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

Safe lookup in Agda.
Theorem proving in Agda

\[
_+_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}
\]

\[
\text{zero} + n = n
\]

\[
suc m + n = suc (m + n)
\]

\[
\text{assoc} : \{i \ j \ k : \mathbb{N}\} \to i + (j + k) \equiv (i + j) + k
\]

\[
\text{assoc zero } j \ k = \text{refl}
\]

\[
\text{assoc } (\text{suc } i) j \ k = \text{cong suc } (\text{assoc } i \ j \ k)
\]

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

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

**data** \( \text{Fin} : \mathbb{N} \rightarrow \text{Set} \) **where**

\[
\begin{align*}
\text{zero} & : \{ n : \mathbb{N} \} \rightarrow \text{Fin} (\text{suc} n) \\
\text{suc} & : \{ n : \mathbb{N} \} \rightarrow \text{Fin} n \rightarrow \text{Fin} (\text{suc} n)
\end{align*}
\]

\( \text{Fin} \) is the initial algebra of the following functor:

\[
\text{TFin} : (\mathbb{N} \rightarrow \text{Set}) \rightarrow \mathbb{N} \rightarrow \text{Set}
\]

\[
\text{TFin} X n = (\sum [m : \mathbb{N}] (\text{suc} m \equiv n)) \\
\uplus (\sum [m : \mathbb{N}] (\text{suc} m \equiv n) \times X m)
\]
Equality types

```
data _≡_ { A : Set } : A → A → Set where
  refl : (a : A) → a ≡ a
```

Proof: _≡_ is an equivalence relation (using pattern matching):

```
sym : { A : Set } (a b : A) → a ≡ b → b ≡ a
sym a .a (refl .a) = refl a
trans : { A : Set } (a b c : A) → a ≡ b → b ≡ c → a ≡ c
trans a .a b (refl .a) q = q
```
**J : the eliminator**

\[ J : \{ A : \text{Set} \} \]
\[ (M : (a b : A) \to a \equiv b \to \text{Set}) \]
\[ \to ((a : A) \to M a a (\text{refl} a)) \]
\[ \to (a b : A) (p : a \equiv b) \to M a b p \]

\[ J M m a .a (\text{refl} .a) = m a \]

- 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:

\[
\begin{align*}
\text{sym} : & \{ A : \text{Set} \} \ (a \ b : A) \to a \equiv b \to b \equiv a \\
\text{sym} = & \ J (\lambda \ a' \ b' \ _ \to b' \equiv a') \\
& (\lambda \ a' \to \text{refl} \ a') \\
\text{trans} : & \{ A : \text{Set} \} \ (a \ b \ c : A) \to a \equiv b \to b \equiv c \to a \equiv c \\
\text{trans} \ a \ b \ c = & \ J (\lambda \ a' \ b' \ _ \to b' \equiv c \to a' \equiv c) \\
& (\lambda \ a' \to \lambda \ q' \to q') \\
& a \ b
\end{align*}
\]
Uniqueness of Identity Proofs

- Can all pattern matching programs derived using $J$?

\[
\text{\textit{uip}} : \{ A : \text{Set} \} (a \ b : A) (p \ q : a \equiv b) \rightarrow p \equiv q \\
\text{\textit{uip}} . b \ b (\text{refl} \ . b) (\text{refl} \ . b) = \text{refl} (\text{refl} \ b)
\]

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

A 2nd eliminator $K$

$K : \{ A : \text{Set} \}$

$(M : (a : A) \to a \equiv a \to \text{Set})$

$\to ((a : A) \to M a (\text{refl} \ a))$

$\to (a : A) (p : a \equiv a) \to M a p$

$K \ M \ m \ a \ (\text{refl} \ . a) = m \ a$

using $K$ and $J$ we can derive $\text{uip}$:

$\text{uip} : \{ A : \text{Set} \} (a \ b : A) (p \ q : a \equiv b) \to p \equiv q$

$\text{uip} = J \ (\lambda a' \ b' \ p' \to (q' : a' \equiv b') \to p' \equiv q')$

$(K \ (\lambda a'' \ q'' \to \text{refl} \ a'' \equiv q''))$

$(\lambda a'' \to \text{refl} \ (\text{refl} \ a''))$
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 $K$?
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{align*}
\text{Ineutr} : & \quad \text{trans refl } p \equiv p \\
\text{rneutr} : & \quad \text{trans } p \ \text{refl} \equiv p \\
\text{assoc} : & \quad \text{trans (trans } p \ q) \ r \equiv \text{trans } p \ (\text{trans } q \ r) \\
\text{linv} : & \quad \text{trans (sym } p) \ p \equiv \text{refl} \\
\text{rinv} : & \quad \text{trans } p \ (\text{sym } p) \equiv \text{refl}
\end{align*}
Groupoids form a model of Type Theory in which $uip$ doesn’t hold. Hence $uip$ is not derivable from $J$ only.
We can view the lack of \textit{uip} as an incompleteness of Martin-Löf’s original formulation of equality types. This can easily be fixed by adding \textit{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

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

but we cannot show

\[ \text{eq} : f \equiv g \]

because if such a proof exists.
Then there is one in normal from (\textit{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:

$$\text{ext} : \{ A B : \text{Set} \} \ (f \ g : A \to B) \to ((a : A) \to f \ a \equiv g \ a) \to f \equiv g$$

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.
Why don’t we add *ext* as an axiom?

Disadvantage: this induces non-canonical elements in other types.

\[
\text{strange} : \mathbb{N} \\
\text{strange} = \text{subst} \ (\lambda _ \rightarrow \mathbb{N}) \ (\text{ext} \ f \ g \ \text{exteq}) \ 0
\]

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?
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

\[
\text{subst} : \{ A : \text{Set} \} \{ B : A \to \text{Set} \} \{ a b : A \} \to a \equiv b \to B a \to B b
\]

by recursion over \( B \).
- Other constants, in particular

\[
\text{cong} : \{ A B : \text{Set} \} (f : A \to B) \{ a b : A \} \to a \equiv b \to f a \equiv f b
\]

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 \textit{uip} and \textit{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-irrelevance.
- 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:

\[ \text{Contr} : \text{Set} \to \text{Set} \]

\[ \text{Contr} \ A = \sum [a : A] \ ((a' : A) \to a \equiv a') \]

We define the inverse image of a function:

\[ _{-1} : \{ A B : \text{Set} \} \ (f : A \to B) \ (b : B) \to \text{Set} \]

\[ (f^{-1}) \ b = \sum [a : \_] \ (f \ a \equiv b) \]

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

\[ \text{Weq} : \{ A B : \text{Set} \} \ (f : A \to B) \to \text{Set} \]

\[ \text{Weq} \ f = (b : \_) \to \text{Contr} \ ((f^{-1}) \ b) \]
• Two types are weakly equivalent, if there is a weak equivalence between them:

\[ \_ \approx \_ : (A \ B : \text{Set}) \to \text{Set} \]
\[ A \approx B = \Sigma [f : (A \to B)] \ (\text{Weq} \ f) \]

• Weak equivalence is reflexive:

\[ \text{refl} \approx : \{ A : \text{Set} \} \to A \approx A \]

• Hence equality implies weak equivalence:

\[ \equiv \to \approx : \{ A \ B : \text{Set} \} \to A \equiv B \to A \approx B \]
\[ \equiv \to \approx \text{refl} = \text{refl} \approx \]

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

\[ \text{postulate unival} : \{ A \ B : \text{Set} \} \to \text{Weq} \ (\equiv \to \approx \{ A \} \{ B \}) \]