
- Indeed, isomorphic types are propositionally indistinguishable in Type Theory.
- Leibniz principle: isomorphic sets should be equal!?

Univalent Type Theory

- Vladimir Voevodsky proposed a
  new principle for Type Theory: the univalence principle.
- This is inspired by models of Homotopy theoretic models of Type Theory.
- He defines the notion of *weak equivalence* of types.

Voevodsky's Univalence Principle 

Equality of types is weakly equivalent to weak equivalence

- Using this principle we can show that isomorphic types are equal.
- It also implies *ext*.
- However, it is incompatible with *uip* and K.

Dimensions of types

- A type which has exactly one element is 0-dimensional.
  the contractible types.
- A type whose equality is 0-dimensional is 1-dimensional
  the propositions.
- A type whose equality is 1-dimensional is 2-dimensional
  the sets
- There are higher dimensional types, such as the universe of small sets (dimension 3).

Conclusions

- If we want to construct a univalent Type Theory we have to give up UIP.
- We can add the Univalence Principle as an axiom, but this destroys the computational structure of Type Theory.
- However, eliminating extensionality principles seems to rely on proof-irrelvance.
- Can we develop an extensional type theory which is not proof-irrelevant?
- And where univalence is provable?

- A type is contractible, if it has precisely one element:

$$\begin{aligned}&\mathit{Contr} : \mathit{Set} \to \mathit{Set}\\ &\mathit{Contr}\ A = \Sigma\,[a : A]\ ((a' : A) \to a \equiv a')\end{aligned}$$

- We define the inverse image of a function:

$$\begin{aligned}&\_^{\,-1} : \{A\ B : \mathit{Set}\}\ (f : A \to B)\ (b : B) \to \mathit{Set}\\ &(f^{\,-1})\ b = \Sigma\,[a : \_]\ (f\ a \equiv b)\end{aligned}$$

- A function is a weak equivalence if the inverse image is everywhere contractible:

$$\begin{aligned}&\mathit{Weq} : \{A\ B : \mathit{Set}\}\ (f : A \to B) \to \mathit{Set}\\ &\mathit{Weq}\ f = (b : \_) \to \mathit{Contr}\ ((f^{\,-1})\ b)\end{aligned}$$

- Two types are weakly equivalent, if there is a weak equivalence between them:

$$\_ \approx \_ : (A\ B : Set) \to Set$$
$$A \approx B = \Sigma\,[f : (A \to B)]\,(Weq\ f)$$

- Weak equivalence is reflexive:

$$refl\approx : \{A : Set\} \to A \approx A$$

- Hence equality implies weak equivalence:

$$\equiv\to\approx : \{A\ B : Set\} \to A \equiv B \to A \approx B$$
$$\equiv\to\approx\ refl = refl\approx$$

- Univalence is to postulate that the above map is a weak equivalence:

$$postulate\ unival : \{A\ B : Set\} \to Weq\ (\equiv\to\approx\{A\}\ \{B\})$$