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

\[ T : U \rightarrow \text{Set} \]

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):

- String : Set,
- Integer : Set,
- Integer → Integer : Set,
- etc.
Polymorphic Types

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

E.g.

\[ \text{List} : \text{Set} \to \text{Set}, \]

\[ \text{List}(A) = \text{set of lists of elements of set } A. \]

Polymorphic types allow more generic programs.

One definition of a library for \text{List} rather than defining this library for each of

- \text{List}(\text{Integer}),
- \text{List}(\text{Char}),
- \text{List}(\text{String}),
- ...
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
    \[ \text{Tuple}(A, n) \]
  where
  \[ \text{Tuple} : \text{Set} \rightarrow \mathbb{N} \rightarrow \text{Set} \]
- Allows to define functions, the result type of which depends on the argument, e.g.
  \[ f : (n : \mathbb{N}) \rightarrow \text{Tuple}(\mathbb{N}, n) \]
Examples of Dependent Types

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

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

where

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

Matrix multiplication gets type

$$\text{matmult} : (n, m, k : \mathbb{N})$$

$$\to \text{Mat}(n, m) \to \text{Mat}(m, k) \to \text{Mat}(n, k)$$
The predicate

\[ \text{Sorted} : \text{List}(\mathbb{N}) \to \text{Set} \]

s.t. there exists \( p : \text{Sorted}(l) \) iff \( l \) is sorted is a dependent type.

The set \((l : \text{List}(\mathbb{N})) \times \text{Sorted}(l)\) is the set of sorted lists.

The set

\[(l : \text{List}(\mathbb{N})) \to (l' : \text{List}(\mathbb{N})) \times \text{Sorted}(l') \times (\text{EqElements}(l, l'))\]

is the set of sorting functions on \(\text{List}(\mathbb{N})\).
Logical Framework

- Basic logic framework has 2 main constructions:
  - The dependent function type \((x : A) \rightarrow B(x)\) for \(A : \text{Set}\), \(x : A \Rightarrow B(x) : \text{Set}\).
  - Elements are roughly speaking
    \[
    \{ f : A \rightarrow \bigcup_{x:A} B(x) \mid \forall x \in A. f(x) \in B(x) \}\n    \]
  - \(A \rightarrow B\) is the special case \((x : A) \rightarrow B\) where \(B\) does not depend on \(x\).
  - The dependent product \((x : A) \times B(x)\) for \(A : \text{Set}\), \(x : A \Rightarrow B : \text{Set}\).
  - Elements are roughly speaking
    \[
    \{ \langle a, b \rangle \mid a \in A, b \in B(a) \}\n    \]
Set vs. Type

We will use two type levels.

- **Set**, the type of sets = small types.
- **Type** the collection of big types.

\[
\text{Set} \subseteq \text{Type}, \text{Set} : \text{Type}.
\]

**Type** closed under (dependent) functions and products, but in the simplest version under nothing else.

So we have for instance, if \( A : \text{Set} \), then

\[
A \rightarrow \text{Set} : \text{Type}
\]

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:**
  
  \[ \text{Bool} : \text{Set} \]

- **Introduction rules:**
  
  \[ \text{tt} : \text{Bool} \quad \text{ff} : \text{Bool} \]

- **Elimination/equality rules:**
  
  If then else.
Visualisation of Bool

2 Constructors, both no arguments.
The Disjoint Union

**Formation rule:**

\[
\begin{array}{c}
A : \text{Set} \\
B : \text{Set}
\end{array}
\Rightarrow
\begin{array}{c}
A + B : \text{Set}
\end{array}
\]

**Introduction rules:**

\[\text{inl} : A \rightarrow (A + B)\]
\[\text{inr} : B \rightarrow (A + B)\]

(Additional premises of formation rule suppressed).

**Elimination/equality rule:**

\[
\text{case } x \text{ of } \\
\{ \text{inl}(a) \rightarrow \cdots \}
\]
\[
\{ \text{inr}(b) \rightarrow \cdots \}
\]
Visualisation of A+B

Both \texttt{inl} and \texttt{inr} have \textbf{one non-inductive argument}.
The $\Sigma$-Type

- **Formation rule:**

  \[
  \frac{A : \text{Set} \quad B : A \rightarrow \text{Set}}{\Sigma(A, B) : \text{Set}}
  \]

- **Introduction rule:**

  \[
  \frac{a : A \quad b : B(a)}{p(a, b) : \Sigma(A, B)}
  \]

- **Elimination/equality rule:**

  \[
  \text{case } x \text{ of } \{ \ p(a, b) \rightarrow \cdots \}
  \]
Visualisation of $\sum(A,B)$

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

- **Formation rule:**
  \[ \mathbb{N} : Set \]

- **Introduction rules:**
  \[
  \begin{align*}
  0 & : \mathbb{N} \\
  n : \mathbb{N} & \rightarrow S(n) : \mathbb{N}
  \end{align*}
  \]

- **Elimination/equality rule:**
  Induction/primitive recursion.
Visualisation of $\mathbb{N}$

- 0 has **no arguments**.
- S has one **inductive argument**.
Assume $A : \text{Set}$, $B : A \to \text{Set}$.

$W(A, B)$ is the type of well-founded recursive trees with branching degrees $(B(a))_{a : A}$.
The W-Type

- **Formation rule:**

\[
\begin{array}{c}
A : \text{Set} \quad B : A \to \text{Set} \\
\hline
W(A, B) : \text{Set}
\end{array}
\]

- **Introduction rule:**

\[
\begin{array}{c}
a : A \\
\hline
b : B(a) \to W(A, B) \\
\hline
\sup(a, b) : W(A, B)
\end{array}
\]

- **Elimination/equality rule:**

   Induction over trees.
Visualisation of $W(A,B)$

\[ W(A, B) \]

\[ b(x) \ (x : B(a)) \]

$\text{sup}$ has 2 arguments:

- First argument is \textbf{non-inductive}.
- Second argument is \textbf{inductive}, indexed over $B(a)$.

$B(a)$ \textbf{depends on the first argument} $a$. 

Anton Setzer: Inductive Recursive Definitions
Universes

A universe is a family of sets

Given by
- a set $U : \text{Set}$ of codes for sets,
- a decoding function $T : U \rightarrow \text{Set}$.
Universes

**Formation rules:**

\[
\frac{\text{U} : \text{Set}}{a : \text{U}} \quad \frac{\text{T}(a) : \text{Set}}{}
\]

**Introduction and Equality rules:**

\[
\frac{\text{:\(\hat{\text{N}}\) : U}}{\text{T}(\hat{\text{N}}) = \text{N}}
\]

\[
\frac{a : \text{U} \quad b : \text{T}(a) \rightarrow \text{U}}{\hat{\Sigma}(a, b) : \text{U}}
\]

\[
\text{T}(\hat{\Sigma}(a, b)) = \Sigma(\text{T}(a), \text{T} \circ b)
\]

Similarly for other type formers (except for U).

**Elimination/equality rules:** Induction over U.
Visualisation of U

\[ \Sigma(T(a), T \circ b) \]

\[ T(b(x)) \]

\[ \hat{N} \]

\[ \mathbb{N} \]

\[ U \]

\[ T(a) \]

\[ b(x) \ (x : T(a)) \]

\[ T(a) \]

\[ x : T(a) \]
Analysis

- Elements of \( U \) are defined \textit{inductively}, while defining \( T(a) \) for \( a : U \) \textit{recursively}.

- \( \hat{\Sigma} \) has two \textit{inductive arguments}
  - Second argument depends on \( T(a) \).
    - \( T(a) \) \textit{depends} on \( T \) applied to first argument \( a \).
  - \( T(\hat{\Sigma}(a, b)) \) \textit{is defined from}
    - \( T(a) \).
    - \( T(b(x)) (x : T(a)) \).

- Principles for defining a universe can be generalised to \textbf{higher type universes}, where \( T(a) \) can be an element of any type, e.g. \( \text{Set} \to \text{Set} \).
Advanced Example

Set of lists of natural numbers with distinct elements.

Inductive-recursive definition of

\[ \text{Freshlist} : \text{Set} \]

\[ _\#_ : \text{Freshlist} \rightarrow \mathbb{N} \rightarrow \text{Set} \]

Constructors:

\[
\begin{align*}
nil : & \text{Freshlist} \\
nil \# m & = \top \\
\text{cons} : & (n : \mathbb{N}, l : \text{Freshlist}, l \# n) \rightarrow \text{Freshlist} \\
\text{cons}(n, l, p) \# m & = (l \# m) \land (n \neq m)
\end{align*}
\]
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.
Several constructors can be \textit{encoded into one} constructor:

Assume constructors $C_i : (a : A_i) \rightarrow U \ (i = 1, \ldots, n)$.

Replace them by one constructor

$C : (i : \{1, \ldots, n\}, a : A_i) \rightarrow U$.

Only required: finite sets.
Will be part of the logical framework.
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.
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 $U : \text{Set}$, $T : U \rightarrow D$ for some type $D$.
- Let $D : \text{Type}$ be fixed.
  - In case of a standard universe
    \[ D = \text{Set} \]
  - In case of higher order universes
    \[ D = \text{Fam}(\text{Set}) \rightarrow \text{Fam}(\text{Set}) \]
    or higher types.
  - In case of inductive definitions ($T$ is trivial)
    \[ D = \{\ast\} \]
We introduce a type of codes for ind.-rec. definitions:

\[ \text{OP}_D : \text{Type} \]

If \( \gamma : \text{OP}_D \), we introduce \((U_\gamma, T_\gamma)\) ind.-rec.:

\[ U_\gamma : \text{Set} \]
\[ T_\gamma : U_\gamma \rightarrow D \]
Further, we define the **set of arguments** of the constructor $\text{intro}_\gamma$ of $U_\gamma$.

Argument set has to be defined, before $U_\gamma$, $T_\gamma$ has been introduced.

Will be defined for arbitrary $U : \text{Set}$, $T : U \to D$

\[
\gamma : \text{OP}_D
\]

\[
F^U_\gamma : (U : \text{Set}) \to (T : U \to D) \to \text{Set}
\]

**Introduction Rule for $U_\gamma$:**

\[
\text{intro}_\gamma : F^U_\gamma(U_\gamma, T_\gamma) \to U_\gamma
\]
Furthermore, we have to define the result of $T_\gamma$ applied to $\text{intro}_\gamma(a)$.

Again, we have to define it before the definition of $U_\gamma$, $T_\gamma$ is finished.

So we define

$$F^T_\gamma : (U : \text{Set}) \rightarrow (T : U \rightarrow D) \rightarrow F^U_\gamma(U, T) \rightarrow D$$

Equality Rule for $T_\gamma$:

$$T_\gamma(\text{intro}_\gamma(a)) = F^T_\gamma(U_\gamma, T_\gamma, a)$$
\[ F_\gamma \text{ as a Functor} \]

We have

\[ F_U^\gamma : (U : \text{Set}) \to (T : U \to D) \to \text{Set} \]
\[ F_T^\gamma : (U : \text{Set}) \to (T : U \to D) \to F_U^\gamma(U, T) \to D \]

\[ F_U^\gamma, F_T^\gamma \text{ will form the object part of a functor} \]

\[ F_\gamma : \text{Fam}(D) \to \text{Fam}(D) \]

where

\[ \text{Fam}(D) := (U : \text{Set}) \times (U \to D) \]

and \( \langle U_\gamma, T_\gamma \rangle \) is the initial algebra of \( F_\gamma \).

(Slight modification of the proof in the paper is needed.)
Elimin./Equal. Rules for $U_{\gamma}$, $T_{\gamma}$

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

- Initial case for $\text{OP}_D$: No arguments.
- We need only to define the result of $T_\gamma$ applied to the constructor, i.e. require one element $\psi : D$.

\[
\begin{align*}
\psi : D \\
\text{init}(\psi) : \text{OP}_D \\
F^U_{\text{init}(\psi)}(U, T) &= \{\ast\} : \text{Set} \\
F^T_{\text{init}(\psi)}(U, T, \ast) &= \psi : D
\end{align*}
\]
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 $\psi : A \to \text{OP}_D$.

\[
\begin{align*}
A : \text{Set} & \quad \psi : A \to \text{OP}_D \\
\text{nonind}(A, \psi) : \text{OP}_D
\end{align*}
\]

\[
\begin{align*}
F^U_{\text{nonind}(A, \psi)}(U, T) &= (a : A) \times F^U_{\psi(a)}(U, T) : \text{Set} \\
F^T_{\text{nonind}(A, \psi)}(U, T, \langle a, b \rangle) &= F^T_{\psi(a)}(U, T, b) : D
\end{align*}
\]
For an inductive argument we need to know

- The set $A$, over which the argument is indexed over.
  - $A = \{ \ast \}$ give the special case of a single argument.

Depending on the result of $T$ applied to the arguments of $A$, i.e. depending on $A \rightarrow D$, the later arguments of the constructor:

We need a function $\psi : (A \rightarrow D) \rightarrow \text{OP}_D$.

\[
\begin{align*}
A : \text{Set} & \quad \psi : (A \rightarrow D) \rightarrow \text{OP}_D \\
\text{ind}(A, \psi) : \text{OP}_D \\
F^U_{\text{ind}(A, \psi)}(U, T) &= (a : A \rightarrow U) \times F^U_{\psi(T \circ a)}(U, T) : \text{Set} \\
F^T_{\text{ind}(A, \psi)}(U, T, \langle a, b \rangle) &= F^T_{\psi(T \circ a)}(U, T, b) : D
\end{align*}
\]
Examples

- If $\psi, \psi' : \text{OP}_D$, let $\psi +_{\text{OP}} \psi'$ be the code for the ind.-rec. definitions with the constructors of $\psi$ and $\psi'$ coded into one constructor.

- Ordinary inductive definitions correspond to elements of $\text{OP}\{\ast\}$.
  - Then $T_\gamma : U_\gamma \rightarrow \{\ast\}$ is trivial.

- Code for $\mathbb{N}$ is

  \[
  \begin{align*}
  \text{init}(\ast) \\
  +_{\text{OP}} \text{ind}(\{\ast\}, \lambda x. \text{init}(\ast)) : \text{OP}\{\ast\}
  \end{align*}
  \]
Examples

- Code for $A + B$ is

  \[
  \text{nonind}(A, \lambda x. \text{init}(\ast)) \\
  +_{\text{OP}} \text{nonind}(B, \lambda x. \text{init}(\ast)) : \text{OP}\{\ast}\}
  \]

- Code for $W(A, B)$ is

  \[
  \text{nonind}(A, \lambda x. \text{ind}(B(x), \lambda y. \text{init}(\ast))) : \text{OP}\{\ast}\}
  \]

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

  \[
  \text{init}(\mathbb{N}) \\
  +_{\text{OP}} \text{ind}(\{\ast\}, \lambda A. \text{ind}(A(\ast), \lambda B. \text{init}(\Sigma(A(\ast), B)))) \\
  : \text{OP}_{\text{Set}}
  \]
4. Results

- Generalisation to **indexed inductive-recursive definitions** has been developed.
  - Corresponds to the simultaneous ind.-rec. definitions of several sets $U_\gamma(i) : \text{Set } (i : I)$, together with $T_\gamma(i) : U_\gamma(i) \to D[i]$.

- Special case: identity type.
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.
\( \text{OP}_D \) and Generic Programming

\( \text{OP}_D \) is a very general data type of types. Allows to define functions which take

- an element of \( \gamma : \text{OP}_D \),
- and an element of \( U_\gamma \),

and compute

- a new element \( \gamma' : \text{OP}_D \)
- and a new element of \( U_{\gamma'} \).

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.

Anton Setzer: Inductive Recursive Definitions
OP\textsubscript{D} and Generic Programming

Marcin Benke, Patrik Jansson and Peter Dybjer have used weak versions of OP\textsubscript{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.
In order to define models of type theory (or other theories) inside type theory, one often needs to define

- a $U : \text{Set}$
- together with sets $T : U \rightarrow \text{Set}$ simultaneously inductively.

So $T(x)$ is not fixed but defined inductively by referring to the inductive definition of $U$ and other sets $T(y)$.

Therefore we cannot refer to $T(x)$ negatively as in

$$\hat{\Sigma} : (x : U) \rightarrow (T(x) \rightarrow U) \rightarrow U$$
Example

For instance one defines simultaneously inductively

\[
\begin{align*}
\text{Types} & : \text{Set} \\
\text{Terms} & : \text{Types} \to \text{Set}
\end{align*}
\]

with constructors like

\[
\begin{align*}
ap & : (A, B : \text{Types}) \\
& \to \text{Terms}(A \Rightarrow B) \\
& \to \text{Terms}(A) \\
& \to \text{Terms}(B)
\end{align*}
\]

(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 $[|KPM|, |KPM^+|]$.
- 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.