Slide 1

# A Taste of Intuitionistic Type Theory
# Part 3

**Midland Graduate School, 18.10.00 - 15.11.00**

`http://www.cs.nott.ac.uk/~txa/itt/`

**Thorsten Altenkirch**

**November 15, 2000**

Slide 2

## Set = Prop

- Avoid repetition of isomorphic structures on the propositional and set-theoretic level.
- Generalize non-dependent concepts to dependent ones to gain expressiveness.
- Internalize program extraction, be able to **derive** the axiom of choice.
- A **proposition** corresponds to the set of its proofs.
- A **set** is true if it is non empty.

**Slide 3**

## Judgements of Type Theory

| | |
|---|---|
| Typing | $M : \sigma$ |
| Definitional Equality | $M = N$ |

**Slide 4**

## Judgements of Type Theory (with contexts)

| | |
|---|---|
| Context validity | $\vdash \Gamma\ \mathbf{Cxt}$ |
| Typing | $\Gamma \vdash M : \sigma$ |
| Definitional Equality | $\Gamma \vdash M = N$ |

Slide 5

## The conversion rule

$$\frac{M : \sigma \qquad \sigma = \tau}{M : \tau}$$

Slide 6

## Set and Type

- **Set** = small sets
- **Type** = large sets or classes
- **Set** : **Type**
- Cumulativity

$$\frac{A : \mathbf{Set}}{A : \mathbf{Type}}$$

-

$$\kappa \in \{\mathbf{Type}, \mathbf{Set}\}$$

**Slide 7**

## $\Pi$-types

Generalize

- Implication $A \to B$,
- Non-dependent function space $A \to B$,
- Universal quantification $\forall x : A.B(x)$

**Slide 8**

## Rules for $\Pi$-types

$$\frac{A : \kappa \qquad \begin{array}{c}[x : A] \\ \vdots \\ B : \kappa\end{array}}{\{x : A\}B : \kappa}\,\Pi\text{-form} \qquad \frac{A : \kappa \qquad \begin{array}{c}[x : A] \\ \vdots \\ B : \kappa\end{array}}{\{x : A\}B : \kappa}\,\Pi\text{-intro}$$

$$\frac{f : \{x : A\}B \qquad a : A}{fa : B[x := a]}\,\Pi\text{-elim} \qquad \frac{}{([x : \sigma]M)N = M[x := N]}\,\Pi\text{-comp}$$

**Slide 9**

## Hiding parameters

Consider

$$
\begin{aligned}
f &: \{a : A\}(Ba) \to C \\
a &: A \\
b &: Ba \\
f\ a\ b &: C
\end{aligned}
$$

The first parameter can (usually) be inferred from the 2nd. Hence we write

$$
\begin{aligned}
f &: \{a|A\}(Ba) \to C \\
a &: A \\
b &: Ba \\
f\ b &: C
\end{aligned}
$$

**Slide 10**

## Σ-types

Generalize

- Conjunction $A \wedge B$,
- Non-dependent cartesian product $A \times B$,
- Existential quantification $\exists x : A.B(x)$

## Constants for Σ-types

Slide 11

```
$[Sigma : {A:Set}(A→Set)→Set];

$[pair : {A|Set}{B:A→Set}{a:A}(B a)→(Sigma A B)];

$[R_Sigma : {A|Set}{B|A→Set}{C:(Sigma A B)→Set}
                ({a:A}{b:B a}(C (pair B a b)))
            → {p:Sigma A B}C p];

[[A:Set][B:A→Set][C:(Sigma A B)→Set][a:A][b:B a]
 [h:{a:A}{b:B a}(C (pair B a b))]
   R_Sigma C h (pair B a b) ⟹ h a b
];
```

## Projections for Σ

Slide 12

```
$[fst[A|Set][B|A→Set][p:Sigma A B] =
        R_Sigma ([_:Sigma A B]A)
                ([a:A][b:B a]a) p : A];

$[snd[A|Set][B|A→Set][p:Sigma A B] =
        R_Sigma ([p:Sigma A B]B (fst p))
                ([a:A][b:B a]b) p : B (fst p)];
```

Slide 13

## The axiom of choice

```
[ac[A,B|Set][R:A→B→Set][p:{a:A}Sigma B ([b:B]R a b)] =
        pair ([f:A→B]{a:A}R a (f a))
           ([a:A]fst (p a)) ([a:A]snd (p a)) :
        Sigma (A→B) ([f:A→B]{a:A}R a (f a))];
```

Slide 14

## $\times$ and $\wedge$

```
[Prod[A,B:Set] = Sigma A ([_:A]B)];

[pair'[A,B|Set][a:A][b:B] = pair ([_:A]B) a b : Prod A B];

[fst'[A,B|Set][p:Prod A B] = fst p : A];

[snd'[A,B|Set][p:Prod A B] = snd p : B];
```

Slide 15

## A trivial Set

```
$[Unit : Set];

$[unit : Unit];

[R_Unit : C:Unit→Set(C unit)
                → x:UnitC x];

[[C:Unit→Set][h:C unit]
   R_Unit C h unit ⟹ h];
```

Slide 16

## +-types

Generalize

- Conjunction $A \vee B$,
- Disjoint union $A + B$,
- No dependent version!

Slide 17

## Disjoint union

```
$[Plus : Set→Set→Set];

$[inl : {A,B:Set}A→(Plus A B)];
$[inr : {A,B:Set}B→(Plus A B)];
$[Rec_Plus : {A,B|Set}{C:(Plus A B)→Set}
                ({a:A}(C (inl A B a)))
             → ({b:B}(C (inr A B b)))
             → {x:Plus A B}C x];

[[A,B:Set][C:(Plus A B)→Set][a:A][b:B]
 [h0:{a:A}(C (inl A B a))][h1:{b:B}(C (inr A B b))]
   Rec_Plus C h0 h1 (inl A B a) ⟹ h0 a
|| Rec_Plus C h0 h1 (inr A B b) ⟹ h1 b];
```

Slide 18

## The empty Set

```
$[Empty : Set];

$[R_Empty : {C:Empty→Set}
                {x:Empty}C x];
```

Slide 19

## The Identity Set

```
$[Id : {A|Set}A→A→Set];

$[refl : {A|Set}{a:A}Id a a];

$[Rec_Id : {A|Set}{C:{a,b|A}(Id a b)→Set}
                ({a:A}C (refl a))
             → {a,b|A}{p:Id a b}C p];

[[A:Set][C:{a,b|A}(Id a b)→Set][h:{a:A}C (refl a)][a:A]
   Rec_Id C h (refl a) ⟹ h a];
```

Slide 20

## Id is a *Groupoid*

```
[A|Set];

Goal {a,b|A}{p:Id a b}Id (transId' p (refl a)) p;
Goal {a,b|A}{p:Id a b}Id (transId' (refl b) p) p;
Goal {a,b,c,d|A}{p:Id a b}{q:Id b c}{r:Id c d}
        Id (transId' r (transId' q p))
           (transId' (transId' r q) p);

Goal {a,b|A}{p:Id a b}Id (transId' (symId p) p) (refl a);
```

Slide 21

## Is Id a singleton ?

```
Goal {a|A}{p:Id a a}Id p (refl a);
```

Slide 22

## Natural numbers

```
$[Nat : Set];
$[0 : Nat];
$[s : Nat → Nat]

$[Rec_Nat : {C:Nat→Set}(C 0)
        → ({n:Nat}(C n)→(C (s n)))
        → {n:Nat}(C n)];

[[C:Nat→Set][h0:C 0][h1:{n:Nat}(C n)→(C (s n))][n:Nat]
   Rec_Nat C h0 h1 0 ⟹ h0
|| Rec_Nat C h0 h1 (s n) ⟹ h1 n (Rec_Nat C h0 h1 n)];
```

Slide 23

## Example

```
[Rec_Nat'[A|Set][a:A][f:Nat→A→A][n:Nat] =
        Rec_Nat ([_:Nat]A) a f n];

[add = Rec_Nat' ([n:Nat]n)
         ([m:Nat][f:Nat → Nat][n:Nat]s (f n))];

[Even[m:Nat] = Sigma Nat ([n:Nat]Id m (add n n))];

[Odd[m:Nat] = Sigma Nat ([n:Nat]Id m (s (add n n)))];

Goal {n:Nat}(Even n)∨(Odd n);
```

Slide 24

## Inductive definitions

```
Inductive [Nat : Set]
Constructors
   [O : Nat]
   [s : Nat → Nat];

Inductive [BT : Set]
Constructors
   [r : BT]
   [sp : BT→BT→BT];

Inductive [Ord : Set]
Constructors
   [O' : Ord]
   [s' : Ord → Ord]
   [lim : (Nat → Ord) → Ord];
```

Slide 25

## Example: nth

```
Inductive [Vec : Nat → Set]
Parameters [A|Set]
Constructors
        [vnil : Vec 0]
        [vcons : n:NatA→(Vec n)→(Vec (s n))];

[hd_Vec[n|Nat][as:Vec|A (s n)] =
  Vec_elim ([n:Nat][_:Vec|A n](not (Id 0 n))→A)
   ([H:not (Id 0 0)]Rec_Empty' A (H (refl 0)))
   ([n1:Nat][x1:A][_:Vec n1][_:(not (Id 0 n1))→A]
    [_:not (Id 0 (s n1))]x1)
   as (pa1 n) : A];
```

Slide 26

```
[tl_vec[n|Nat][as:Vec|A (s n)] =
  Vec_elim ([n'3:Nat][_:Vec|A n'3]
               m:Nat(Id n'3 (s m))→Vec|A m)
   ([m:Nat][H:Id 0 (s m)]Rec_Empty' (Vec|A m) (pa1 m H))
   ([n1:Nat][_:A][x2:Vec n1]
    [_:m:Nat(Id n1 (s m))→Vec|A m][m:Nat]
    [H:Id (s n1) (s m)]subst (Vec|A) (pa2 n1 m H) x2)
   as n (refl (s n))];

Discharge A;
```

Slide 27

```
Inductive [Fin : Nat → Set] ElimOver Type
Constructors
        [O_Fin : n:NatFin (s n)]
        [s_Fin : n|Nat(Fin n)→(Fin (s n))];

[A|Set];
[nth[n|Nat][i:Fin n][as:Vec|A n] =
        Fin_elim ([n:Nat][_:Fin n](Vec|A n)→A)
           ([n:Nat][v:Vec|A (s n)]hd_Vec v)
           ([n|Nat][_:Fin n][h:(Vec|A n)→A]
                [v:Vec|A (s n)]h (tl_Vec v))
        i as];
Discharge A;
```

Slide 28

## Pattern matching

```
$[div2 : Nat → Nat];

[[n:Nat]
   div2 0 ⟹ 0
|| div2 (s 0) ⟹ 0
|| div2 (s (s n)) => s (div2 n)];

$[add : Nat → Nat → Nat];

[[m,n:Nat]
   add 0 n ⟹ n
|| add (s m) n ⟹ s(add m n)]
```

Slide 29

```
[A|Set]

$[hd_Vec : n|Nat(Vec|A (s n))→A];

[[n:Nat][a:A][as:Vec|A n]
   hd_Vec (vcons a as) ⟹ a];

$[nth : n|Nat(Vec|A n)→(Fin n)→A];

[[n:Nat][i:Fin n][as:Vec|A n][a:A]
   nth (vcons a as) (0_Fin n) ⟹ a
|| nth (vcons a as) (s_Fin i) ⟹ nth as i];

Discharge A;
```

Slide 30

## Exercises

1. Define a function which maps finite numebrs into the natural number with the same value

   `fin2Nat : {n|Nat}(Fin n)→Nat`

   using pattern matching. Then define

   `emb : {n|Nat}(Fin n)→(Fin (S n))`

   which preserves the value. This can be verified by showing

   `embOk : {n|Nat}{i:Fin n}Id (fin2Nat i) (fin2Nat (emb i))`

**Slide 31**

2. Consider the following inductive definition of the relation $\leq$:

```
Inductive [Le:Nat->Nat->Prop]
Constructors
        [le0 : n:NatLe 0 n]
        [leS : m,n|Nat(Le m n)->(Le (S m) (S n))];
```

Establish (using pattern matching or the eliminators) that `Le` is a partial order, i.e.

```
leRefl : {n:Nat}Le n n
leTrans : {i,j,k|Nat}(Le i j)→(Le j k)→(Le i k)
leAsym : {i,j|Nat}(Le i j)→(Le j i)→(Id i j)
```

To show antisymmetry it may be helpful to establish the following inversion lemmas first:

```
leInv0 : {i|Nat}(Le i 0)→(Id i 0)
leInv1 : {i,j|Nat}(Le (S i) (S j))→ (Le i j)
```

**Slide 32**

3. Show that `Le` is decidable:

```
leDec : {i,j|Nat}(Le i j) ∨ (Not (Le i j))
```

How does this give rise to a decision procedure?

**Slide 33**

4. We say that an element is accessible wrt a relation if the set of smaller elements is always finite. This can be expressed by the following inductive definition:

```
Inductive [Acc : A→Set]
Parameters [A|Set][R:A→A→Set]
Constructors
   [acc : {a:A}({b:A}(R b a)→(Acc b))→(Acc a)];
```

Show that all natural numbers are accessible wrt. $<$ which can be defined as

```
Inductive [Lt:Nat→Nat→Prop]
Constructors
        [lt0 : {n:Nat}Le O (S n)]
        [ltS : {m,n|Nat}(Lt m n)→(Lt (S m) (S n))];
```

That is establish:

```
leAcc : {i:Nat}Acc Le i
```