THE LEGACY OF
VLADIMIR VOEVODSKY

HOMOTOPY TYPES & RESIZING RULES

A FRESH LOOK AT THE IMPREDICATIVE SORT OF CIC

NICOLAS TABAREAU

Road Map

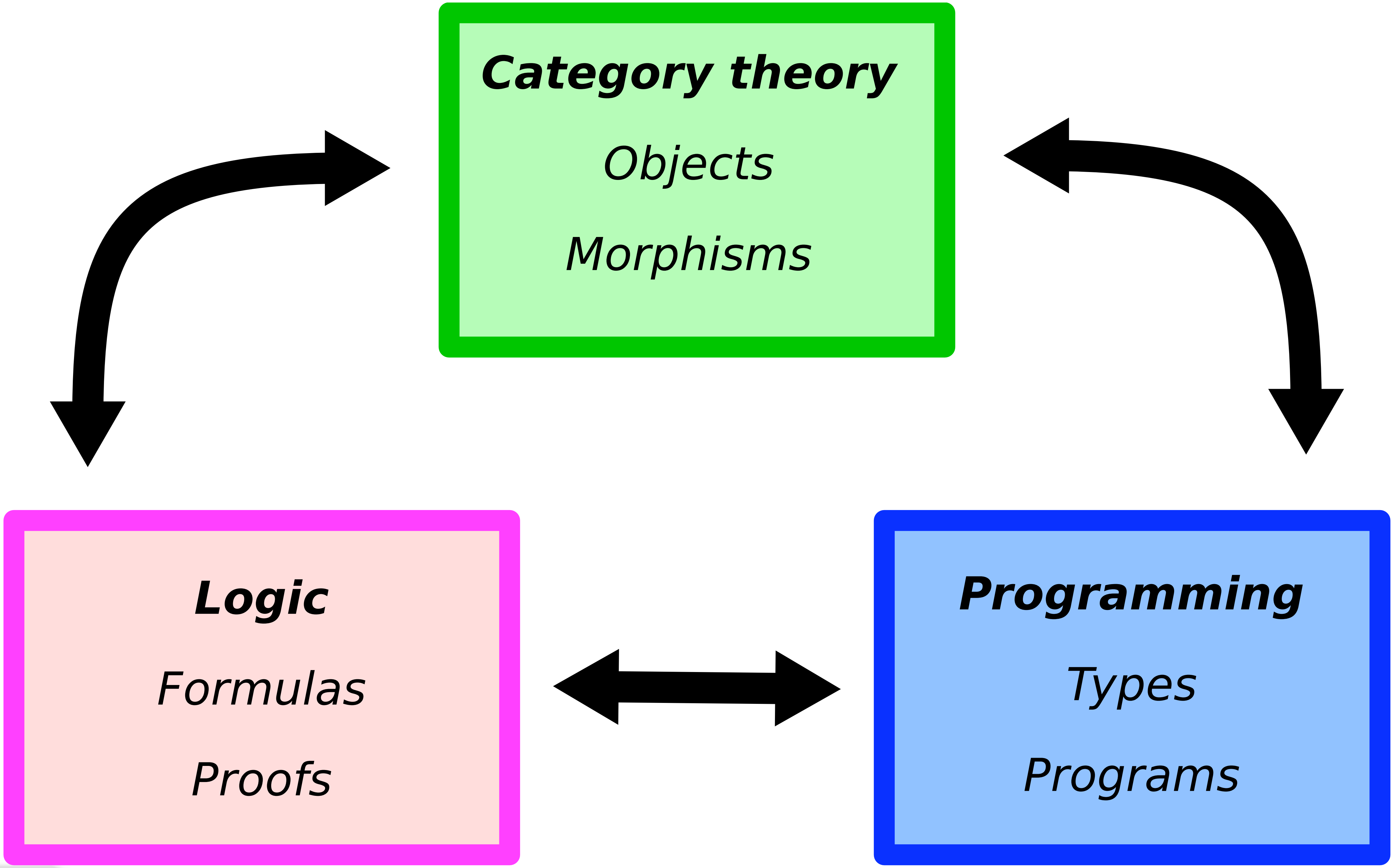
In this talk, I will recall two notions introduced by V.V. in 2006 in "*A very short note on homotopy λ-calculus*".

1. Homotopy types in type theory
2. Universe resizing rules

I will then explain how those two notions allow for a fresh look at the impredicative universe of CIC.

First, what is Type Theory about ?

The denotational semantics trinity



Category theory

Objects

Morphisms

Logic

Formulas

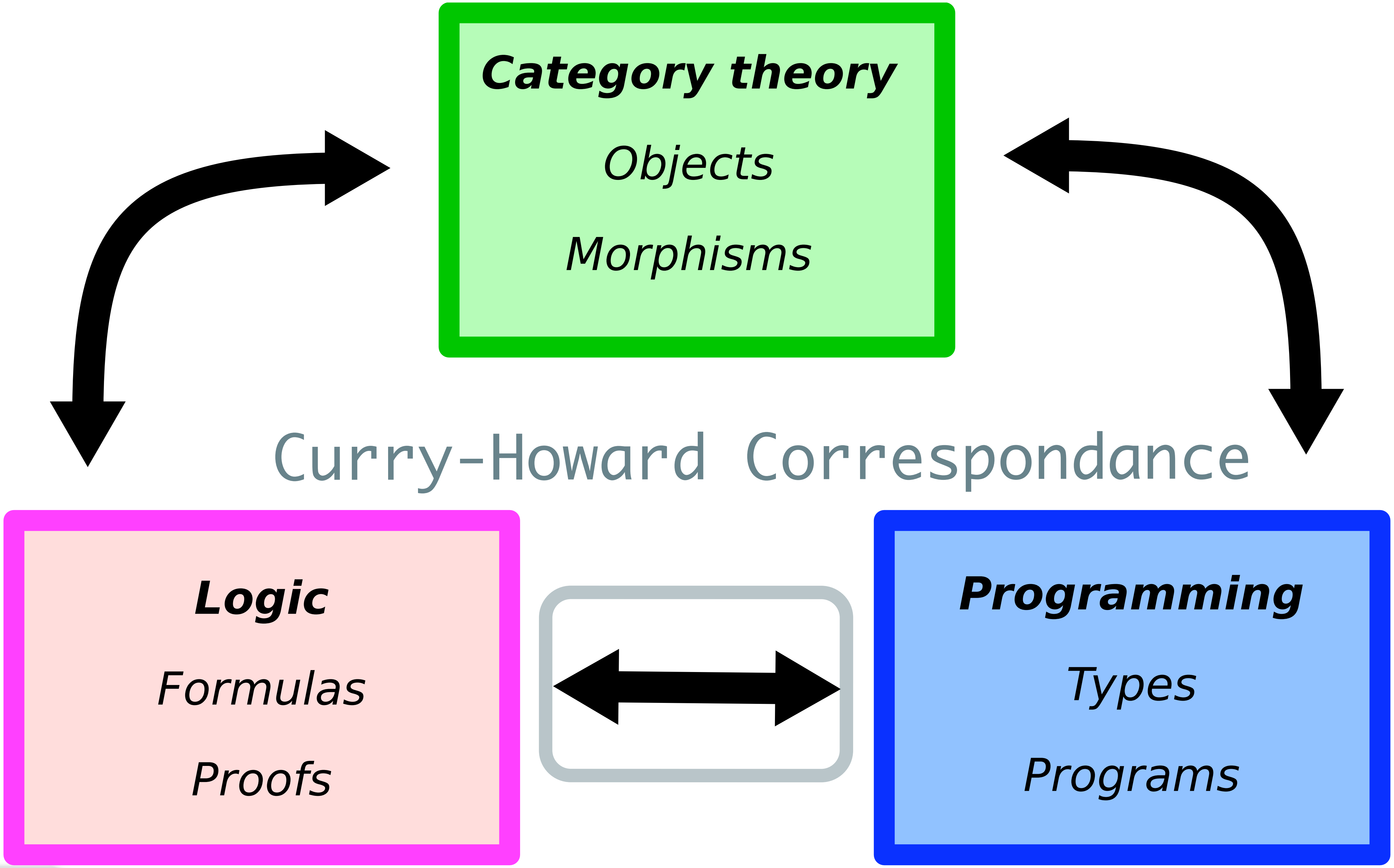
Proofs

Programming

Types

Programs

The denotational semantics trinity



Category theory

Objects

Morphisms

Curry-Howard Correspondance

Logic

Formulas

Proofs

Programming

Types

Programs

The simply typed λ-calculus

variable	$\dfrac{}{x : A \vdash x : A}$
abstraction	$\dfrac{\Gamma, x : A \vdash P : B}{\Gamma \vdash \lambda x.P : A \Rightarrow B}$
application	$\dfrac{\Gamma \vdash P : A \Rightarrow B \qquad \Delta \vdash Q : A}{\Gamma, \Delta \vdash PQ : B}$
weakening	$\dfrac{\Gamma \vdash P : B}{\Gamma, x : A \vdash P : B}$
contraction	$\dfrac{\Gamma, x : A, y : A \vdash P : B}{\Gamma, z : A \vdash P[x, y \leftarrow z] : B}$
exchange	$\dfrac{\Gamma, x : A, y : B, \Delta \vdash P : C}{\Gamma, y : B, x : A, \Delta \vdash P : C}$

Intuitionistic minimal logic

axiom	$\dfrac{}{A \vdash A}$
$\Rightarrow$ I	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$
$\Rightarrow$ E	$\dfrac{\Gamma \vdash A \Rightarrow B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}$
weakening	$\dfrac{\Gamma \vdash B}{\Gamma, A \vdash B}$
contraction	$\dfrac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$
exchange	$\dfrac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}$

Intuitionistic minimal logic

Curry
Howard

axiom

$$\frac{}{A \vdash A}$$

⇒ I

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

⇒ E

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}$$

weakening

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}$$

contraction

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

exchange

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}$$

Other correspondances

Cut elimination ⇔ β-reduction

Type Theory of Coq

Lifting the Curry-Howard correspondance to **dependent types** ⇒ more complex formulas

$$\frac{\prod n : \mathrm{nat}.\ \sum m : \mathrm{nat}.\ \mathrm{Id}\ (m, n + 1)}{\forall\ n : \mathrm{nat}.\ \exists\ m : \mathrm{nat}.\ m = n + 1}$$

Type Theory of Coq

Lifting the Curry-Howard correspondance to **dependent types** ⇒ more complex formulas

$$\textsc{Prod/Sigma}\;\frac{\Gamma, x : A \vdash B\ \texttt{type}}{\Gamma \vdash \Pi/\Sigma x : A.B\ \texttt{type}}$$

Type Theory of Coq

Lifting the Curry-Howard correspondance to **dependent types** ⇒ more complex formulas

$$\textsc{Prod/Sigma}\quad \frac{\Gamma, x : A \vdash B\ \texttt{type}}{\Gamma \vdash \Pi/\Sigma x : A.B\ \texttt{type}}$$

Type checking ⟺ Correctness checking

Type Theory and Logic

Types	Logic
A	proposition
$a : A$	proof
$B(x)$	predicate
$b(x) : B(x)$	conditional proof
$\mathbf{0}, \mathbf{1}$	$\bot, \top$
$A + B$	$A \vee B$
$A \times B$	$A \wedge B$
$A \to B$	$A \Rightarrow B$
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$
Id_A	equality $=$

Type Theory and Logic

Types	Logic
A	proposition
$a : A$	proof
$B(x)$	predicate
$b(x) : B(x)$	conditional proof
$\mathbf{0}, \mathbf{1}$	$\bot, \top$
$A + B$	$A \vee B$
$A \times B$	$A \wedge B$
$A \rightarrow B$	$A \Rightarrow B$
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$
Id_A	equality $=$

How is equality modeled ?

Equality in Type Theory

Equality is described using Martin-Löf Identity Type.

$$\mathsf{refl} : \prod_{a:A} (a =_A a)$$

Path induction: Given a family

$$C : \prod_{x,y:A} (x =_A y) \to \mathcal{U}$$

and a function

$$c : \prod_{x:A} C(x, x, \mathsf{refl}_x),$$

there is a function

$$f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x, y, p)$$

such that

$$f(x, x, \mathsf{refl}_x) :\equiv c(x).$$

Equality in Type Theory

Equality is described using Martin-Löf Identity Type.

$$\mathsf{refl} : \prod_{a:A} (a =_A a)$$

Leibniz principle of "Indiscernability of Identicals"

Path induction: Given a family

$$C : \prod_{x,y:A} (x =_A y) \to \mathcal{U}$$

and a function

$$c : \prod_{x:A} C(x, x, \mathsf{refl}_x),$$

there is a function

$$f : \prod_{(x,y:A)} \prod_{(p:x=_A y)} C(x, y, p)$$

such that

$$f(x, x, \mathsf{refl}_x) :\equiv c(x).$$

Equality in Type Theory

A formulation using the type system:

$$\textsc{Id}\ \frac{\Gamma \vdash T\ \texttt{type} \quad \Gamma \vdash A, B : T}{\Gamma \vdash \texttt{Id}_T\ A\ B\ \texttt{type}} \qquad \textsc{Id-Intro}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash \texttt{refl}_T\ t : \texttt{Id}_T\ t\ t}$$

$$\textsc{Id-Elim}\ (\mathrm{J})\ \frac{\Gamma \vdash i : \texttt{Id}_T\ t\ u \qquad \Gamma, x : T, e : \texttt{Id}_T\ t\ x \vdash P\ \texttt{type} \qquad \Gamma \vdash p : P\{t/x, \texttt{refl}_T\ t/e\}}{\Gamma \vdash \texttt{J}_{\lambda x\ e.P}\ i\ p : P\{u/x, i/e\}}$$

Type and Set Theory

Types	Sets
A	set
$a : A$	element
$B(x)$	family of sets
$b(x) : B(x)$	family of elements
$\mathbf{0}, \mathbf{1}$	$\emptyset, \{\emptyset\}$
$A + B$	disjoint union
$A \times B$	set of pairs
$A \to B$	set of functions
$\sum_{(x:A)} B(x)$	disjoint sum
$\prod_{(x:A)} B(x)$	product
Id_A	$\{ (x, x) \mid x \in A \}$

Problem with Identity Type

The following definitions should coincides with equality.

Functional Extensionality:

$$(f \sim g) :\equiv \prod_{x:A} (f(x) = g(x)).$$

Univalence:

$$(A \simeq B) :\equiv \sum_{f:A \to B} \mathsf{isequiv}(f)$$

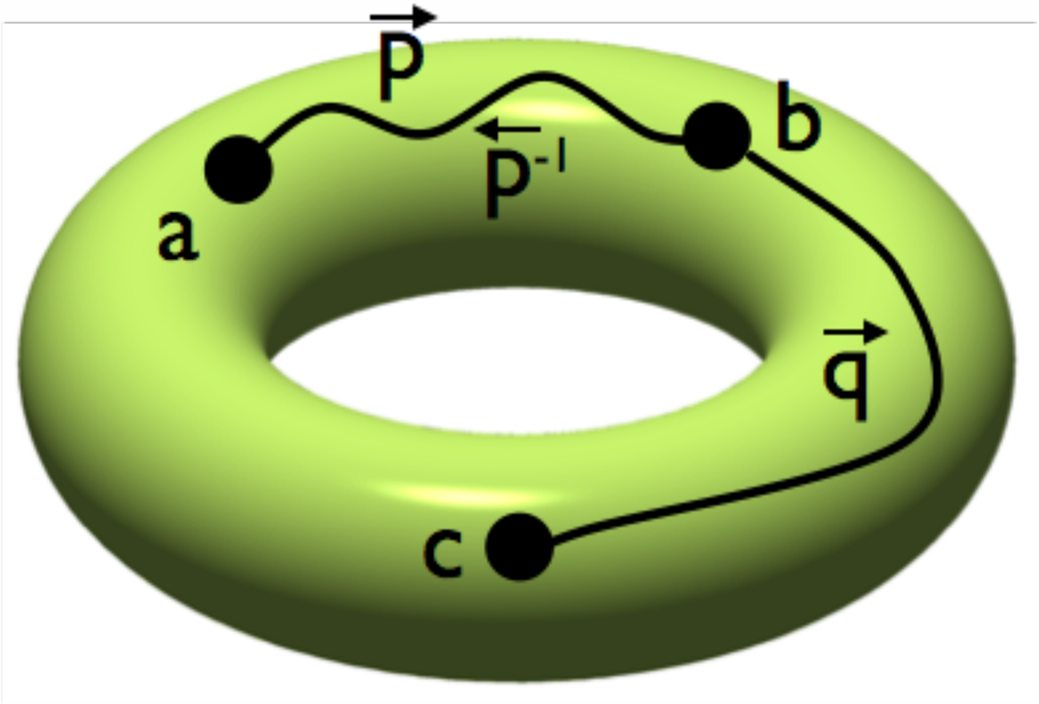
where $\mathsf{isequiv}(f) \;:\equiv\; \Big(\sum_{g:B\to A} (f \circ g \sim \mathrm{id}_B)\Big) \times \Big(\sum_{h:B\to A} (h \circ f \sim \mathrm{id}_A)\Big)$

Type and Homotopy Theory

Types	Homotopy
A	space
$a : A$	point
$B(x)$	fibration
$b(x) : B(x)$	section
$\mathbf{0}, \mathbf{1}$	$\emptyset, *$
$A + B$	coproduct
$A \times B$	product space
$A \to B$	function space
$\sum_{(x:A)} B(x)$	total space
$\prod_{(x:A)} B(x)$	space of sections
Id_A	path space A^I

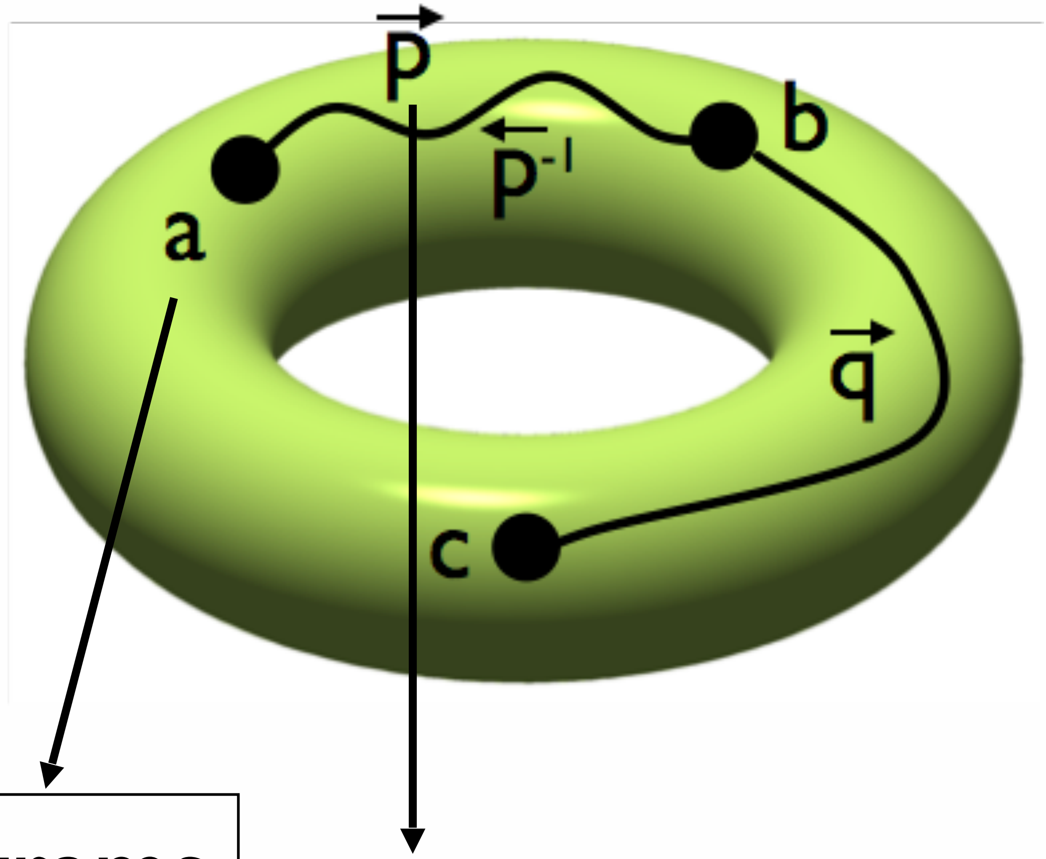
∞-groupoids and equality

type T is a space



∞-groupoids and equality

type T is a space

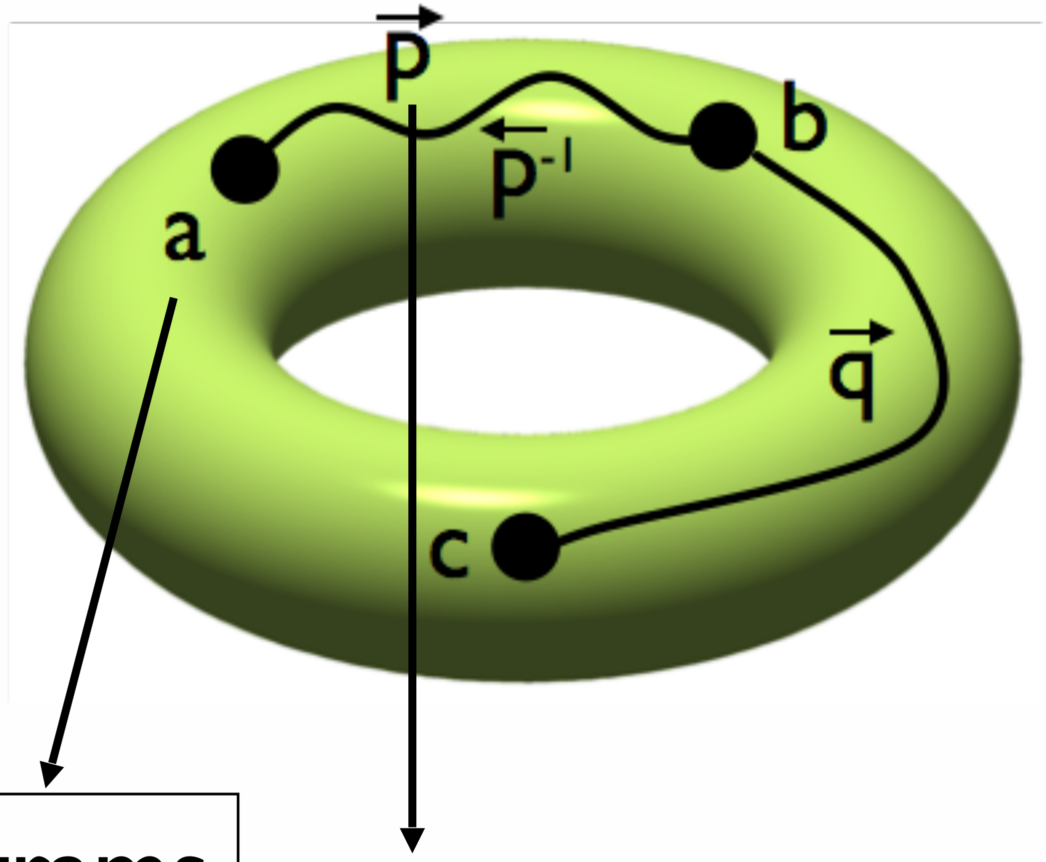


programs
a:T are
points

proofs of equality
p : a = b
are paths

∞-groupoids and equality

type T is a space



programs a:T are points

proofs of equality p : a = b are paths

Path operations:

$id : a =_T a$

$p^{-1} : b =_T a$

$q \circ p : a =_T c$

Homotopies:

left-id : $id \circ p =_{a=b} p$

right-id : $p \circ id =_{a=b} p$

assoc : $r \circ (q \circ p) =_{a=d} (r \circ q) \circ p$

A Hierarchy of Types

A Hierarchy of Types

One of the main contribution of V.V. in type theory is the notion of levels of homotopy of types.

A Hierarchy of Types

Types are classified by the complexity of their equality/identity type.

Simplest (singleton) types are called contractible:

$$\mathrm{isContr}(A) :\equiv \sum_{(a:A)} \prod_{(x:A)} (a = x).$$

A Hierarchy of Types

Types are classified by the complexity of their equality/identity type.

Proposition have a contractible equality:

$$\mathrm{isProp}(P) :\equiv \prod_{x,y:P} (x = y).$$

A Hierarchy of Types

Types are classified by the complexity of their equality/identity type.

Then, n-Types are defined inductively:

Define the predicate is-n-type : $\mathcal{U} \to \mathcal{U}$ for $n \geq -2$ by recursion as follows:

$$\text{is-}n\text{-type}(X) :\equiv \begin{cases} \mathsf{isContr}(X) & \text{if } n = -2, \\ \prod_{(x,y:X)} \text{is-}n'\text{-type}(x =_X y) & \text{if } n = n' + 1. \end{cases}$$

A Hierarchy of Types

This defines the following hierarchy:

Level of Type	**Homotopy Type Theory**
(-2)-Type	unit / contactible type
(-1)-Type	h-propositions
0-Type	h-sets
1-Type	h-groupoids
…	…
Type	∞-groupoids

Extensional principles

The following definitions should coincides with equality.

Functional Extensionality:

$$(f \sim g) :\equiv \prod_{x:A} (f(x) = g(x)).$$

Univalence:

$$(A \simeq B) :\equiv \sum_{f:A \to B} \mathsf{isequiv}(f)$$

where $\mathsf{isequiv}(f) \;:\equiv \Big(\sum_{g:B \to A} (f \circ g \sim \mathsf{id}_B)\Big) \times \Big(\sum_{h:B \to A} (h \circ f \sim \mathsf{id}_A)\Big)$

Extensional principles

It’s time for white board.

A Hierarchy of Universes

A Hierarchy of Universes

To avoid paradox à la Russell, we need to introduce a hierarchy of universes in type theory.

$$\vdash U_i : U_{i+1}$$

A Hierarchy of Universes

This is a sufficient condition to ensure consistency, but it is often a bit overkilled and one would like to relax it.

A Hierarchy of Universes

Syntactically, the management of the hierarchy can be improved by universe polymorphism which allows to use the same definition at different levels.

A Hierarchy of Universes

V.V. has proposed a semantic way to relax the hierarchy, based on so-called resizing rules.

Resizing Rules

Resizing rule for equivalent types.

$$(RR5) \quad \frac{U : Univ \qquad \Gamma \vdash X_1 : U \qquad \Gamma \vdash is : weq\, X_1\, X_2}{\Gamma \vdash X_2 : U}$$

(from V.V. 's talk at Bergen, 2011)

Resizing Rules

In a classical setting, every mere proposition is equivalent to either True or False.

True and False can be typed in the lowest universe.

Resizing Rules

Resizing rule for mere propositions.

$$\textbf{RR1} \qquad \frac{\Gamma \vdash is : isaprop\, X}{\Gamma \vdash X : UU}$$

Resizing Rules

Resizing rule for mere propositions.

$$\textbf{RR1} \qquad \frac{\Gamma \vdash is : isaprop\, X}{\Gamma \vdash X : UU}$$

This is corresponds to the impredicativity of Prop

A Fresh Look at Prop

A Fresh Look at Prop

This suggests that Prop should be interpreted as a universe of mere propositions.

A Fresh Look at Prop

This suggests that Prop should be interpreted as a universe of mere propositions.

Problem: In Coq,

$$x =_A y$$

is in Prop for all type A

A Fresh Look at Prop

Problem: In Coq,

$$x =_A y$$

is in Prop for all type A

This means that the current Prop is implicitly assuming that every type is an h-set !

A Fresh Look at Prop

One possible way out
(as done in the HoTT Coq library):

Treat Prop as a taboo and not use it.

A Fresh Look at Prop

But maybe we can do better and fix it ?

A Fresh Look at Prop

But maybe we can do better and fix it ?

The rest of this talk is joint work with Gaetan Gilbert and Matthieu Sozeau.

Gaetan is implementing this feature, to be integrated hopefully in a future version Coq.

Prop under the Knife of HoTT

When an inductive type is defined in Prop, it can be eliminated only when building a Prop.

Prop under the Knife of HoTT

When an inductive type is defined in Prop, it can be eliminated only when building a Prop.

This corresponds to the fact that propositional truncation can be eliminated

$$(A \to B) \to (||A|| \to B)$$

only when B is a mere proposition.

Prop under the Knife of HoTT

First motto:

> “Defining an inductive type in Prop corresponds to using propositional truncation”

Prop under the Knife of HoTT

First motto:

> “Defining an inductive type in Prop corresponds to using propositional truncation”

That is, morally, every type in Prop is squashed.

When Props produce Types

In CIC, there is the so-called singleton elimination:

> “A singleton definition has only one constructor and all the arguments of this constructor have type Prop.”

When Props produce Types

In CIC, there is the so-called singleton elimination:

> "A singleton definition has only one constructor and all the arguments of this constructor have type Prop."

This covers for instance conjunction or the accessibility predicate but also equality !

When Props produce Types

With this new insight, singleton elimination can be seen as a syntactic condition on P:Prop which ensures that

$$||P|| \cong P$$

Problem

Allowing squashed equality to be unsquashed
is implicitly assuming that every type is an h-set

UIP hard-coded

Problem

The problem is that it doesn't take into account the number of occurrences of parameters/arguments in the return type.

When Props produce Types (II)

```
Inductive eq (A:Type)(x:A): A -> Prop
:= eq_refl : eq A x x.
```

a variable that occurs twice must be in h-sets.

When Props produce Types (II)

```
Inductive eq (A:Type)(x:A): A -> Prop
:= eq_refl : eq A x x.
```

`x` occurs twice

a variable that occurs twice must be in h-sets.

When Props produce Types (II)

What about functions occurring in the return type ?

```
Vect (A : Prop) : nat -> Prop :=
   nil   : Vect A 0
| cons : A -> forall n : nat,
                  Vect A n -> Vect A (S n)
```

When Props produce Types (II)

What about functions occurring in the return type ?

```
Vect (A : Prop) : nat -> Prop :=
  nil  : Vect A 0
| cons : A -> forall n : nat,
              Vect A n -> Vect A (S n)
```

`S` **must be injective**

What about multiple constructors ?

```
Inductive le : nat -> nat -> Prop :=
   le_O : forall n : nat, O <= n
|  le_S : forall n m : nat, m <= n -> S m <= S n
```

What about multiple constructors ?

```
Inductive le : nat -> nat -> Prop :=
    le_O : forall n : nat, O <= n
|   le_S : forall n m : nat, m <= n -> S m <= S n
```

the return types of different constructors must be orthogonal

What about multiple constructors ?

```
Inductive le : nat -> nat -> Prop :=
    le_O : forall n : nat, O <= n
|   le_S : forall n m : nat, m <= n -> S m <= S n
```

Sums don't preserve mere propositions in general, but they do for disjoint sums.

the return types of different constructors must be orthogonal

Remark
Definitions Matter

```
Inductive le' (n : nat) : nat -> Prop :=
   le_n : n <= n
|   le_S : forall m : nat, n <= m -> n <= S m
```

Remark
Definitions Matter

```
Inductive le' (n : nat) : nat -> Prop :=
   le_n : n <= n
|   le_S : forall m : nat, n <= m -> n <= S m
```

the criterion does not work for this (equivalent) definition

When a Prop is h-Prop

1. every argument that does not appear
   in the return type must be in Prop
2. every argument/parameters that appears
   more than once in the return type must be h-Set
3. every argument that appears exactly once is OK
4. the return types of different constructors
   must be orthogonal

When a Prop is -1-Type

1. every argument that does not appear in the return type must be in -1-Type
2. every argument/parameters that appears more than once in the return type must be 0-Type
3. every argument that appears exactly once is OK
4. the return types of different constructors must be orthogonal

Going to Higher Level

This characterisation generalises to n-types

1. every argument that does not appear in the return type must be in n-Type
2. every argument/parameters that appears more than once in the return type must be (n+1)-Type
3. every argument that appears exactly once is OK
4. the return types of different constructors must be orthogonal

Going to Higher Level

This characterisation generalises to n-types

1. every argument that does not appear in the return type must be in n-Type
2. every argument/parameters that appears more than once in the return type must be (n+1)-Type
3. every argument that appears exactly once is OK
4. ~~the return types of different constructors must be orthogonal~~

only for mere proposition

Remark

This characterisation is very similar to what Jesper Cockx et al. use to do pattern-matching without K in Agda.

Remark

This characterisation is very similar to what Jesper Cockx et al. use to do pattern-matching without K in Agda.

We have extended it in February with Jesper,
I can talk about it offline.

What is this Impredicative Universe ?

The least we get is a new version of Coq:

- compatible with UIP
- compatible with univalence
- admitting the axiom :

```
forall (P:Prop) (x y : P), x = y
```

We Want More !

We Want More !

Replace the admissible axiom with a definitional equality:

$$\text{forall } (P{:}\mathrm{Prop})\ (x\ y : P),\ x \equiv y$$

Problem

Congruence with pattern-matching and fixpoints requires to apply inversion lemma even to neutral terms … and this potentially infinitely many times.

Problem

Congruence with pattern-matching and fixpoints requires to apply inversion lemma even to neutral terms … and this potentially infinitely many times.

A naive implementation gives rise to an undecidable type checker !

Acc is not an SProp

Perfectly valid mere proposition,
but with infinite unfolding …

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
   Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

Acc is not an SProp

Perfectly valid mere proposition,
but with infinite unfolding …

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
  Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

```
Definition Acc_inv : Acc R x -> forall y:A, R y x -> Acc R y.
```

Acc is not an SProp

Perfectly valid mere proposition,
but with infinite unfolding …

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
   Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

```
Definition Acc_inv : Acc R x -> forall y:A, R y x -> Acc R y.
```

a ≡ Acc_intro x (Acc_inv a) ≡ Acc_intro x (Acc_inv ...)

Acc is not an SProp

It is not possible to guess how many times an inhabitant of Acc R x has to be unfolded.

Termination-unfolding criterion

We need to enforce termination of inversion through a syntactic check similar to the guard condition for fixpoints.

That is, recursive arguments of a constructor must have as indices strict sub terms of the indices of the return type.

Examples

```
Inductive le : nat -> nat -> Prop :=
   le_O : forall n : nat, 0 <= n
|  le_S : forall n m : nat, m <= n -> S m <= S n
```

Examples

```
Inductive le : nat -> nat -> Prop :=
    le_O : forall n : nat, O <= n
|   le_S : forall n m : nat, m <= n -> S m <= S n
```

m is a strict subterm of S m

Examples

```
Inductive le : nat -> nat -> Prop :=
   le_O : forall n : nat, O <= n
|  le_S : forall n m : nat, m <= n -> S m <= S n
```

✓

m is a strict subterm of S m

Examples

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A)
                        : Prop :=
   Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

Examples

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A)
                    : Prop :=
   Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

y is not related to x

Examples

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A)
                    : Prop :=
   Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

y is not related to x

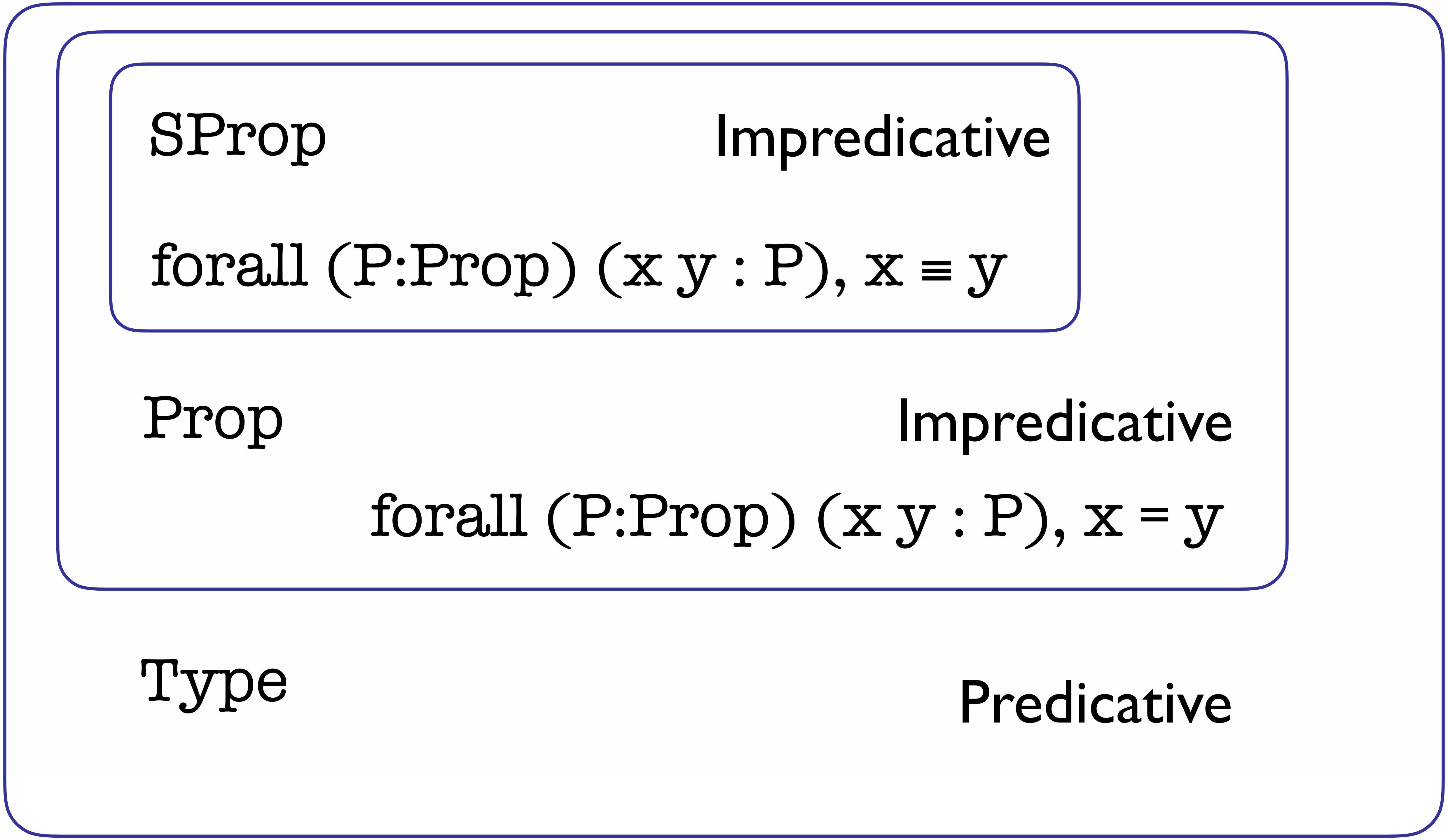
Remark

This syntactic characterisation of mere propositions is incomplete as for instance singleton types are not accepted.

This is somehow a good point because allowing singleton types in a definitional proof-irrelevant universe implies UIP *(Peter L.L.)*.

The Big Picture

The Big Picture



SProp Impredicative

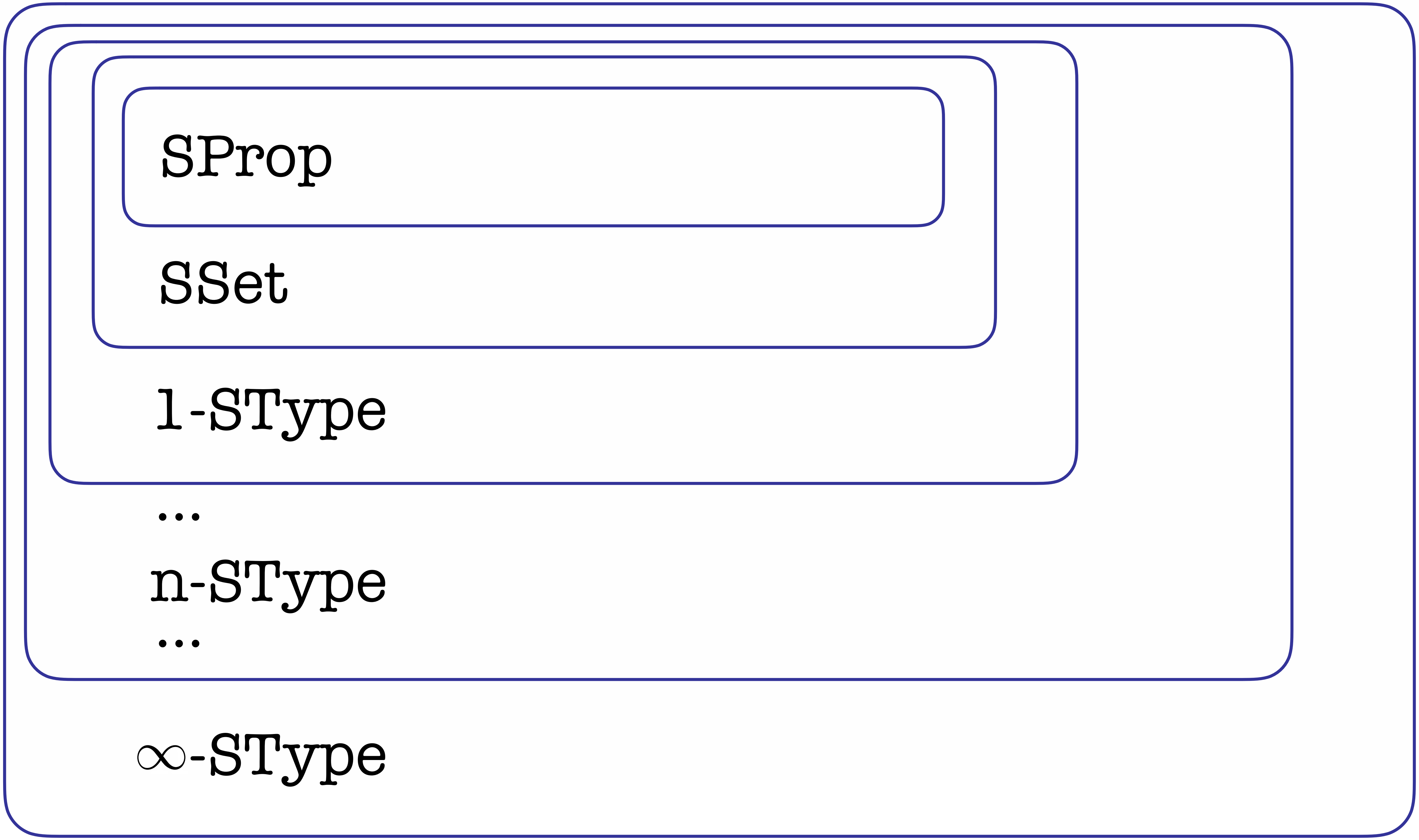
forall (P:Prop) (x y : P), x ≡ y

Prop Impredicative

forall (P:Prop) (x y : P), x = y

Type Predicative

Getting High(er) ?



SProp

SSet

1-SType

…

n-SType

…

∞-SType

V.V. has already sketched this in 2006!

$$\begin{tikzcd}
U_{0,0} \arrow[r, equal] & U_{1,0} \arrow[r, equal] \arrow[d] & U_{2,0} \arrow[r, equal] \arrow[d] & U_{3,0} \arrow[r, equal] \arrow[d] & \dots \\
 & U_{1,1} \arrow[r] & U_{2,1} \arrow[r] \arrow[d] & U_{3,1} \arrow[r] \arrow[d] & \dots \\
 & & U_{2,2} \arrow[r] & U_{3,2} \arrow[r] \arrow[d] & \dots \\
 & & & U_{3,3} \arrow[r] & \dots
\end{tikzcd}$$

A very short note on homotopy λ-calculus
Vladimir Voevodsky, 2006

Demo

Doggy bag

1. Prop can be turned into a syntactic approximation of mere propositions
2. To get definitional proof-irrelevance, we also need to restrict recursive types with a guard condition
3. This should be (hopefully) available soon in Coq
4. It may be extended to deal with a // hierarchy of universes that encodes for homotopy levels.