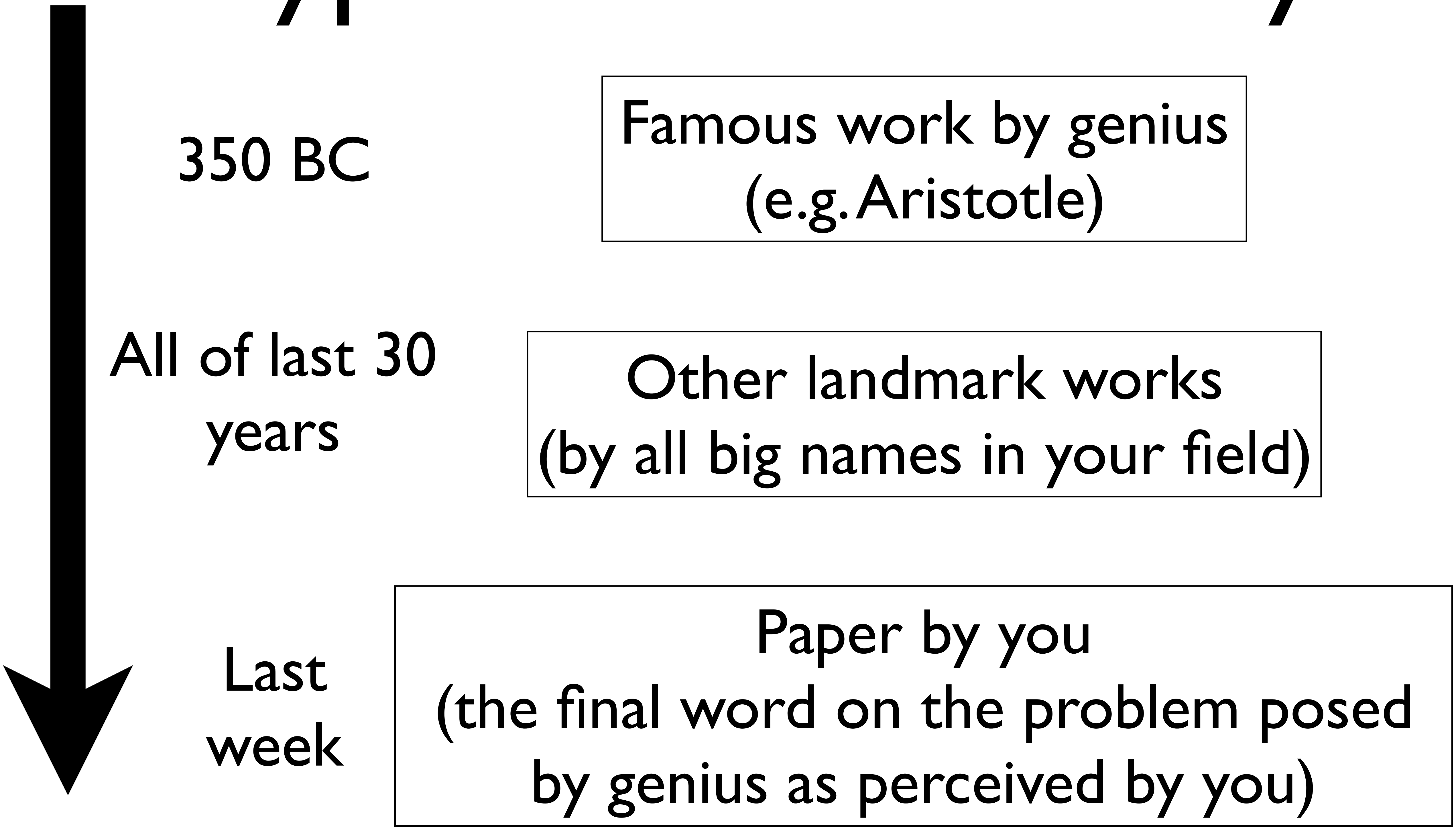
A biased history of equality in type theory

Some equations are more equal than others

James Chapman
Institute of Cybernetics, Tallinn

Semantics days - Andu

A typical biased history



time

350 BC

Famous work by genius
(e.g. Aristotle)

All of last 30 years

Other landmark works
(by all big names in your field)

Last week

Paper by you
(the final word on the problem posed by genius as perceived by you)

Why should you care about equality?

- You’re a type theorist and it’s your favourite topic?
- It’s often necessary when using theorem provers and implementations of dependently typed programming.
    - Coq, Agda, Epigram, etc.
- When you start having to use equality proofs to ‘fix up’ you types your carefully crafted proofs turn into a hideous and agonising mess. (Maybe this is just me...)
- Different implementors/theoreticians take a variety of approaches with different trade-offs - it’s not over yet!

In the beginning there was Martin-Löf’s theory of types

- Known as type theory for short.
- Based on ideas of intuitionistic mathematics, as conceived by BHK, it is “intended to be a full-scale system for intuitionistic mathematics”
- It is at the same time a programming language and a logic.
    - Proposition = Type
    - Proof = Program

Type theory formally

- Type theory is usually presented as system of logical inference rules.
  - First we define a number of judgements:
    - $\Gamma \vdash$ — Γ *is a context*
    - $\Gamma \vdash T$ — *T is a type*
    - $\Gamma \vdash T = T‘$ — *T is equal to T’*
    - $\Gamma \vdash t : T$ — *t is at term of type T*
    - $\Gamma \vdash t = t’ : T$ — *t is equal to t’*
  - ~~Next we give the rules that inhabit the judgements.~~

Definitional equality (for computers)

- The equality judgements on the previous slide are known as definitional equality.
- e.g. internal rules like beta-equality and also the equality we use when making a *definition*:
  - `f 0 = x`
  - `f (n + 1) = y`
- This is the the equality the type checker uses.
  - `f 0` equals `x`, we cannot distinguish between them.

Definitional equality cont.

- There are choices here: e.g. do we include eta-equality for functions, pairs, and unit:
  - `λ x . f x = f`
  - `(fst p, snd p) = p`
  - `void = u`
- Coq doesn't, Agda and Epigram do.
- My view is to have the strongest possible decidable equality: it saves you work.
- Definitional equality is an essential component of the type checker.

Type checking example

```
data Vec (A : Set) : Nat → Set where
  nil  : Vec A zero
  cons : A → Vec A n → Vec A (suc n)

app : Vec A m → Vec A n → Vec A (m + n)
app nil         ws = ws
app (cons v vs) ws = cons v (app vs ws)
```

Typing constraints from definition of app:

```
Vec A (zero  + n) = Vec A n
Vec A (suc m + n) = Vec A (suc (m + n))
```

Propositional Equality (for people)

- Definitional equality is the equality you use when make a definition. f x = x
- Propositional equality is what you use when stating a proposition (e.g x ≅ x + 0) that you will prove yourself.
- Next we move on to discussing different varieties of propositional equality

Equality reflection

- One seemingly simple option is to define an provable equality type ≅ and then a reflection rule:

$$\frac{p : X \cong Y}{X = Y}$$

- But, notice we have thrown away the proof.
- With this rule, typechecking become undecidable. This is the choice taken in ETT
- In ETT type checking requires proof.

Rolling your own prop. equality types

- Type theory is a rich language we are free to define, as a new type, an equivalence (or any other) relation on elements of another type.
- It is not *built-in* in any sense, nor need it be.
- We can define it by a type valued recursive function or as in inductive data type.
- If it is defined inductively we can choose which properties are assumed and which are derived.

Recursive equality on Nat

```
data One : Set where
  void : One

data Zero : Set where

data Nat : Set where
  z : Nat
  s : Nat → Nat

req : Nat → Nat → Set
req z     z     = One
req z     (s n) = Zero
req (s m) z     = Zero
req (s m) (s n) = req m n
```

We can then prove that this is an equivalence relation. E.g.

```
trans : ∀m n o : req m n -> req n o -> req m o
```

Inductive equality for Nat

```
data ieq : Nat → Nat → Set where
  zeq : ieq z z
  seq : ∀ m n →
        ieq m n → ieq (s m) (s n)
```

We are free to either add refl., sym. and trans. as constructors or prove them.

Here it is simpler to prove them (fewer constructors here means fewer case in our proofs).

In a more complex situation than Nat adding these properties might be really adding something...

A propositional equality for all types

- Do we really have to define our own equality every time we define a new type? No!

- We (either the user or the language implementor) *can* define a propositional equality for all types.

- In the remaining slides we will look at the different design decisions and issues for Propositional Equality.

The original ‘Identity’ type

```
data Id {X : Set} : X → X → Set where
  refl : (x : X) → Id x x

subst : (X : Set)
        (P : X → Set)
        (x x' : X) →
        Id x x' → P x → P x'
subst P .x .x (refl x) t = t
```

subst is a (useful) simplification, really we get given an eliminator where P depends on x x’ and the equality proof.

Intentional vs. extensional

- Extensional equality equates things that behave the same.
- Intentional equality equates things if they are constructed in the same way.
- addition on natural numbers defined by induction on the first arg. is extensionally equal to addition defined by induction on the second arg. but not intentionally
  - `∀m n. Id (plus1 m n) (plus2 m n)` ✔
  - `Id plus1 plus2` ✖

What's wrong with Id?

- It turns out to be a bit weak.
  - It is not extensional.
  - We cannot prove that all equality proofs are equal to each other or equal to refl (proof irrelevance or eta equality).
    - We can prove eta for other types but not for equality itself: we don't have `refl = p`. (H&S)
  - The internals of pattern matching use equality but need something stronger than the ordinary eliminator for the equality type (McBride's thesis)

How do we improve it?

- We hand-craft a stronger eliminator for equality.
- It becomes a built-in type with a stronger computation principle than the usual automatically defined one (Thomas Streicher's Axiom K). This is sufficient for proof irrelevance and pattern matching.
- Also, we can generalise the definition itself (McBride's John Major equality) which turns out to be more convenient than Id and K.

John Major Equality

```
data JM {X : Set} : X → {Y : Set} →
                    Y → Set where
  refl : (x : X) → JM x x

resp : {X : Set}(P : X → Set)
       (f : (x : X) → P x)
       (x x' : X) → JM x x' →
       JM (f x) (f x')
resp P f .x .x (refl x) = refl (f x)
```

Notice that resp is not well typed for Id.

We would need to use subst to define it. Our experience is if you have to use subst then you're doing it wrong.

But! Still no extensionality :(

- We can simulate it using setoids (Bishop sets).
- Bishop's idea is that a set always comes equipped with its own equivalence relation.
- We have complete control: we can add extensionality or any other property as an axiom for the particular set.
- But, it is technically awkward to use and we need to define it manually for each type.
- Can we do better?

Yes! Observational Equality

- It internalises the setoid construction but adds significant automation
- An appropriate notion of equality and reasoning principles are computed for each type.
  - This is made possible by using a ‘closed type theory’. Rather than extending the theory with each new datatype definition:
    - We have a universe of datatypes (new datatypes are defined as codes in the universe).
    - Equality is defined by recursion on codes.
    - The case for functions computes to the extensionality principle.

Bibliography

- Foundations of Constructive Analysis - Bishop, 1967
- About models for intuitionistic type theories and the notion of definitional equality - Martin-Löf, 1975
- Intuitionistic Type Theory - Martin-Löf, 1984
- A groupoid model refutes uniqueness of identity proofs. Hofmann & Streicher, 1994
- Dependently Typed Functional Programs and their Proofs - McBride, 1999
- Observational Equality Now! Altenkirch, McBride, & Swierstra, 2007
- Epigram 2 prototype - Brady, Chapman, Dagand, Gundry, McBride, Morris, and Norell, 2010