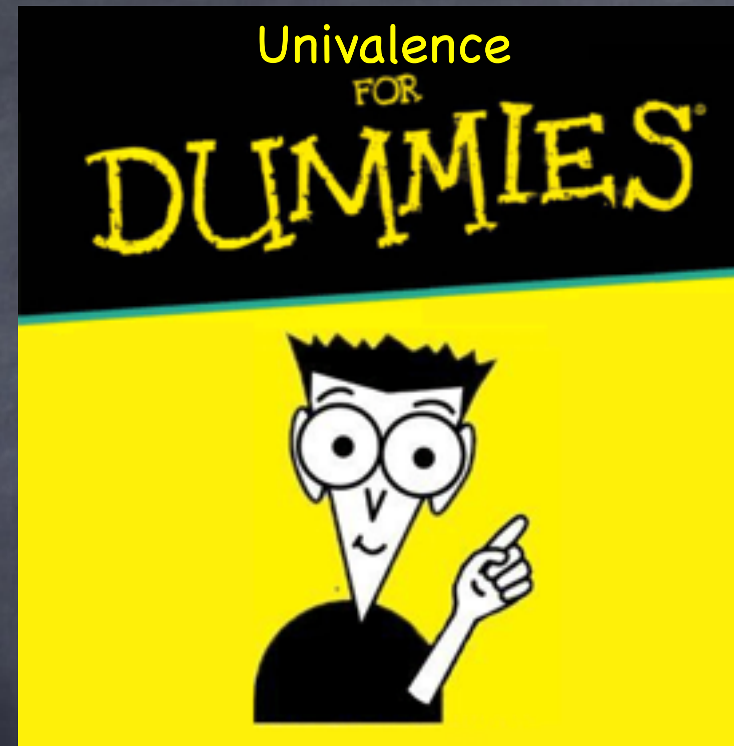

Thorsten Altenkirch
Functional Programming Laboratory
School of Computer Science

The University of Nottingham

- Email discussion with David McAllester
- Why do we believe that univalence is sound?
- Doesn’t understand the simplicial set model.

The Simplicial Model of Univalent Foundations
Chris Kapulkin, Peter LeFanu Lumsdaine, Vladimir Voevodsky

Triggered by his morphoid theory, comments on classical logic.
Why don’t you use HoTT?

Other alternatives: cubical – but not easy to explain univalence.
However: I realized that there is an intuitive explanation for a limited form of univalence.

Groupoids!

July 3, 2015, Friday
TLCA Invited Talk
(chair: Peter Dybjer)
9:00: Martin Hofmann.
The Groupoid Interpretation of Type Theory, a Personal Retrospective

Sets with Isomorphism are a groupoid.
This is a trivial way to construct a univalent universe of sets.

Suggested to Martin to talk about groupoids at TLCA.

- Groupoids : univalent universe of sets
- Setoids : univalent universe of propositions
- Can we do 2-groupoids?
- This gets complicated !

There is an even simpler example: Prop with <-> are a setoid.
This gives rise to a univalent universe of propositions in Setoids.

Can we move up step by step?
2-groupoids already look very hard.
Better check this on a computer, which gets us to another talk…

Type Theory eats itself without indigestion

joint work with Ambrus Kaposi

Thorsten Altenkirch
Functional Programming Laboratory
School of Computer Science
The University of Nottingham

Type Theory in Type Theory?

Type Theory should eat itself
James Chapman, LFMTP 2008

A Formalisation of a
Dependently Typed Language
as an Inductive-Recursive Family
Nils Anders Danielsson
TYPES 2006

- No pre-terms ! Only typed objects.
- Verified Metatheory
- Template Type Theory

Guilhem Jaber, Nicolas Tabareau,
Matthieu Sozeau.
Extending Type Theory with Forcing.
LICS 2012

Previous work by Chapman and Danielsson.
Looking for a more canonical and concise approach.

Main motivation for me verified metatheory.

Template programming: extend your programming language by new constructs.
Can be applied to Type Theory.
Paper by Jaber,Tabareau & Sozeau demonstrates this idea on presheaf models.
Eg add principles of guarded type theory.

```
data Ty : Set where
  ι    : Ty
  _⇒_ : Ty → Ty → Ty

data Con : Set where
  •   : Con
  _,_ : Con → Ty → Con

data Var : Con → Ty → Set where
  zero : Var (Γ , σ) σ
  suc  : Var Γ σ → Var (Γ , τ) σ

data Tm :  Con → Ty → Set where
  var : Var Γ σ → Tm Γ σ
  _$_ : Tm Γ (σ ⇒ τ) → Tm Γ σ → Tm Γ τ
  ƛ   : Tm (Γ , σ) τ → Tm Γ (σ ⇒ τ)
```

Simply Typed λ-calculus

STL is easy. Just to demonstrate the idea that we restrict ourselves to typed objects.

Moving to dependent types much harder – so far.

OMITTED

- Substitution

```
_[_] : Tm Γ σ → Tms Γ Δ → Tm Γ σ
```

- $\beta\ \eta$ - Equality

```
data _~_ : Tm Γ σ → Tm Γ σ → Set
```

- Terms as quotient

```
Tm Γ σ / ~
```

We need to define substitution.
Explicit or Implicit. Explicit is usually better.
Really should consider terms as a quotient!

Dependent Types

```
data Con : Set
data Ty : Con → Set
data Tm : (Γ : Con) → Ty Γ → Set
data Tms : Con → Con → Set
```

For DTP we need to define Con, Ty , Tm mutually.

Also throw in Tms = Substitutions to use explicit substitutions.

Induction-Induction

```
_,_   : (Γ : Con) → Ty Γ → Con
```

```
_[_]T : Ty Δ → Tms Γ Δ → Ty Γ
```

```
_,_   : (δ : Tms Γ Δ){A : Ty Δ} → Tm Γ (A [ δ ]T) → Tms Γ (Δ , A)
```

A categorical semantics for inductive-inductive definitions
TA, Frederik Forsberg, Peter Morris and Anton Setzer
CALCO 2011

Even worse: constructor types mention other constructors.

Inductive - Inductive types.
add one constructor at a time. Use dialgebras.
Understand this better now.

Coerce

```
coe   : A ~Ty B → Tm Γ A → Tm Γ B
```

It is getting worse. Coercion rule shows that we also need to define ~ mutually.

Dependent Types II

```
data Con : Set
data Ty : Con → Set
data Tm : (Γ : Con) → Ty Γ → Set
data Tms : Con → Con → Set
data _~Con_ : Con → Con → Set
data _~Ty_ : Ty Γ → Ty Γ → Set
data _~Tm_ : Tm Γ A → Tm Γ A → Set
data _~Tms_ : Tms Γ Δ → Tms Γ Δ → Set
```

This starts looking a bit ugly!

Boilerplate

- ~s are equivalence relations
- constructors are congruences
- Ty, Tm, Tms are families of setoids

But it is much worse. We need to say that all families are functorial wrt all their indices.

Can’t see the content of too much boilerplate.
Maybe we can automatically generate the boilerplate. But better..

I was told not to mention housewives in my talk.

But I couldn’t help it.

Higher Inductive Types (HITs) to the rescue

```
data S¹ : Set where
  base : S¹
  loop : base ≡ base
```

- HITs which are sets can be useful.
- Quotient Inductive Types (QITs)
- Examples in the HoTT book:
  - Cauchy reals (11.3)
  - Cumulative hierarchy of sets (10.5)

The infinite tree example

```
data T₀ : Set where
  leaf : T₀
  node : (ℕ → T₀) → T₀
```

```
data _~_ : T₀ → T₀ → Set where
  leaf : leaf ~ leaf
  node : (∀ {n} → f n ~ g n) → node f ~ node g
  perm : isIso f → node g ~ node (g ∘ f)
```

```
T = T₀ / _~_
```

Define !

```
nodeT : (ℕ → T) → T
```

```
[ node f ] ≡ nodeT (λ i → [ f i ])
```

You can lift the constructors to finite trees by sequencing the eliminator.

But there seems to be no general way to do this for infinite trees.
The problem boils down to comute function types and equivalence classes.
This corresponds to instances of the axiom of choice (not provable in HoTT).

Infinite trees as a QIT

```
data T : Set where
  leaf : T
  node : (ℕ → T) → T
  perm : isIso f → node g ≡ node (g ∘ f)
  isSet  : {e0 e1 : u ≡ v} → e0 ≡ e1
```

Using HITs there is an easy way out.

Here also force this to be a set.
Omitted in subsequent examples.

Dependent types as a QIIT

```
data Con where
  •     : Con
  _,_   : (Γ : Con) → Ty Γ → Con

data Ty where
  _[_]T : Ty Δ → Tms Γ Δ → Ty Γ
  U     : Ty Γ
  El    : (A : Tm Γ U) → Ty Γ
  Π     : (A : Ty Γ)(B : Ty (Γ , A)) → Ty Γ

data Tms where
  ε    : Tms Γ •
  _,_  : (δ : Tms Γ Δ) → Tm Γ (A [ δ ]T) → Tms Γ (Δ , A)
  id   : Tms Γ Γ
  _∘_  : Tms Δ Σ → Tms Γ Δ → Tms Γ Σ
  π₁   : Tms Γ (Δ , A) →  Tms Γ Δ

data Tm where
  _[_]t : Tm Δ A → (δ : Tms Γ Δ) → Tm Γ (A [ δ ]T)
  π₂    : (δ : Tms Γ (Δ , A)) → Tm Γ (A [ π₁ δ ]T)
  app   : Tm Γ (Π A B) → Tm (Γ , A) B
  lam   : Tm (Γ , A) B → Tm Γ (Π A B)
```

```
[id]T : A [ id ]T ≡ A
[][]T : (A [ δ ]T) [ σ ]T ≡ A [ δ ∘ σ ]T
U[]   : U [ δ ]T ≡ U
El[]  : El A [ δ ]T ≡ El (coe (TmΓ= U[]) (A [ δ ]t))
Π[]   : (Π A B) [ δ ]T ≡ Π (A [ δ ]T) (B [ δ ^ A ]T)
```

```
idl    : id ∘ δ ≡ δ
idr    : δ ∘ id ≡ δ
ass    : (σ ∘ δ) ∘ ν ≡ σ ∘ (δ ∘ ν)
,∘     : (δ , a) ∘ σ ≡ (δ ∘ σ) , coe .. (a [ σ ]t)
π_1β   : π_1 (δ , a) ≡ δ
πη     : (π_1 δ , π_2 δ) ≡ δ
εη     : {σ : Tms Γ •} → σ ≡ ε
```

```
[id]t : t [ id ]t ≡[ [id]T ]≡ t
[][]t : (t [ δ ]t) [ σ ]t ≡[ [][]T ]≡  t [ δ ∘ σ ]t
π_2β   : π_2 (δ , a) ≡[ π_1β ]≡ a
lam[] : (lam t) [ δ ]t ≡[ Π[] ]≡ lam (t [ δ ^ A ]t)
Πβ     : app (lam t) ≡ t
Πη     : lam (app t) ≡ t
```

The Recursor

```
record Motives : Set₁ where
  field
    Conᴹ : Set
    Tyᴹ  : Conᴹ → Set
    Tmsᴹ : Conᴹ → Conᴹ → Set
    Tmᴹ  : (Γᴹ : Conᴹ) → Tyᴹ Γᴹ → Set
```

```
record Methods (M : Motives) : Set₁ where
  field
    •ᴹ     : Conᴹ
    _,Cᴹ_  : (Γᴹ : Conᴹ) → Tyᴹ Γᴹ → Conᴹ
    ...
```

```
module rec (M : Motives)(m : Methods M) where

  Con-elim : Con → Conᴹ
  Ty-elim  : (A : Ty Γ) → Tyᴹ (Con-elim Γ)
  Tms-elim : (δ : Tms Γ Δ)→ Tmsᴹ(Con-elim Γ)(Con-elim Δ)
  Tm-elim  : (t : Tm Γ A) → Tmᴹ (Con-elim Γ) (Ty-elim A)
```

Motives + Methods

=

Algebras

=

Models of TT

Set theoretic model

Problem: Set is not a set!

```
data UU : Set
EL : UU → Set

data UU where
  'Π' : (A : UU) → (EL A → UU) → UU
  'Σ' : (A : UU) → (EL A → UU) → UU
  '⊤' : UU

EL ('Π' A B) =   (x : EL A) → EL (B x)
EL ('Σ' A B) = Σ (EL A) λ x → EL (B x)
EL '⊤' = ⊤
```

```
M : Motives
M = record
      { Conᴹ =                 UU
      ; Tyᴹ  = λ ⟦Γ⟧      → EL ⟦Γ⟧ → UU
      ; Tmsᴹ = λ ⟦Γ⟧ ⟦Δ⟧ → EL ⟦Γ⟧ → EL ⟦Δ⟧
      ; Tmᴹ  = λ ⟦Γ⟧ ⟦A⟧ → (γ : EL ⟦Γ⟧) → EL (⟦A⟧ γ)
      }
```

```
m : Methods M
m = record
      { •ᴹ     =              '⊤'
      ; _,Cᴹ_  = λ ⟦Γ⟧ ⟦A⟧ → 'Σ' ⟦Γ⟧ ⟦A⟧

      ...

      ; [id]Tᴹ = refl
      ; [][]Tᴹ = refl

      ...
```

```
⟦_⟧C : Con → UU
⟦_⟧T : Ty Γ → EL (⟦ Γ ⟧C) → UU
⟦_⟧s : Tms Γ Δ → EL (⟦ Γ ⟧C) → EL (⟦ Δ ⟧C)
⟦_⟧t : (t : Tm Γ A) → (γ : EL (⟦ Γ ⟧C)) → EL (⟦ A ⟧T γ)
```

The logical predicate translation (almost finished)

- Inspired by JP Bernardy et al on parametricity for dependent types
- A syntactic translation assigning to
  - each context, an extended context
  - to every type, a logical predicate
  - to every term, a proof that the term satisfies the logical predicate.
  - requires dependent eliminator

Eg Parametricity and dependent types
JP Bernardy; P Jansson R Paterson, ICFP 2010

This doesn’t include “internal parametricity” from 2012, joint with Moulin.

The presheaf interpretation (started)

- Fix a category C
- Contexts are interpreted as presheaves
- Types as families of presheaves
- Substitutions are natural transformations
- Terms are global sections

Normalisation by evaluation

- Normal forms are a presheaf over the category of variable substitutions.
- We can generalise NBE from simple types to dependent types.
- However, the normal forms have types which are not normal.

Compare to
Categorical reconstruction of a reduction free normalization proof
TA, M.Hofmann, T Streicher
CTCS 95

Normal forms with normal types?

- Can we define a mutual datatype of normal forms with normal types?
- No equations, no truncation!
- Use this to define semi-simplicial types?

- We need to define normalisation mutual with normal forms!
- Even in the simplest case (only variables) this leads to a new coherence problem!
- substitution is defined by recursion

joint work with Frederik Forsberg.

```
_[_]T : Ty Δ → Vars Γ Δ → Ty Γ
_[_]v : Var Δ A → (δ : Vars Γ Δ) → Var Γ (A [ δ ]T)

data Vars where
  ε     : Vars Γ •
  _,_   : (δ : Vars Γ Δ){A : Ty Δ} → Var Γ (A [ δ ]T)
             → Vars Γ (Δ , A)

wk : {A : Ty Γ} → Vars (Γ , A) Γ

data Var where
  vz : Var (Γ , A) (A [ wk ]T)
  vs : Var Γ A → Var (Γ , B) (A [ wk ]T)
```

Complete Failure !

Your goal was to model univalent universes but you can only eliminate into a set!

2-level theory

- Start with a strict type theory (with K)
- Introduce a universe with a univalent equality, can only eliminate into the universe.
- Syntax of Type Theory has to be defined in the strict theory
- However we can use the univalent universe to build models.