Inductive-Recursive Definitions

Anton Setzer
(Swansea University; joint work with Peter Dybjer)

(Cambridge, Logic and Semantics Seminar, May 2009)

1. Dependent Type Theory.
2. Sets in Martin-Löf Type Theory and Principles of Ind.-Rec.
3. Closed Formalisation of Induction-Recursion.
4. Results.

Goal of this Talk

- Define an extension of Martin-Löf Type Theory (MLTT) which allows to define all types definable in standard extensions of MLTT without any encoding.
- Gives rise to a proof theoretically very strong extension of positive inductive definitions.
- New principle where we define

$$\mathrm{T} : \mathrm{U} \rightarrow \mathrm{Set}$$

in such a way that the domain U of T depends on T.

New Grant

- Induction-Recursion topic of an EPSRC grant involving Neil Ghani, Peter Hancock (Glasgow Strathclyde), Thorsten Altenkirch (Nottingham) and A. S. (Swansea).

1. Dep. Type Theory

- **Dependent type theory** (version used here: **Martin-Löf Type Theory**) is functional programming based on dependent types.
- Set will in the following denote a "small type".
- Most types used in ordinary programming languages are simple types (no dependencies):
  - $\mathrm{String} : \mathrm{Set}$,
  - $\mathrm{Integer} : \mathrm{Set}$,
  - $\mathrm{Integer} \to \mathrm{Integer} : \mathrm{Set}$,
  - etc.

Polymorphic Types

- Polymorphic types ("generics") allow types to depend on other types.
  - E.g.

$$\mathrm{List} : \mathrm{Set} \to \mathrm{Set} \quad ,$$

    $\mathrm{List}(A)$ = set of lists of elements of set A.

- Polymorphic types allow more generic programs.
  - One definition of a library for List
  - rather than defining this library for each of
    - $\mathrm{List}(\mathrm{Integer})$,
    - $\mathrm{List}(\mathrm{Char})$,
    - $\mathrm{List}(\mathrm{String})$,
    - $\cdots$

Dependent Types

- Dependent types allow types to depend on other types and elements of other types.
  - Simple examples:
    - The set of n**-tuples** of elements of A is

$$\mathrm{Tuple}(A, n)$$

where

$$\mathrm{Tuple} : \mathrm{Set} \to \mathbb{N} \to \mathrm{Set}$$

- Allows to define functions, the result type of which depends on the argument, e.g.

$$f : (n : \mathbb{N}) \to \mathrm{Tuple}(\mathbb{N}, n)$$

Examples of Dependent Types

- The set of $n \times m$-**matrices** (of some fixed set) is

$$\mathrm{Mat}(n, m)$$

  where

$$\mathrm{Mat} : \mathbb{N} \to \mathbb{N} \to \mathrm{Set}$$

- Matrix multiplication gets type

$$\begin{aligned} \mathrm{matmult} : (n, m, k : \mathbb{N}) \\ \to \mathrm{Mat}(n, m) \to \mathrm{Mat}(m, k) \to \mathrm{Mat}(n, k) \end{aligned}$$

Predicates as Dependent Types

- The predicate

$$\mathrm{Sorted} : \mathrm{List}(\mathbb{N}) \to \mathrm{Set}$$

  s.t. there exists $p : \mathrm{Sorted}(l)$ iff l is sorted is a dependent type.
- The set $(l : \mathrm{List}(\mathbb{N})) \times \mathrm{Sorted}(l)$ is the set of sorted lists.
- The set

$$(l : \mathrm{List}(\mathbb{N})) \to (l' : \mathrm{List}(\mathbb{N})) \times \mathrm{Sorted}(l') \times (\mathrm{EqElements}(l, l'))$$

  is the set of sorting functions on $\mathrm{List}(\mathbb{N})$.

Logical Framework

- Basic logic framework has 2 main constructions:
- The dependent function type $(x : A) \to B(x)$ for $A : \mathrm{Set}$, $x : A \Rightarrow B(x) : \mathrm{Set}$.
  - Elements are roughly speaking

$$\{f : A \to \bigcup_{x:A} B(x) \mid \forall x \in A. f(x) \in B(x)\}$$

  - $A \to B$ is the special case $(x : A) \to B$ where B does not depend on x.
- The dependent product $(x : A) \times B(x)$ for $A : \mathrm{Set}$, $x : A \Rightarrow B : \mathrm{Set}$.
  - Elements are roughly speaking

$$\{\langle a, b \rangle \mid a \in A, b \in B(a)\}$$

Set vs. Type

- We will use two type levels.
  - $\underset{\sim}{\mathrm{Set}}$, the type of sets = small types.
  - $\underset{\sim}{\mathrm{Type}}$ the collection of big types.
  - $\mathrm{Set} \subseteq \mathrm{Type}$, $\mathrm{Set} : \mathrm{Type}$.
  - Type closed under (dependent) functions and products, but in the simplest version under nothing else.
    - So we have for instance, if $A : \mathrm{Set}$, then

$$A \to \mathrm{Set} : \mathrm{Type}$$

      type of **predicates over A.**
- Higher hierarchies are considered.
  Universes provide a much more powerful type hierarchy.

2. Sets in MLTT and Ind.-Rec.

- Simples type = type of Booleans.
- **Formation rule:**

$$\mathrm{Bool} : \mathrm{Set}$$

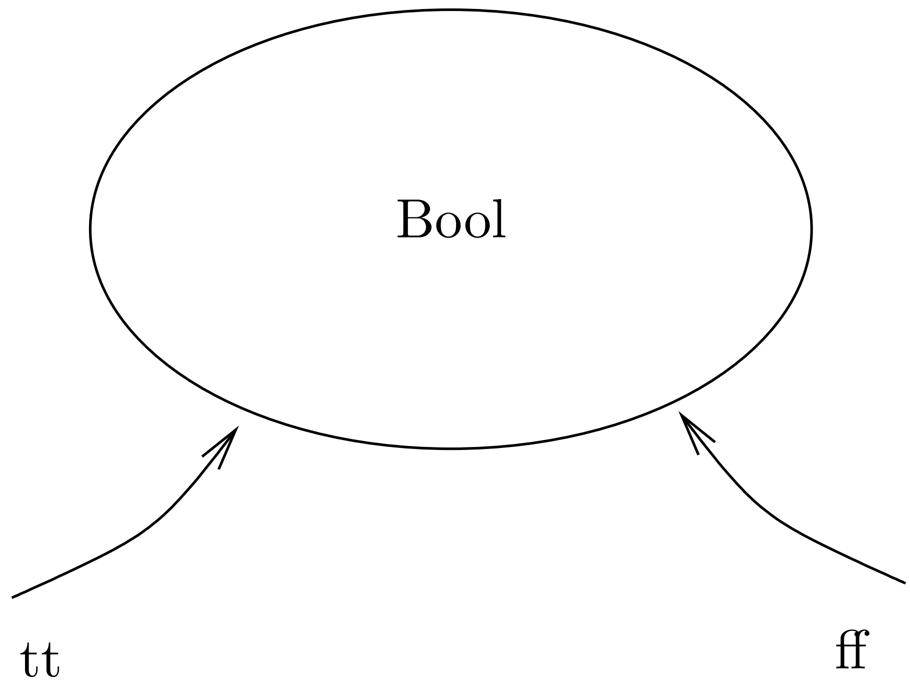
- **Introduction rules:**

$$\mathrm{tt} : \mathrm{Bool} \qquad \mathrm{ff} : \mathrm{Bool}$$

- **Elimination/equality rules:**

  If then else.

Visualisation of Bool



2 Constructors, both **no arguments**.

The Disjoint Union

- **Formation rule:**

$$\frac{A : \mathrm{Set} \qquad B : \mathrm{Set}}{A + B : \mathrm{Set}}$$

- **Introduction rules:**

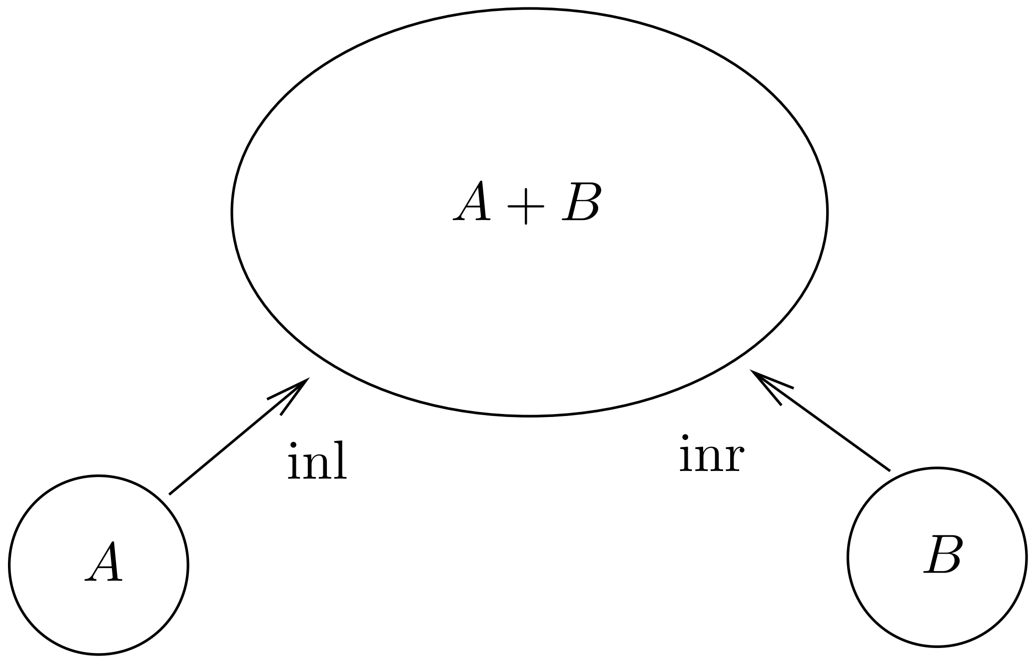
$$\mathrm{inl} : A \to (A + B)$$
$$\mathrm{inr} : B \to (A + B)$$

(Additional premises of formation rule suppressed).

- **Elimination/equality rule:**

$$\begin{array}{l} \mathrm{case}\ x\ \mathrm{of} \\ \qquad \{\ \mathrm{inl}(a) \ \to\ \cdots \\ \qquad\ \ \ \mathrm{inr}(b) \ \to\ \cdots \} \end{array}$$

Visualisation of A+B



- Both inl and inr have **one non-inductive argument**.

The Σ-Type

- **Formation rule:**

$$\frac{A : \mathrm{Set} \qquad B : A \to \mathrm{Set}}{\Sigma(A, B) : \mathrm{Set}}$$

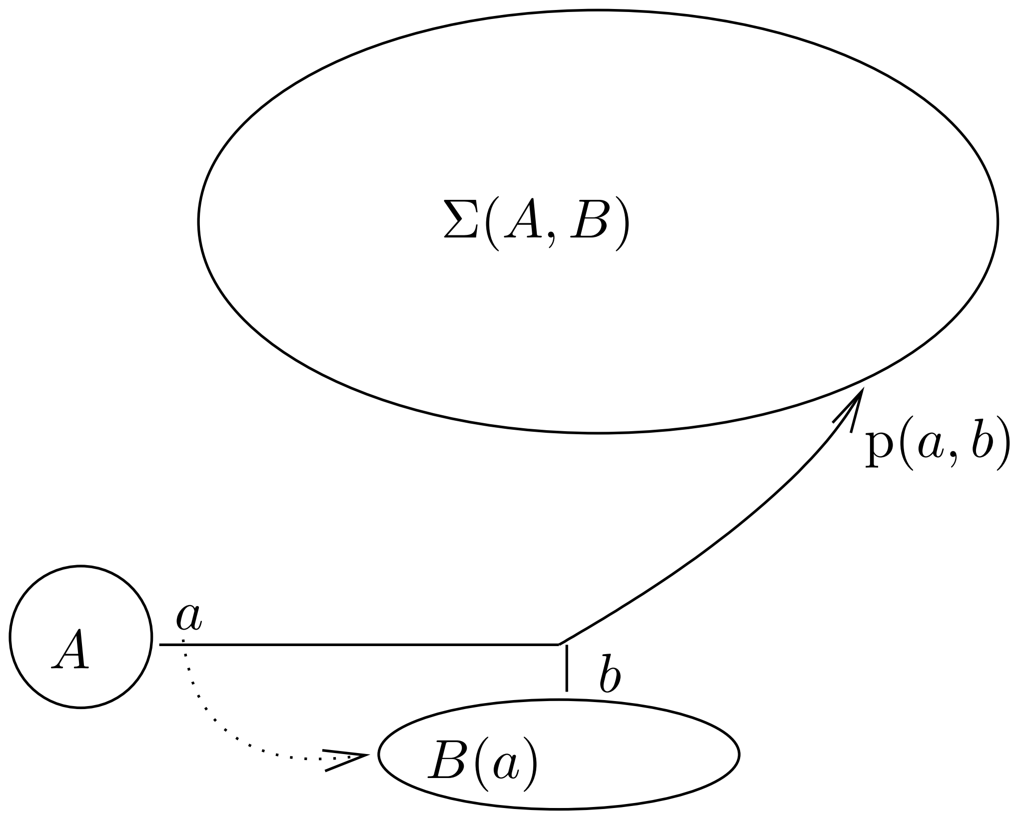
- **Introduction rule:**

$$\frac{a : A \qquad b : B(a)}{\mathrm{p}(a, b) : \Sigma(A, B)}$$

- **Elimination/equality rule:**

$$\begin{array}{l} \mathrm{case}\ x\ \mathrm{of} \\ \qquad \{\ \mathrm{p}(a, b) \ \to \ \cdots \} \end{array}$$

Visualisation of Σ(A,B)



- p has **2 non-inductive arguments**.
- The type of the 2nd argument **depends** on the the 1st argument.

Natural numbers

- **Formation rule:**

$$\mathbb{N} : \mathrm{Set}$$

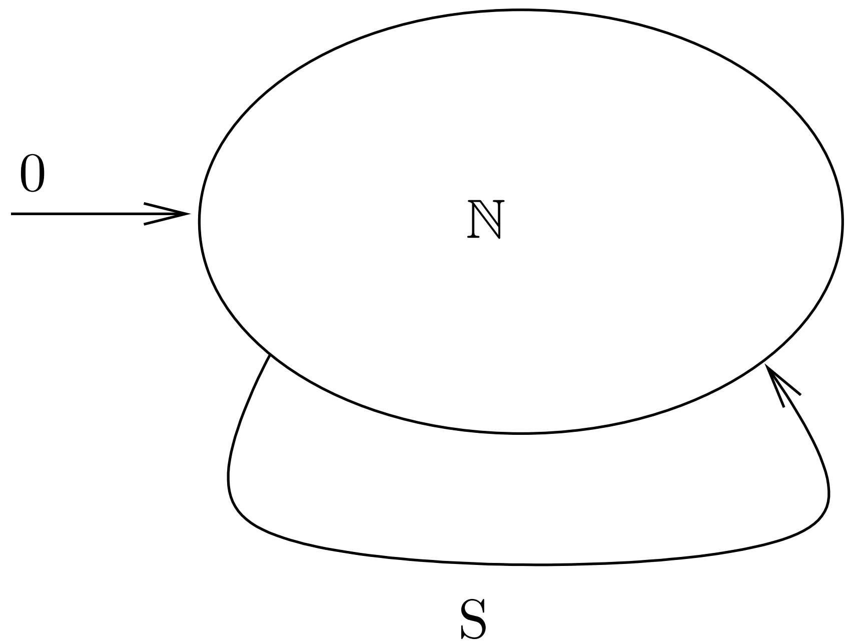
- **Introduction rules:**

$$0 : \mathbb{N} \qquad \frac{n : \mathbb{N}}{\mathrm{S}(n) : \mathbb{N}}$$

- **Elimination/equality rule:**

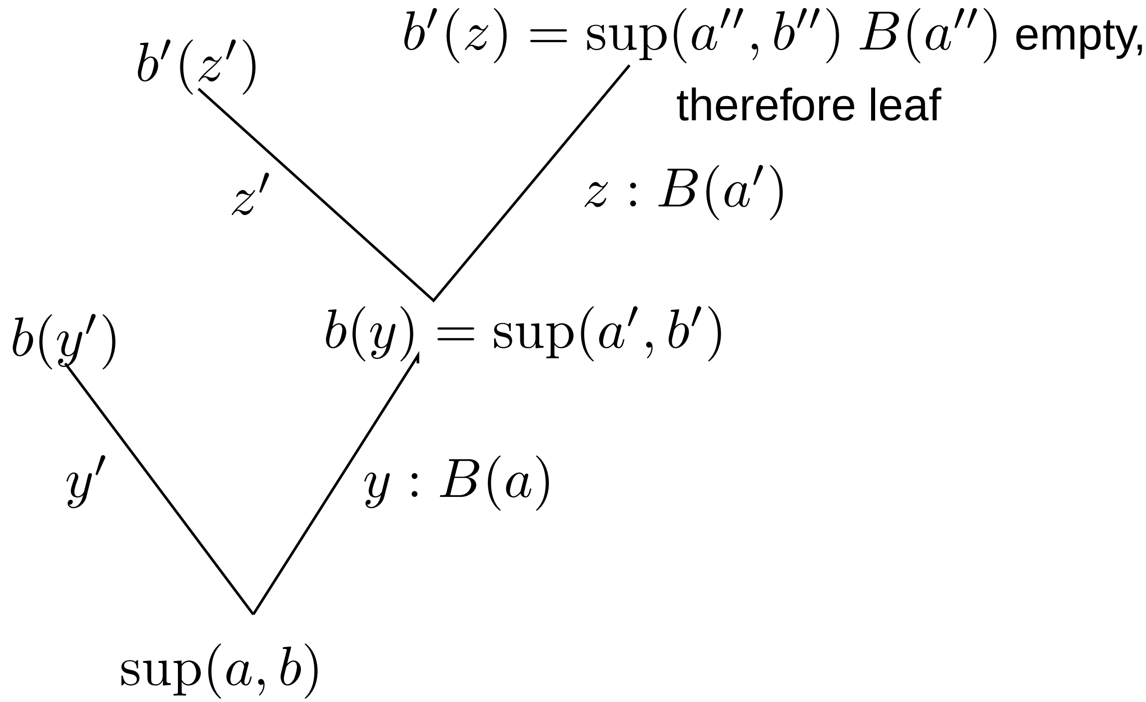
  Induction/primitive recursion.

Visualisation of $\mathbb{N}$



- 0 has **no arguments**.
- S has one **inductive argument**.

W-Type



Assume $A : \mathrm{Set}$, $B : A \to \mathrm{Set}$.
$\mathrm{W}(A, B)$ is the type of **well-founded recursive trees** with branching degrees $(B(a))_{a:A}$.

The W-Type

- **Formation rule:**

$$\frac{A : \mathrm{Set} \qquad B : A \rightarrow \mathrm{Set}}{\mathrm{W}(A, B) : \mathrm{Set}}$$

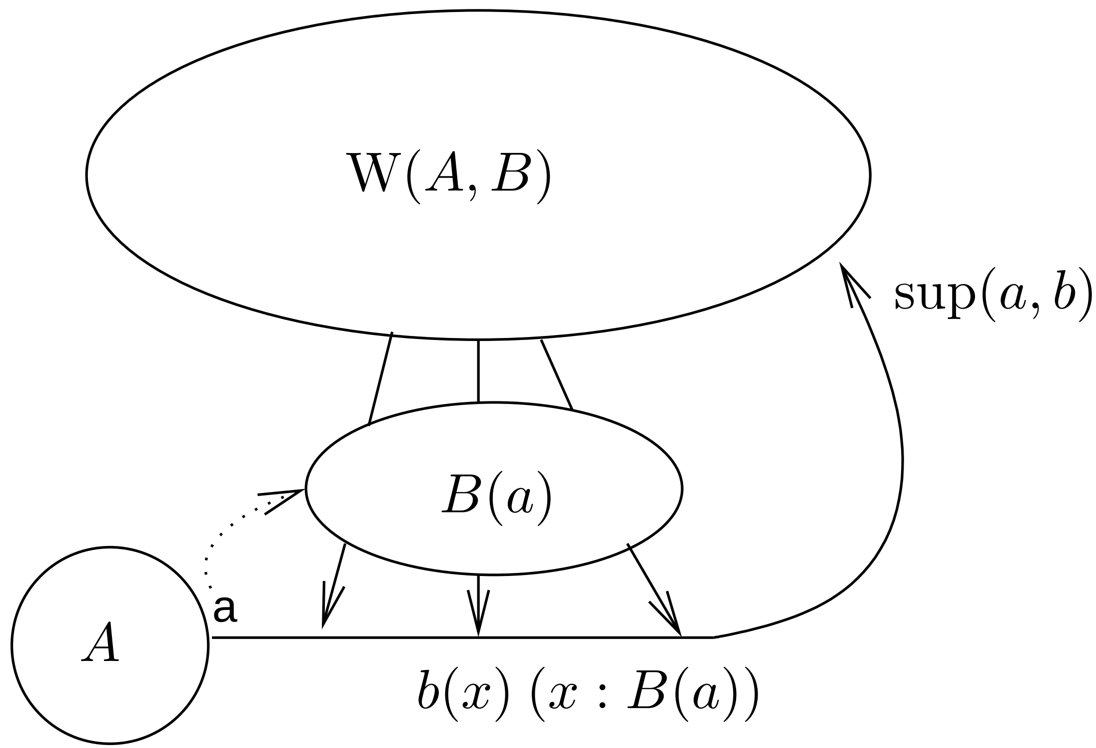
- **Introduction rule:**

$$\frac{a : A \qquad b : B(a) \rightarrow \mathrm{W}(A, B)}{\mathrm{sup}(a, b) : \mathrm{W}(A, B)}$$

- **Elimination/equality rule:**

  Induction over trees.

Visualisation of W(A,B)



sup has **2 arguments**:

- First argument is **non-inductive**.
- Second argument is **inductive**, indexed over $B(a)$.
- $B(a)$ **depends on the first argument a**.

Universes

- A universe is a family of sets
- Given by
  - a set $\mathrm{U} : \mathrm{Set}$ of **codes** for sets,
  - a **decoding function** $\mathrm{T} : \mathrm{U} \to \mathrm{Set}$.

Universes

- **Formation rules:**

$$\mathrm{U} : \mathrm{Set} \qquad \frac{a : \mathrm{U}}{\mathrm{T}(a) : \mathrm{Set}}$$

- **Introduction and Equality rules:**

$$\widehat{\mathbb{N}} : \mathrm{U} \qquad \mathrm{T}(\widehat{\mathbb{N}}) = \mathbb{N}$$

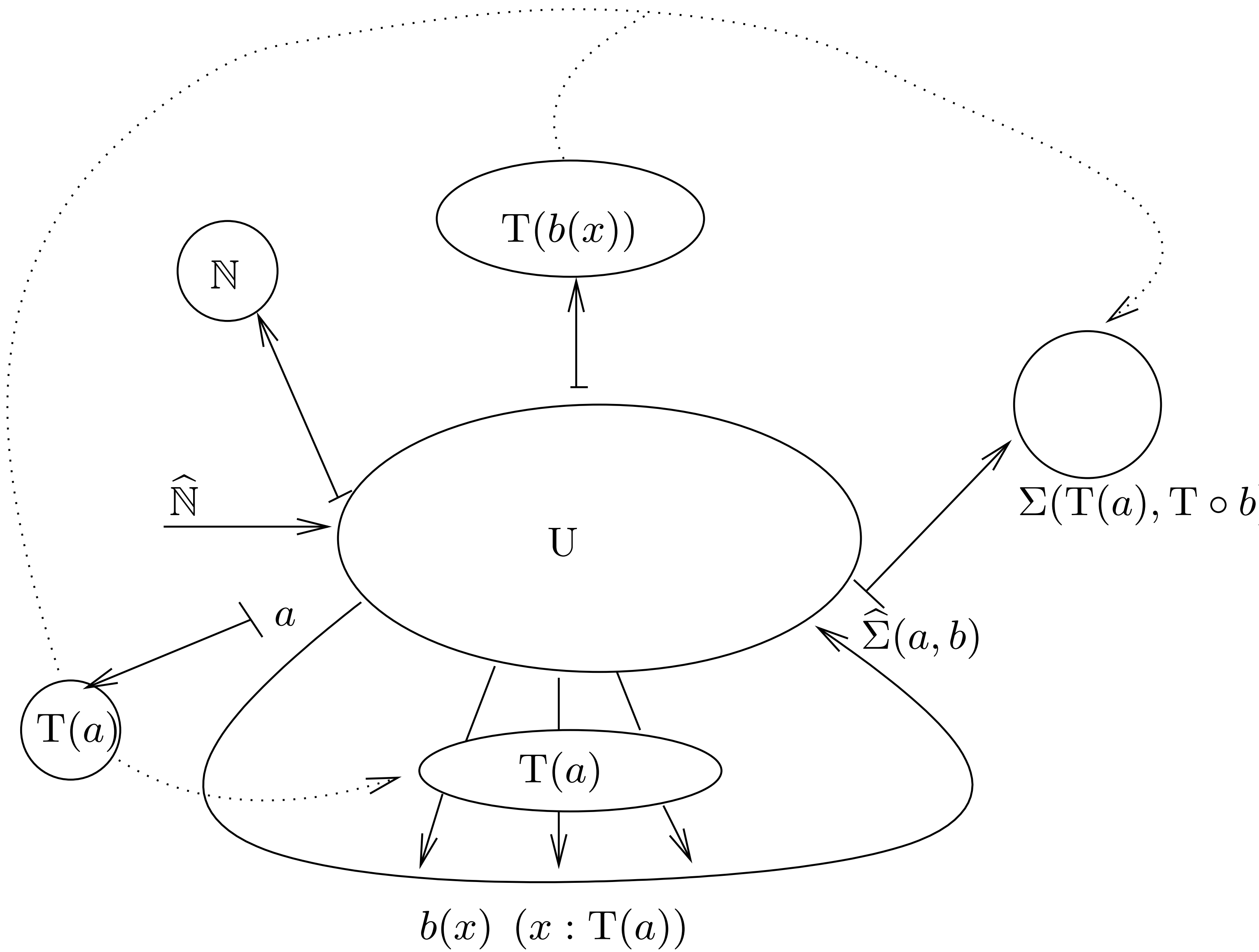
$$\frac{a : \mathrm{U} \qquad b : \mathrm{T}(a) \to \mathrm{U}}{\widehat{\Sigma}(a, b) : \mathrm{U}}$$

$$\mathrm{T}(\widehat{\Sigma}(a, b)) = \Sigma(\mathrm{T}(a), \mathrm{T} \circ b)$$

Similarly for other type formers (except for U).

- **Elimination/equality rules:** Induction over U.

Visualisation of U



Analysis

- Elements of U are defined **inductively**, while defining $\mathrm{T}(a)$ for $a : \mathrm{U}$ **recursively**.
- $\widehat{\Sigma}$ has two **inductive arguments**
  - Second argument depends on **T(a)**.
    - $\mathrm{T}(a)$ **depends** on T applied to first argument a.
  - $\mathrm{T}(\widehat{\Sigma}(a, b))$ **is defined from**
    - $\mathrm{T}(a)$.
    - $\mathrm{T}(b(x))$ ($x : \mathrm{T}(a)$).
- Principles for defining a universe can be generalised to **higher type universes**, where $\mathrm{T}(a)$ can be an element of any type, e.g. $\mathrm{Set} \rightarrow \mathrm{Set}$.

Advanced Example

- Set of lists of natural numbers with distinct elements.
- Inductive-recursive definition of
  - $\mathrm{Freshlist} : \mathrm{Set}$
  - $\_ \# \_ : \mathrm{Freshlist} \to \mathbb{N} \to \mathrm{Set}$.
- Constructors:

$$\mathrm{nil} : \mathrm{Freshlist} \quad ,$$

$$\mathrm{nil} \# m = \top$$

$$\mathrm{cons} : (n : \mathbb{N}, l : \mathrm{Freshlist}, l \# n) \to \mathrm{Freshlist}$$

$$\mathrm{cons}(n, l, p) \# m = (l \# m) \wedge (n \neq m)$$

3. Closed Formal. of Induct-Rec.

- The above constructions are examples of **inductive-recursive definitions**.
  - Many more sets can be defined in the same way.
- Inductive-recursive Definitions = general concepts which subsumes most standard extensions which have been found up to now.
  - Excludes Mahlo universe and similar constructions.
- Introduced originally by **Peter Dybjer** in a schematic way.
- Here: development of a rule based system, which allows to introduce all ind.-rec. def. by **finitely many rule schemes**.

Encoding of Constructors into one

- Several constructors can be **encoded into one** constructor:
  - Assume constructors $C_i : (a : A_i) \to U$ ($i = 1, \ldots, n$).
  - Replace them by one constructor $C : (i : \{1, \ldots, n\}, a : A_i) \to U$.
- Only required: finite sets.
  Will be part of the logical framework.

Induct. and Non-Induct. Arguments

- Two **kinds of arguments**:
  - **Non-inductive arguments**.
    - Refer to sets previously introduced.
  - **Inductive arguments**.
    - Refer to the set to be defined ind.-rec.
  - Additional **initial case**: constructors with no arguments.

Depend. of Args. on Prev. Ones

- Types of **later arguments** can **depend directly** on **previous non-inductive arguments**.
- Later arguments cannot depend directly on inductive arguments (since nothing is known about the ind.-rec. introduced set U).
  - However, they can **depend on T applied to inductive arguments**.
- Result of T applied to the constructed element can **depend in the same** way on arguments **as can later arguments depend on previous arguments**.

Formalisation

- We introduce inductive-recursively sets $\mathrm{U} : \mathrm{Set}$, $\mathrm{T} : \mathrm{U} \to D$ for some type D.
- Let $D : \mathrm{Type}$ be fixed.
  - In case of a standard **universe**

$$D = \mathrm{Set}$$

  - In case of **higher order universes**

$$D = \mathrm{Fam}(\mathrm{Set}) \to \mathrm{Fam}(\mathrm{Set})$$

    or higher types.
  - In case of inductive definitions (T is trivial)

$$D = \{*\}$$

OP_D

- We introduce a type of **codes for ind.-rec. definitions**:

$$\mathrm{OP}_D : \mathrm{Type}$$

- If $\gamma : \mathrm{OP}_D$, we introduce $(\mathrm{U}_\gamma, \mathrm{T}_\gamma)$ ind.-rec.:

$$\begin{array}{lcl} \mathrm{U}_\gamma & : & \mathrm{Set} \\ \mathrm{T}_\gamma & : & \mathrm{U}_\gamma \to D \end{array}$$

$\mathbf{F}^{\mathrm{U}}_{\gamma}$

- Further, we define the **set of arguments** of the constructor intro_{γ} of U_{γ}.
  - Argument set has to be defined, before U_{γ}, T_{γ} has been introduced.
  - Will be defined for arbitrary $U : \mathrm{Set}$, $T : U \to D$ $\gamma : \mathrm{OP}_D$

$$\mathrm{F}^{\mathrm{U}}_{\gamma} : (U : \mathrm{Set}) \to (T : U \to D) \to \mathrm{Set}$$

- **Introduction Rule for U_{γ}:**

$$\mathrm{intro}_{\gamma} : \mathrm{F}^{\mathrm{U}}_{\gamma}(\mathrm{U}_{\gamma}, \mathrm{T}_{\gamma}) \to \mathrm{U}_{\gamma}$$

$\mathrm{F}^{\mathrm{T}}_{\gamma}$

- Furthermore, we have to define the result of T_{γ} applied to $\mathrm{intro}_{\gamma}(a)$.
  - Again, we have to define it before the definition of U_{γ}, T_{γ} is finished.
- So we define

$$\mathrm{F}^{\mathrm{T}}_{\gamma} : (U : \mathrm{Set}) \to (T : U \to D) \to \mathrm{F}^{\mathrm{U}}_{\gamma}(U, T) \to D$$

- **Equality Rule for T_{γ}:**

$$\mathrm{T}_{\gamma}(\mathrm{intro}_{\gamma}(a)) = \mathrm{F}^{\mathrm{T}}_{\gamma}(\mathrm{U}_{\gamma}, \mathrm{T}_{\gamma}, a)$$

F_γ as a Functor

- We have

$$\begin{array}{lcl} \mathrm{F}^{\mathrm{U}}_\gamma & : & (U : \mathrm{Set}) \to (\mathrm{T} : U \to D) \to \mathrm{Set} \\ \mathrm{F}^{\mathrm{T}}_\gamma & : & (U : \mathrm{Set}) \to (\mathrm{T} : U \to D) \to \mathrm{F}^{\mathrm{U}}_\gamma(U, T) \to D \end{array}$$

- $\mathrm{F}^{\mathrm{U}}_\gamma$, $\mathrm{F}^{\mathrm{T}}_\gamma$ will form the object part of a functor

$$\mathrm{F}_\gamma : \mathrm{Fam}(D) \to \mathrm{Fam}(D)$$

where

$$\mathrm{Fam}(D) := (U : \mathrm{Set}) \times (\mathrm{U} \to D)$$

and $\langle \mathrm{U}_\gamma, \mathrm{T}_\gamma \rangle$ is the initial algebra of F_γ.
(Slight modification of the proof in the paper is needed.)

Elimin./Equal. Rules for U_γ, T_γ

- For elimination and equality rules similar functions $\mathrm{F}_\gamma^{\mathrm{IH}}$, $\mathrm{F}_\gamma^{\mathrm{map}}$ can be defined.
- Not treated here.

Initial Case

- Initial case for OP_D: No arguments.
  - We need only to define the result of T_γ applied to the constructor, i.e. require one element $\psi : D$.

$$\frac{\psi : D}{\mathrm{init}(\psi) : \mathrm{OP}_D}$$

$$\mathrm{F}^{\mathrm{U}}_{\mathrm{init}(\psi)}(U, T) = \{*\} : \mathrm{Set}$$

$$\mathrm{F}^{\mathrm{T}}_{\mathrm{init}(\psi)}(U, T, *) = \psi : D$$

Noninductive Argument

- For an noninductive argument we need to know
  - The set A, the argument is referring to.
  - Depending on A, the later arguments of the constructor, i.e. a function $\psi : A \to \mathrm{OP}_D$.

$$\frac{A : \mathrm{Set} \qquad \psi : A \to \mathrm{OP}_D}{\mathrm{nonind}(A, \psi) : \mathrm{OP}_D}$$

$$\mathrm{F}^{\mathrm{U}}_{\mathrm{nonind}(A,\psi)}(U, T) = (a : A) \times \mathrm{F}^{\mathrm{U}}_{\psi(a)}(U, T) : \mathrm{Set}$$

$$\mathrm{F}^{\mathrm{T}}_{\mathrm{nonind}(A,\psi)}(U, T, \langle a, b \rangle) = \mathrm{F}^{\mathrm{T}}_{\psi(a)}(U, T, b) : D$$

Inductive Argument

- For an inductive argument we need to know
  - The set A, over which the argument is indexed over.
    - $A = \{*\}$ give the special case of a single argument.
  - Depending on the result of T applied to the arguments of A, i.e. depending on $A \to D$, the later arguments of the constructor:
    We need a function $\psi : (A \to D) \to \mathrm{OP}_D$.

$$\frac{A : \mathrm{Set} \qquad \psi : (A \to D) \to \mathrm{OP}_D}{\mathrm{ind}(A, \psi) : \mathrm{OP}_D}$$

$$\mathrm{F}^{\mathrm{U}}_{\mathrm{ind}(A,\psi)}(U, T) = (a : A \to U) \times \mathrm{F}^{\mathrm{U}}_{\psi(T \circ a)}(U, T) : \mathrm{Set}$$

$$\mathrm{F}^{\mathrm{T}}_{\mathrm{ind}(A,\psi)}(U, T, \langle a, b \rangle) = \mathrm{F}^{\mathrm{T}}_{\psi(T \circ a)}(U, T, b) : D$$

Examples

- If $\psi, \psi' : \mathrm{OP}_D$, let $\psi +_{\mathrm{OP}} \psi'$ be the code for the ind.-rec. definitions with the constructors of ψ and ψ' coded into one constructor.
- Ordinary inductive definitions correspond to elements of $\mathrm{OP}_{\{*\}}$.
  - Then $\mathrm{T}_\gamma : \mathrm{U}_\gamma \to \{*\}$ is trivial.
- Code for $\mathbb{N}$ is

$$\begin{array}{l} \mathrm{init}(*) \\ +_{\mathrm{OP}} \mathrm{ind}(\{*\}, \lambda x.\mathrm{init}(*)) : \mathrm{OP}_{\{*\}} \end{array}$$

Examples

- Code for $A + B$ is

$$\begin{aligned}&\mathrm{nonind}(A, \lambda x.\mathrm{init}(*))\\ &+_{\mathrm{OP}}\mathrm{nonind}(B, \lambda x.\mathrm{init}(*)) : \mathrm{OP}_{\{*\}}\end{aligned}$$

- Code for $\mathrm{W}(A, B)$ is

$$\mathrm{nonind}(A, \lambda x.\mathrm{ind}(B(x), \lambda y.\mathrm{init}(*))) : \mathrm{OP}_{\{*\}}$$

- Code for a universe closed under $\mathbb{N}$, Σ is

$$\begin{aligned}&\mathrm{init}(\mathbb{N})\\ &+_{\mathrm{OP}}\mathrm{ind}(\{*\}, \lambda A.\mathrm{ind}(A(*), \lambda B.\mathrm{init}(\Sigma(A(*), B))))\\ &: \mathrm{OP}_{\mathrm{Set}}\end{aligned}$$

4. Results

- Generalisation to **indexed inductive-recursive definitions** has been developed.
  - Corresponds to the simultaneous ind.-rec. definitions of several sets $\mathrm{U}_\gamma(i) : \mathrm{Set} \quad (i : I)$, together with $\mathrm{T}_\gamma(i) : \mathrm{U}_\gamma(i) \to D[i]$.
- Special case: identity type.

Applications in Generic Programm.

- **Generic** (or better **generative**) programming is the definition of functions, which depend on the structure of types.
  - More than just **simple polymorphism**, in which one forms a type from another type without looking into it.
- Generic programming is used in $C++$ where one can define **typelists** and functions by induction over type lists.
- Similarly, in generic Haskell one defines functions by induction over the definition of data types.
- Goal is highly generic programs, automated software production.

OP_D and Generic Programming

- OP_D is a very general data type of types.
  Allows to define functions which take
  - an element of $\gamma : \mathrm{OP}_D$,
  - and an element of U_γ,
- and compute
  - a new element $\gamma' : \mathrm{OP}_D$
  - and a new element of $\mathrm{U}_{\gamma'}$.
- A very general form of **generic programming**.
- One example is the embedding of an inductive type into the same inductive type, but extended by one more constructor.
  - Not possible to treat this using ordinary polymorphism.

OP_D and Generic Programming

- Marcin Benke, Patrik Jansson and Peter Dybjer have used weak versions of OP_D in generic programming.
- One example is the type of **finitary inductive definitions** (inductive argument not indexed over sets).
- They were able to
  - define a generic decidable equality for such sets,
  - and show that it is an equivalence relation.

Related Structures

- In order to define models of type theory (or other theories) inside type theory, one often needs to define
  - a $\mathrm{U} : \mathrm{Set}$
  - together with sets $\mathrm{T} : \mathrm{U} \to \mathrm{Set}$

  simultaneously inductively.
- So $\mathrm{T}(x)$ is not fixed but defined inductively by referring to the inductive definition of U and other sets $\mathrm{T}(y)$.
- Therefore we cannot refer to $\mathrm{T}(x)$ negatively as in

$$\widehat{\Sigma} : (x : \mathrm{U}) \to (\mathrm{T}(x) \to \mathrm{U}) \to \mathrm{U}$$

Example

- For instance one defines simultaneously inductively

$$\begin{array}{lcl} \mathrm{Types} & : & \mathrm{Set} \\ \mathrm{Terms} & : & \mathrm{Types} \to \mathrm{Set} \end{array}$$

with constructors like

$$\begin{array}{ll} \mathrm{ap}: & (A, B : \mathrm{Types}) \\ & \to \mathrm{Terms}(A \mathbin{\widehat{\to}} B) \\ & \to \mathrm{Terms}(A) \\ & \to \mathrm{Terms}(B) \end{array}$$

(More precisely additional dependency on contexts needed).

Conclusion

- Introduction into dependent type theory (Martin-Löf Type Theory).
- Ind-rec. definitions as a generalisation of the underlying principles.
- Introduction of a type theory of ind.-rec. definitions.
- Contains a data type OP_D of codes for ind.-rec. definitions.
- Proof-theoretic strength known to be in $[|\mathrm{KPM}|, |\mathrm{KPM}^+|]$.
- Applications in generic programming.

Future Research

- Integration of Mahlo principle ("**Mahlo-inductive-rec. definitions**").
- Combination with coalgebras (couniverses).
- Integration of extended principles like the one just mentioned.
- More **examples** for usage of truly **inductive-recursive definitions** in programming.
  - Only known non-universe examples are:
    - Modelling of partial functions in type theory.
    - Normalisation proof of Martin-Löf type theory.
  - Expected that there are many more applications.
- More applications in generic/generative programming.