A Mini Course on Martin-Löf Type Theory
Algebras, Coalgebras, and Interactive Theorem Proving

Anton Setzer
Swansea University, Swansea UK

Lisbon, 9 September 2015

Type Theory and Interactive Theorem Proving

Key Philosophical Principles of Martin-Löf Type Theory

Setup of Martin-Löf Type Theory

Basic Types in Martin-Löf Type Theory

The Logical Framework

Inductive Data Types (Algebras) in Type Theory

Coinductive Data Types (Coalgebras) in Type Theory

Computer-Assisted Theorem Proving

- A lot of research has been invested in Computer-assisted Theorem Proving.
- Motivation
  - Guarantee that proofs are correct.
  - Especially a problem in software verification (lots of boring cases).
  - Can be essential in critical software.
  - Help of machine in constructing proofs (proof search).
  - Ideally the mathematician can concentrate on the key ideas and the machine deals with the details.
  - Ideally one could have a machine assisted proof in demonstrating that the proof is correct and then concentrate in the presentation on the key ideas.
  - Desire to have systems as powerful as computer algebra systems such as Maple and MATLAB.
Interactive vs Automated Theorem Proving

- **Automated Theorem Proving:** User provides the problem, machine finds the proof.
  - Works only for **restricted theories**, which often need to be **finitizable**.
- **Interactive Theorem Proving:** Proof is carried out by the **user**.
- In reality **hybrid approaches**:
  - In Automated Theorem Proving **hints** in the form of **intermediate lemmata** are given by the user before starting the automated proof search.
  - In Interactive Theorem Proving **proof tactics** and **automated theorem proving tools** are used to prove the elementary steps.
- **Warning:** Theorem Proving still **hard work**.
  - It’s like relationship between the **idea of a program** and **writing the program**.
  - The machine **doesn’t allow any gaps**.

Types in Programming

- Simple Types are used in programming to
  - help obtaining correct programs,
  - help writing programs.
- For instance assume you have given \(a, f\) and **want to construct a solution for \(p\)** below. We **solve the goal** using \(f\) (functional application written \(f \, x\)). We have **a new goal of type** \(\text{Int}\) We **solve the goal** using \(a\)
  \[
  \begin{align*}
  a & : \text{Int} \\
  a & = \cdots \\
  f & : \text{Int} \rightarrow \text{String} \\
  f & = \cdots \\
  p & : \text{String} \\
  p & = \{! \, !\} \, f \, \{! \, !\} \, f \, \{! \, !\} \, f \, a
  \end{align*}
  \]

Dependent Types

- Formulas are considered as types, and elements of those proofs are proofs of that formula.
- Formulas with free variables are **dependent types**:
- The formula \(x \equiv 0\) depends on \(x : \mathbb{N}\).

Formulas give rise to new Type Constructs

- A proof of
  \[
  \forall x : A. B(x)
  \]
  is a function which computes from
  \[
  a : A
  \]
  a proof of
  \[
  B(a)
  \]
- So a proof is an element of the **dependent function type**
  \[
  (x : A) \rightarrow B(x)
  \]
  the set of functions mapping \(a : A\) to an element of \(B(a)\).
Dependent Types in Other Settings

- Dependent types occur as well naturally in mathematics:
- The type of $\text{Mat}(n, m)$ of $n \times m$ matrices depends on $n, m$.
- Matrix multiplication has type
  
  $$\text{matmult} : (n, m, k : \mathbb{N}) \rightarrow \text{Mat}(n, m) \rightarrow \text{Mat}(m, k) \rightarrow \text{Mat}(n, k)$$

- In simply typed languages we can only have
  
  $$\text{matmult} : \text{Mat} \rightarrow \text{Mat} \rightarrow \text{Mat}$$

Dependent Types in Generic Programming

- In general dependent types allow to define more **generic** or **generative programs**.
- Example: **Marks of a lecture course**:
  A lecture course may have different components (exams, coursework).
- On next slide **Set** is the collection of small types (notation for historic reasons used).

Dependent Types in Generative Programming

```plaintext
numberOfComponents : Lecture \rightarrow \mathbb{N}
numberOfComponents \ell = \cdots

\text{Marks} : (\ell : \text{Lecture}) \rightarrow \text{Set}
\text{Marks} \ell = \text{Mark}^{\text{numberOfComponents}} \ell

\text{Weighting} : (\ell : \text{Lecture}) \rightarrow \text{Set}
\text{Weighting} \ell = \text{Percentage}^{\text{numberOfComponents}} \ell

\text{finalMark} : (\ell : \text{Lecture}) \rightarrow \text{Marks} \ell \rightarrow \text{Weighting} \ell \rightarrow \text{Mark}
\text{finalMark} \ell m w = \cdots
```

Generative Programming

- You can add that the weightings add up to 100%.
- In general you can describe complex data structures using dependent types.
Interactive Theorem Provers based on Dependent Types

- Agda (based on Martin-Löf Type theory).
- Coq (based on calculus of constructions, impredicative).
  - Formalisation of four colour problem.
  - Microsoft has invested in it (but development happening at INRIA, France).
  - Project of proving Kepler conjecture in it.
  - Inspired Voevodsky to develop Homotopy Type Theory.
- Epigram (based on Martin-Löf Type theory, intended as a programming language).
- Idris (relatively new language).
- Cayenne (programming language, no longer supported).
- LEGO (theorem prover from Edinburgh, no longer supported).
- Many more.

Key Philosophical Principles of Martin-Löf Type Theory

- Martin-Löf Type Theory developed to provide a new foundation of mathematics.
- Idea to develop a theory where we have direct insight into its consistency.
- By Gödel’s 2nd Incompleteness theorem we know we cannot prove the consistency of any reasonable mathematical theory.
- However, we want mathematics to be meaningful.
  - We don’t want to have a proof of Fermat’s last theorem and a counter example.
- Mathematics is meaningful, because we have an intuition about why it is correct.
Example: Induction

- For instance that if we have proofs of
  \[ A(0) \]
  \[ \forall n : \mathbb{N}. A(n) \to A(n + 1) \]
  we can convince ourselves that \( \forall n : \mathbb{N}. A(n) \) holds.
  - Because for every \( n : \mathbb{N} \) we can construct a proof of \( A(n) \) by using
    the base case and \( n \) times the induction step.
- Martin-Löf Type Theory is an attempt to formalise the reasons
  why we believe in the consistency of mathematical constructions.

Objects of Type Theory

- We have a direct good understanding of finite objects.
- Finite objects can always be encoded into natural numbers, and
  individual natural numbers are easy to understand.
- In general finite objects can be represented as terms.
  Examples of terms:
  
  \[
  \begin{align*}
  \text{zero} \\
  \text{suc zero} \\
  \text{suc zero} + \text{suc zero} \\
  [] & \quad \text{(empty list)} \\
  \text{cons zero} [] & \quad \text{(result of adding in front of the empty list zero)}
  \end{align*}
  \]

Beyond Finitism

- Some terms are in normal form, e.g. \( \text{suc (suc (suc zero))} \)
- Other terms have reductions, e.g.
  \[
  \text{zero + suc zero} \to \text{suc (zero + zero)} \to \text{suc zero}.
  \]
- Martin-Löf uses program for terms as above, which evaluate
  according to the reduction rules.

- We can form a mathematical theory where we have finitely many finite objects, and convince ourselves of its consistency.
- The resulting theory is not very expressive however.
- In order to talk about something which of infinite nature, we
  introduce the concept of a type.
A **type** \( A \) is given by a collection of rules which allow us to conclude that certain objects are elements of that type.

\[ a : A \]

and how to form from an element \( a : A \) an element of another type \( B \).

We don’t consider a type as a set of elements (although when working with one often thinks like that). That would mean that we have an infinite object per se.

**Example: Natural Numbers**

For instance we have

\[
\begin{align*}
\text{zero} : \mathbb{N} \\
\text{if } n : \mathbb{N} \text{ then } \text{suc } n : \mathbb{N}
\end{align*}
\]

This is written as rules

\[
\begin{array}{c}
\text{zero} : \mathbb{N} \\
\text{suc } n : \mathbb{N}
\end{array}
\]

We can conclude for instance

\[ \text{suc } (\text{suc } \text{zero}) : \mathbb{N} \]

Furthermore if we have another type \( B \), i.e.

\[ B : \text{Set} \]

and if we have

\[
\begin{align*}
\text{b} : B \\
\text{g} : B \rightarrow B
\end{align*}
\]

we can form

\[
\begin{align*}
\text{h} : \mathbb{N} \rightarrow B \\
\text{h zero} &= \text{b} \\
\text{h } (\text{suc } n) &= \text{g } (\text{h } n)
\end{align*}
\]

These rules express what we informally describe as **iteration**

\[ \text{h } n = \text{g}^n \text{ b} \]

We will later introduce stronger elimination rules for natural numbers (dependent higher type primitive recursion).

**Representation of Infinite Objects by Finite Objects**

This doesn’t mean that we can’t speak of infinite objects. We can have for instance a collection of sets (or universe)

\[ U : \text{Set} \]

which contains a code for the set of natural numbers

\[ \widehat{\mathbb{N}} : U \]

We can consider an operation \( T \), which decodes codes in \( U \) into sets, i.e. we have the rule

\[
\begin{array}{c}
\text{u} : U \\
T \text{u} : \text{Set}
\end{array}
\]

Then we can add a rule

\[ T \widehat{\mathbb{N}} = \mathbb{N} : \text{Set} \]

\( \widehat{\mathbb{N}} \) is still a finite object, but it represents (via \( T \)) a type which has infinitely many elements.
Constructive Mathematics

- Before we already said that propositions should be considered as types.
- Elements of such types should be proofs.
- These proofs will give constructive content of proofs.
- A proof
  \[ p : (\exists x : A.B(x)) \]
  should allow us to compute an
  \[ a : A \text{ s.t. } B(a) \text{ is true} \]

Similarly from a proof \( p : A \lor B \)
we should able to compute a Boolean value, such that if it is true, \( A \) holds, and if it false \( B \) holds.

- Problem: We can’t get in general a proof of \( A \lor \neg A \)
  unless we can decide whether \( A \) or \( \neg A \) holds

Link between Logic and Computer Programming

- Constructive Mathematics provides a direct link between proofs/logic and programs/data.
- In type theory there is no distinction between a data type and a logical formula (radical propositions as types).
- Allows to write programs in which data and logical formulas are mixed.

BHK-Interpretation of Logical Connectives

The Brouwer-Heyting-Kolmogorov (BHK) Interpretation of the logical connectives is the constructive interpretation of the logical operators.

- A proof of \( A \land B \)
  is given by a proof of \( A \) and a proof of \( B \)
- A proof of \( A \lor B \)
  is given by a proof of \( A \) or a proof of \( B \)
  plus the information which of the two holds.
BHK-Interpretation of Logical Connectives

- A proof of \( A \rightarrow B \)
  is a function (program) which
  computes from a proof of \( A \) a proof of \( B \)
- A proof of \( \forall x : A. B(x) \)
  is a function (program) which
  for every \( a : A \) computes a proof of \( B(a) \)
- A proof of \( \exists x : A. B(x) \)
  consists of
  an \( a : A \) plus a proof of \( B(a) \)

Intuitionistic Logic

- We don’t obtain stability
  \( \neg\neg A \rightarrow A \)
- So we cannot carry out indirect proofs:
  - An indirect proof is as follows: itmm In order to proof \( A \) assume \( \neg A \)
  - Then derive a contradiction
  - So \( \neg A \) is false (i.e. we have \( \neg\neg A \).
  - By stability we get \( A \).
- Stability is not provable in general constructively:
  - If we have \( \neg\neg A \) we have a method which from a proof of \( \neg A \) computes
    a proof of \( \bot \).
  - This does not give as a method to compute a proof of \( A \).

Double Negation Interpretation

- However one can interpret formulas from classical logic into
  intuitionistic logic so that a formula is classically provable iff its
  translation is intuitionistically provable.
- Double negation interpretation (not part of this course).
Double Negation Interpretation

▶ Easy to see with ∨:
Intuitionistically we have
\[\neg(\neg(\neg A \lor B)) \leftrightarrow \neg(\neg A \land \neg B)\]
If we replace
\[\neg A \lor B\]
by
\[\neg A \land \neg B\]
then
\[\neg A \land \neg B\]
behaves intuitionistically (with double negated formulas) like classical ∨.
▶ Especially tertium non datur is provable
\[\neg A = \neg(\neg A \land \neg A)\]

Judgements of Type Theory

▶ The statements of type theory are called "judgements".
▶ There are four judgements of type theory:
  ▶ A is a type written as
    \[A : \text{Set}\]
  ▶ A and B are equal types written as
    \[A = B : \text{Set}\]
  ▶ a is an element of type A written as
    \[a : A\]
  ▶ a, b are equal elements of type A written as
    \[a = b : A\]
Setup of Martin-Löf Type Theory

- The notion of reduction
  \[ s \rightarrow t \]
  corresponds to computation rules where term \( s \) evaluates to \( t \).

- In type theory one uses instead
  \[ s = t \]
  which is the reflexive/symmetric/transitive closure of \( \rightarrow \) or equivalence relation containing \( \rightarrow \).

- In most rules when concluding
  \[ s = t : A \]
  it is actually the case that we have a reduction
  \[ s \rightarrow t \]

Examples of Dependent Judgements

- We have as well dependent judgements, for instance for expressing
  \[
  \text{if } x : N \text{ then suc } x : N
  \]
  which we write
  \[ x : N \Rightarrow \text{suc } x : N \]

- Examples:
  \[
  \begin{align*}
  x : N, y : N & \Rightarrow x + y : N \\
  x : N & \Rightarrow x + \text{zero} = x : N \\
  x : \text{List} & \Rightarrow \text{Sorted } x : \text{Set} \\
  & \Rightarrow \text{Sorted } [] = \text{True} : \text{Set}
  \end{align*}
  \]

- In general a dependent judgement has the form
  \[
  x_1 : A_1, x_2 : A_2(x_1), \ldots, x_n : A_n(x_1, \ldots, x_{n-1}) \Rightarrow \theta(x_1, \ldots, x_n)
  \]
  where, if write \( \vec{x} \) for \( x_1, \ldots, x_n \)
  \[ \theta(\vec{x}) \]
  is one of the four judgements before
  \[
  B(\vec{x}) : \text{Set} \quad \text{or} \quad B(\vec{x}) = B'(\vec{x}) : \text{Set} \quad \text{or} \quad b(\vec{x}) : B(\vec{x}) \quad \text{or} \quad b(\vec{x}) = b'(\vec{x}) : B(\vec{x})
  \]
Judgements in Agda

- In the theorem prover Agda we can define functions and objects by writing

  \[ n : \mathbb{N} \]
  \[ n = \text{zero} \]
  \[ f : \mathbb{N} \to \mathbb{N} \]
  \[ f \text{ zero} = \text{suc zero} \]
  \[ f \text{ (suc m)} = \text{suc (suc(f m))} \]

- = above is a reduction rule.

- We can type in a term e.g. \[ f n \]

and compute its normal form which is in this case

\[ \text{suc zero} \]

We can check whether \( t : A \) by type checking

\[ a : A \]
\[ a = t \]

However we can check \( t = s : A \) only indirectly via its consequences.

The judgement \( s = t : A \) is built-in as part of the machinery of Agda.

Four Kinds of Rules for each Type

For each type \( A \) there are 4 kinds of rules:

- **Formation rules:**
  They form a new type e.g.
  \[ N : \text{Set} \]

- **Introduction Rules:**
  They introduce elements of a type, e.g.
  \[
  \begin{align*}
  \text{zero} : \mathbb{N} \\
  n : \mathbb{N} \\
  \text{suc} n : \mathbb{N}
  \end{align*}
  \]

- **Elimination Rules:**
  They allow to construct from an element of one type elements of another type.
  For instance iteration for \( \mathbb{N} \) would correspond to the rule
  \[
  B : \text{Set} \quad b : B \quad g : B \to B \quad n : \mathbb{N} \\
  \h n : B
  \]

  where
  \[ h := \text{iter} B b g \]
Equality Rules:
They show how if we introduce an element of that type and then eliminate it how it is computed (we use \( h \) as before)

\[
\begin{align*}
\frac{B : \text{Set} \quad b : B \quad g : B \rightarrow B}{h \ Zero = b : B} \\
\frac{B : \text{Set} \quad b : B \quad g : B \rightarrow B \quad n : \mathbb{N}}{h (\text{suc} n) = g (h n) : B}
\end{align*}
\]

Equality Versions of the Rules

- There are as well equality versions of the above rules.
- They express that if the premises of a rule are equal the conclusions are equal as well.
- For instance the equality version of the rule

\[
\frac{n : \mathbb{N}}{\text{suc} n : \mathbb{N}}
\]

is

\[
\frac{n = m : \mathbb{N}}{\text{suc} n = \text{suc} m : \mathbb{N}}
\]

Canonical vs Non-Canonical Elements

- The elements introduced by an introduction rule start with a constructor.
- For instance the constructors of \( \mathbb{N} \) are

\[
\begin{align*}
\text{zero} \quad \text{and} \quad \text{suc}
\end{align*}
\]

- Elements introduced by an introduction rule are called canonical elements.
  - Canonical elements of \( \mathbb{N} \) are for instance

\[
\begin{align*}
\text{zero} \quad \text{suc (zero + zero)}
\end{align*}
\]
  
  where \(+\) is defined using elimination rules.
- Elements introduced by an elimination rule are non-canonical elements. For instance

\[
\text{zero + zero}
\]
- Using the equality rules, every non canonical element of a type is supposed to evaluate to a canonical element of that type.

Canonical elements of \( \mathbb{N} \)

- A canonical element of \( \mathbb{N} \) can be evaluated further.
- E.g. we have

\[
\text{suc (zero + zero)} \rightarrow \text{suc zero}
\]
- In case of a function type \( \lambda x.t \) is considered to be canonical.
- Note that in

\[
\lambda x.x : \mathbb{N} \rightarrow \mathbb{N}
\]

\( x \) doesn’t start with a constructor (doesn’t even make sense to ask for it, because it is an open term).
So here it is crucial that it is only required that a canonical element starts with a constructor.
Canonical elements of \( \mathbb{N} \)

- The type checking of equality is based on this notation of canonical element or head normal form.
  - In order to check \( s = t : \mathbb{N} \)
    we first reduce \( s \) and \( t \) to canonical form.
  - If they start with different constructors, \( s \) and \( t \) are different. E.g. if \( s \rightarrow \text{zero} \), \( t \rightarrow \text{suc} \ t' \) there is no need to evaluate \( t' \).
  - If they have the same constructor, e.g. \( s \rightarrow \text{suc} \ s' \ t \rightarrow \text{suc} \ t' \) then we compare \( s' \) and \( t' \).

The Type of Booleans

- One of the Simplest types is the type of Booleans.
- **Formation rule:**
  \[ \mathbb{B} : \text{Set} \]
- **Introduction rules:**
  \[ \text{tt} : \mathbb{B} \quad \text{ff} : \mathbb{B} \]
- **Elimination rule:**
  \[ x : \mathbb{B} \Rightarrow C(x) : \text{Set} \quad \text{step}_{\text{tt}} : C(\text{tt}) \quad \text{step}_{\text{ff}} : C(\text{ff}) \quad b : \mathbb{B} \]
  \[ \text{elim}_{\mathbb{B}}(\text{step}_{\text{tt}}, \text{step}_{\text{ff}}, b) : C(b) \]
Visualisation (Booleans)

2 Constructors, both no arguments.

Booleans in Agda

data B : Set where
  tt : B
  ff : B

¬ : B → B
¬ tt = ff
¬ ff = tt

Finite Types

Similar versions for types with 0, 1, 3, 4, ... elements.

Special case ∅.

Empty Type

Formation rule:
∅ : Set

Introduction rules:
There is no introduction rule.

Elimination rule:

Equality rules:
There is no equality rule.
Basic Types in Martin-Löf Type Theory

The Logical Framework (LF)

- When writing elimination rules we need to deal with notions such as
  - $C(x)$ is a set depending on $x : B$.
  - instantiate $x = \text{tt}$ and get $C(\text{tt})$.
- Idea of the logical framework (LF) is
  - Instead of saying $x : B \Rightarrow C(x) : \text{Set}$
    
    we write
    
    $C : B \rightarrow \text{Set}$
  - Then we can apply $C$ to $\text{tt}$ and obtain
    
    $C \text{ tt} : \text{Set}$
- We will introduce the LF more formally later.

LF and Foundations

- From a foundational point of view the LF is difficult.
  - It treats the collection of sets as an entity, at least as if one considers it naively.
  - The foundations of Martin-Löf Type Theory work best without the LF.
- When using it in the basic type theory below it could be avoided.
- We will use it just as a convenient way of writing the rules nicely.

Rules for Booleans Using the LF

- **Formation rule:**
  
  $B : \text{Set}$

- **Introduction rules:**
  
  $\text{tt} : B \quad \text{ff} : B$

- **Elimination rule:**
  
  $C : B \rightarrow \text{Set} \quad \text{step}_t : C \text{ tt} \quad \text{step}_f : C \text{ ff} \quad b : B$
  
  $\text{elim}_B C \text{ step}_t \text{ step}_f b : C b$

- **Equality rules:**
  
  $\text{elim}_B C \text{ step}_t \text{ step}_f \text{ tt} = \text{step}_t : C \text{ tt}$
  
  $\text{elim}_B C \text{ step}_t \text{ step}_f \text{ ff} = \text{step}_f : C \text{ ff}$
- We can even write
  \[
  \text{elim}_B : (C : \mathbb{B} \to \text{Set}) \\
  \to C \text{ tt} \\
  \to C \text{ ff} \\
  \to \mathbb{B} \\
  \to \text{Set}
  \]

- **Formation rule:**
  \[
  A : \text{Set} \\
  B : \text{Set} \\
  \frac{}{A + B : \text{Set}}
  \]

- **Introduction rules:**
  \[
  \frac{a : A}{\text{inl} \ a : A + B} \\
  \frac{b : B}{\text{inr} \ b : A + B}
  \]

- **Elimination rule:**
  \[
  C : A + B \to \text{Set} \\
  \text{step}_{\text{inl}} : (x : A) \to C (\text{inl} \ x) \\
  \text{step}_{\text{inr}} : (x : B) \to C (\text{inr} \ x) \\
  \frac{c : A + B}{\text{elim}_+ \ C \ \text{step}_{\text{inl}} \ \text{step}_{\text{inr}} \ c : C \ c}
  \]

- **Equality rules:**
  \[
  \text{elim}_+ \ C \ \text{step}_{\text{inl}} \ \text{step}_{\text{inr}} (\text{inl} \ a) = \text{step}_{\text{inl}} \ a : C (\text{inl} \ a) \\
  \text{elim}_+ \ C \ \text{step}_{\text{inl}} \ \text{step}_{\text{inr}} (\text{inr} \ b) = \text{step}_{\text{inr}} \ b : C (\text{inr} \ b)
  \]

- Both \text{inl} and \text{inr} have one non-inductive argument.
A proof of $A \lor B$ is a proof of $A$ or a proof of $B$.

So $A \lor B$ is just $A + B$.

- $\lor$ denotes the logical disjunction.

The $\Sigma$-Type

- Formation rule:
  
  \[
  \begin{array}{c}
  A : \text{Set} \\
  B : A \rightarrow \text{Set} \\
  \hline
  \Sigma A B : \text{Set}
  \end{array}
  \]

- Introduction rule:
  
  \[
  \begin{array}{c}
  a : A \\
  b : B a \\
  \hline
  p a b : \Sigma A B
  \end{array}
  \]

The $\Sigma$-Type

- Elimination rule:
  
  \[
  \begin{array}{c}
  C : \Sigma A B \rightarrow \text{Set} \\
  \hline
  \text{step} : (a : A, b : B a) \rightarrow C (p a b) \\
  \hline
  c : \Sigma A B \\
  \hline
  \text{elim}_\Sigma C \text{ step } c : C c
  \end{array}
  \]

- Equality rule:
  
  \[
  \text{elim}_\Sigma C \text{ step } (p a b) = \text{step } a b : C (p a b)
  \]
Visualisation \((\Sigma (A, B))\)

\[
\Sigma A B
\]

- \(p\) has two non-inductive arguments.
- The type of the 2nd argument depends on the 1st argument.

\[a \vdash b \quad B a\]

\[
\text{Σ A B in Agda}
\]

\[
\text{data } \Sigma (A : \text{Set}) (B : A \rightarrow \text{Set}) : \text{Set where}
\]
\[
p : (a : A) \rightarrow B a \rightarrow \Sigma A B
\]

postulate \(A : \text{Set}\)
postulate \(B : A \rightarrow \text{Set}\)

\[
\pi_0 : \Sigma A B \rightarrow A
\]
\[
\pi_0 (p \ a \ b) = a
\]

\[
\pi_1 : (x : \Sigma A B) \rightarrow B (\pi_0 x)
\]
\[
\pi_1 (p \ a \ b) = b
\]

\[
\exists \text{ as } \Sigma
\]

- With the LF, a formula depending on \(x : A\) is a
  \[
  B : A \rightarrow \text{Set}
  \]
- A proof of \(\exists x : A, B x\) is
  - an \(a : A\)
  - together with a \(b : B a\)
- That’s just an element of
  \[
  \Sigma A B
  \]

\[
\text{Natural numbers}
\]

- **Formation rule:**
  \[
  \mathbb{N} : \text{Set}
  \]
- **Introduction rules:**
  \[
  \begin{align*}
  \text{zero} : \mathbb{N} \\
  S n : \mathbb{N}
  \end{align*}
  \]
- **Elimination rule:**
  \[
  \begin{align*}
  \text{step}_{\text{zero}} : C \text{ zero} \\
  \text{step}_{S} : (n : \mathbb{N}, C n) \rightarrow C (S n)
  \end{align*}
  \]
  \[
  \begin{align*}
  n : \mathbb{N} \\
  \text{elim}_{\mathbb{N}} C \text{ step}_{\text{zero}} \text{ step}_{S} n : C n
  \end{align*}
  \]
Natural numbers

- **Equality rules:**

  \[
  \text{elim}_\mathbb{N} C \, \text{step}_\text{zero} \, \text{step}_\mathbb{N} \, \text{zero} = \text{step}_\text{zero} : C \, \text{zero}
  \]

  \[
  \text{elim}_\mathbb{N} C \, \text{step}_\text{zero} \, \text{step}_\mathbb{N} (S \, n) \\
  = \text{step}_\mathbb{N} \, n \, (\text{elim}_\mathbb{N} C \, \text{step}_\text{zero} \, \text{step}_\mathbb{N} \, n) : C \, (S \, n)
  \]

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

W-Type

**Formation rule:**

\[
\frac{A : \text{Set} \quad B : A \rightarrow \text{Set}}{W \, A \, B : \text{Set}}
\]

**Introduction rule:**

\[
\frac{a : A \quad b : B \, a \rightarrow W \, A \, B}{\text{sup} \, a \, b \, : W \, A \, B}
\]

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

- **Elimination rule:**
  \[
  C : W A B \rightarrow \text{Set} \\
  \text{step} : (a : A) \rightarrow (b : B a \rightarrow W A B) \\
  \rightarrow (ih : (x : B a) \rightarrow C (b x)) \\
  \rightarrow C (\text{sup} a b) \\
  c : W A B \\
  \text{elim}_W C \text{ step} c : C c
  \]

- **Equality rule:**
  \[
  \text{elim}_W C \text{ step} (\text{sup} a b) \\
  = \text{step} a b (\lambda x. \text{elim}_W C \text{ step} (b x)) : C (\text{sup} a b)
  \]

  Here \( \lambda x.t \) is the function mapping \( x \) to \( t \).
  (More details follow below when dealing with the function set).

---

Universes

- A universe is a family of sets
- Given by
  - an set \( U : \text{Set} \) of codes for sets,
  - a decoding function \( T : U \rightarrow \text{Set} \).

---

Visualisation (\( W A B \))

- sup has two arguments
  - First argument is non-inductive.
  - Second argument is inductive, indexed over \( B a \).
  - \( (B a) \) depends on the first argument \( a \).

---

Universes

- **Formation rules:**
  \[
  U : \text{Set} \\
  T : U \rightarrow \text{Set}
  \]

- **Introduction and Equality rules:**
  \[
  \hat{N} : U \\
  T \hat{N} = N \\
  a : U \\
  b : T a \rightarrow U \\
  \hat{\Sigma} a b : U \\
  T(\hat{\Sigma} a b) = \Sigma (T a) (T \circ b)
  \]
  Similarly for other type formers (except for \( U \)).
**Elimination Rules for \( U \)**

- Elimination rule for \( U \) can be defined.
- Not very useful (e.g. one cannot define an embedding of \( U \) into itself using elimination rules).

**Analysis**

- Elements of \( U \) are defined **inductively**, while defining \((T \ a)\) for \( a : U \) **recursively**.
- \( \Sigma \) has two inductive arguments
  - Second argument is indexed over \((T \ a)\).
    - Index set \((T \ a)\) for second argument depends on the \( T \) applied to first argument \( a \).
    - \( T(\Sigma \ a \ b) \) is defined from
      - \((T \ a)\),
      - \((T \ (b \ x))(x : T \ a)\).
- Principles for defining a universe can be generalised to **higher type universes**, where \((T \ a)\) can be an element of any type, e.g. \( \text{Set} \to \text{Set} \).
The Dependent Function Set

- The dependent function set is the unproblematic part of the LF.
- The dependent function set is similar to the non-dependent function set (e.g. \( A \to B \)), except that we allow that the second set to depend on an element of the first set.
- Notation: \( (x : A) \to B \), for the set of functions \( f \) which map an element \( a : A \) to an element of \( B[x := a] \).
- In set-theoretic notation this is:

\[
\{ f \mid f \text{ function} \\
\land \text{dom}(f) = A \\
\land \forall a \in A. f(a) \in B[x := a] \}
\]

Rules of the Dependent Function Set

**Formation Rule**

\[
\frac{A : \text{Set} \quad x : A \Rightarrow B : \text{Set}}{(x : A) \to B : \text{Set}} \quad (\to-F)
\]

**Introduction Rule**

\[
\frac{x : A \Rightarrow b : B}{(\lambda x : A. b) : (x : A) \to B} \quad (\to-I)
\]

**Elimination Rule**

\[
\frac{f : (x : A) \to B \quad a : A}{f \ a : B[x := a]} \quad (\to-\text{El})
\]

**Equality Rule**

\[
\frac{x : A \Rightarrow b : B \quad a : A}{(\lambda x : A. b) \ a = b[x := a] : B[x := a]} \quad (\to-\text{Eq})
\]

The \( \eta \)-Rule

The \( \eta \)-rule has a special status:

**\( \eta \)-Rule**

\[
\frac{f : (x : A) \to B}{f = (\lambda x : A. f \ x) : (x : A) \to B} \quad (\to-\eta)
\]

- The \( \eta \)-rule expresses that every element of \( (x : A) \to B \) is of the form \( \lambda x : A. \text{something} \).
- The \( \eta \)-rule cannot be derived, if the element in question is a variable.
The Dependent Function Set in Agda

- The dependent function set is built into Agda with notation

\[(x : A) \rightarrow B\]

- Elements of \((x : A) \rightarrow B\) are introduced by using
  - either \(\lambda\)-abstraction, i.e. we can define
    \[
    f : (x : A) \rightarrow B \\
    f = \lambda x \rightarrow b
    \]
  - Requires that \(b : B\) depending on \(x : A\).
  - Note that the type \(B\) of \(b\) depends on \(x : A\).
  - or by writing
    \[
    f : (x : A) \rightarrow B \\
    f x = b
    \]

Elimination is application using the same notation as before.
- E.g., if \(f : (x : A) \rightarrow B\) and \(a : A\), then \(f a : B[x := a]\).

Implication

- A proof of \(A \rightarrow B\) is a function which takes a proof of \(A\) and returns a proof of \(B\).
- So implication is nothing but the function type.

Example

- \(\text{lemma} : A \rightarrow A\)
- \(\text{lemma} a = a\)
- \(\text{lemma2} : (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow A \rightarrow C\)
- \(\text{lemma2} f g a = g (f a)\)
Universal Quantification

- $\forall x : A.B$ is true iff, for all $x : A$ there exists a proof of $B$ (with that $x$).
- Therefore a proof of $\forall x : A.B$ is a function, which takes an $x:A$ and computes an element of $B$.
- Therefore the set of proofs of $\forall x : A.B$ is the set of functions, mapping an element $x : A$ to an element of $B$.
- This set is just the dependent function set $(x : A) \rightarrow B$.
- Therefore we can identify $\forall x : A.B$ with $(x : A) \rightarrow B$.

Example: Equality on $\mathbb{N}$

- We define equality on $\mathbb{N}$.
- First we introduce the true and false formulas:
  
  - $\bot$ is defined as $\emptyset$
  - $\top$ has one proof, namely the trivial proof $\text{triv}$

  
  data $\bot : \text{Set}$ where
  data $\top : \text{Set}$ where
  
  - $\text{triv} : \top$

- $== : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set}$
  - $\text{zero} == \text{zero} = \top$
  - $\text{zero} == \text{S} \text{m} = \bot$
  - $\text{S} \text{n} == \text{zero} = \bot$
  - $\text{S} \text{n} == \text{S} \text{m} = n == m$

Example Proof of Reflexivity of $==$

- $\forall x : A.B$ is represented by $(x : A) \rightarrow B$ in Agda.
  - Remember that $\forall x : A.B$ is another notation for $\forall x : A.B$.

refl : $(n : \mathbb{N}) \rightarrow n == n$
refl zero = triv
refl (S n) = refl n
Above we were already using types such as

$$C : B \rightarrow \text{Set}$$

This doesn’t type check yet, since we would need

$$B \rightarrow \text{Set} : \text{Set}$$

and our rules allow this only if we had

$$\text{Set} : \text{Set}$$

Adding

$$\text{Set} : \text{Set}$$
as a rule results however in an inconsistent theory:

- using this rule we can prove everything, especially false formulas.
  The corresponding paradox is called Girard’s paradox.

Instead we introduce a new level on top of Set called Type.

- So besides judgements $$A : \text{Set}$$ we have as well judgements of the form
  $$A : \text{Type}$$

- One rule will especially express
  $$\text{Set} : \text{Type}$$

- Elements of Type are types, elements of Set are small types.
We add rules asserting that if \( A : \text{Set} \) then \( A : \text{Type} \).

Further we add rules asserting that Type is closed under the elements of Set and the function type.

Since \( \text{Set} : \text{Type} \) we get therefore (by closure under the function type):

\[
\mathbb{B} \rightarrow \text{Set} : \text{Type}
\]

### Rules for Set (as an Element of Type)

#### Formation Rule for Set

\[
\text{Set} : \text{Type} \quad (\text{SetIsType})
\]

#### Every Set is a Type

\[
\frac{A : \text{Set}}{A : \text{Type}} \quad (\text{Set2Type})
\]

### Closure of Type

Further we add rules stating that Type is closed under the dependent function type:

**Closure of Type under the dependent function type**

\[
\frac{A : \text{Type} \quad x : A \Rightarrow B : \text{Type} \quad (\Rightarrow \text{FType})}{(x : A) \rightarrow B : \text{Type}}
\]
The construct `data` in Agda is much more powerful than what is covered by type theoretic rules.

In general we can define now sets having arbitrarily many constructors with arbitrarily many arguments of arbitrary types.

```agda
data A : Set where
  C_1 : (a_1 : A^1) → (a_2 : A^2) → ··· (a_{n_1} : A^{n_1}) → A
  C_2 : (a_1 : A^2) → (a_2 : A^2) → ··· (a_{n_2} : A^{n_2}) → A
  ···
  C_m : (a_1 : A^m) → (a_2 : A^m) → ··· (a_{n_m} : A^{n_m}) → A
```

The idea is that `A` as before is the least set such that we have constructors:

- `C_i : (a_{i_1} : A^{i_1}) → ··· (a_{i_{n_i}} : A^{i_{n_i}}) → A`

where a constructor always constructs new elements.

In other words the elements of `A` are exactly those constructed by those constructors.

In the types `A_{ij}` we can make use of `A`.

- However, it is difficult to understand `A`, if we have negative occurrences of `A`.

Example:

```agda
data A : Set where
  C : (A → A) → A
```

What is the least set `A` having a constructor

```agda
C : (A → A) → A
```
Strictly Positive Algebraic Types

- If we have constructed some elements of \( A \) already, find a function \( f : A \rightarrow A \), and add \( C \ f \) to \( A \), then \( f \) might no longer be a function \( A \rightarrow A \). 
  
- In fact, the termination checker issues a warning, if we define \( A \) as above.
- We shouldn’t make use of such definitions.

A “good” definition is the set of lists of natural numbers, defined as follows:

\[
\text{data NList : Set where}
\]
\[
[ ] : \text{NList}
\]
\[
\_ :: _ : \text{N} \rightarrow \text{NList} \rightarrow \text{NList}
\]

- The constructor \( \_ :: \_ \) of NList refers to NList, but in a positive way:
  We have: if \( a : \text{N} \) and \( l : \text{NList} \), then
  \[
  (a :: l) : \text{NList}.
  \]

In general:

\[
\text{data A : Set where}
\]
\[
C_1 : (a_1 : A_1^1) \rightarrow (a_2 : A_2^1) \rightarrow \cdots \rightarrow (a_{n_1} : A_{n_1}^{1}) \rightarrow A
\]
\[
C_2 : (a_1 : A_1^2) \rightarrow (a_2 : A_2^2) \rightarrow \cdots \rightarrow (a_{n_2} : A_{n_2}^{2}) \rightarrow A
\]
\[\cdots\]
\[
C_m : (a_1 : A_1^m) \rightarrow (a_2 : A_2^m) \rightarrow \cdots \rightarrow (a_{n_m} : A_{n_m}^{m}) \rightarrow A
\]

is a strictly positive algebraic type, if all \( A_{ij} \) are
- either types which don’t make use of \( A \)
- or are \( A \) itself.
- And if \( A \) is a strictly positive algebraic type, then \( A \) is acceptable.
The definitions of finite sets, Σ A B, A + B and ℕ were strictly positive algebraic types.

One further Example

- The set of binary trees can be defined as follows:

```
data BinTree : Set where
leaf : BinTree
branch : Bintree → Bintree → BinTree
```

- This is a strictly positive algebraic type.

Extensions of Strictly Positive Algebraic Types

- An often used extension is to define several sets simultaneously inductively.

- Example: the even and odd numbers:

```
mutable
  data Even : Set where
    Z : Even
    S : Odd → Even

  data Odd : Set where
    S' : Even → Odd
```

- In such examples the constructors refer strictly positive to all sets which are to be defined simultaneously.

Extensions of Strictly Positive Algebraic Types

- We can even allow A_i = B_1 → A or even A_i = B_1 → ⋯ → B_l → A, where A is one of the types introduced simultaneously.

- Example (called “Kleene’s O”):

```
data O : Set where
  leaf : O
  succ : O → O
  lim : (N → O) → O
```

- The last definition is unproblematic, since, if we have f : N → O and construct lim f out of it, adding this new element to O doesn’t destroy the reason for adding it to O.

- So again O can be “constructed”.

Functions $f$ from strictly positive algebraic types can now be defined by case distinction as before.

For termination we need only that in the definition of $f$, when have to define $f(C\ a_1 \cdots\ a_n)$, we can refer only to $f$ applied to elements used in $C\ a_1 \cdots\ a_n$.

For instance

- in the Bintree example, when defining $f : \text{Bintree} \to A$ by case-distinction, then the definition of $f(\text{branch}\ l\ r)$ can make use of $f\ l$ and $f\ r$.

- In the example of $O$, when defining $g : O \to A$ by case-distinction, then the definition of $g(\text{lim}\ f)$ can make use of $g(f\ n)$ for all $n : \mathbb{N}$. 

Type Theory and Interactive Theorem Proving

Key Philosophical Principles of Martin-Löf Type Theory

Setup of Martin-Löf Type Theory

Basic Types in Martin-Löf Type Theory

The Logical Framework

Inductive Data Types (Algebras) in Type Theory

Coinductive Data Types (Coalgebras) in Type Theory
Codata Types

- Idea of Codata Types: non-well-founded versions of inductive data types:
  \[
  \text{codata Stream : Set where} \\
  \text{cons : N \to Stream \to Stream}
  \]
- Same definition as inductive data type but we are allowed to have infinite chains of constructors
  \[
  \text{cons } n_0 (\text{cons } n_1 (\text{cons } n_2 \cdots ))
  \]
- **Problem 1:** Non-normalisation.
- **Problem 2:** Equality between streams is equality between all \( n_i \), and therefore undecidable.
- **Problem 3:** Underlying assumption is
  \[
  \forall s : \text{Stream}. \exists n, s'. s = \text{cons } n s'
  \]
  which results in undecidable equality.

Subject Reduction Problem

- In order to repair problem of normalisation restrictions on reductions were introduced.
- Resulted in Coq in a long known problem of **subject reduction**.
- In order to avoid this, in Agda dependent elimination for coalgebras disallowed.
  - Makes it difficult to use.

Coalgebraic Formulation of Coalgebras

- Solution is to follow the long established categorical formulation of coalgebras.
- Final coalgebras will be replaced by weakly final coalgebras.
- Two streams will be equal if the programs producing them reduce to the same normal form.

Algebras and Coalgebras

- Algebraic data types correspond to initial algebras.
  - \( \mathbb{N} \) as an algebra can be represented as introduction rules for \( \mathbb{N} \):
    \[
    \begin{align*}
    \text{zero} : & \mathbb{N} \\
    \mathbb{S} & : \mathbb{N} \to \mathbb{N}
    \end{align*}
    \]
- Coalgebra obtained by “reversing the arrows”.
  - Stream as a coalgebra can be expressed as as elimination rules for it:
    \[
    \begin{align*}
    \text{head} & : \text{Stream} \to \mathbb{N} \\
    \text{tail} & : \text{Stream} \to \text{Stream}
    \end{align*}
    \]
Coinductive Data Types (Coalgebras) in Type Theory

Weakly Initial Algebras and Final Coalgebras

- \( \mathbb{N} \) as a weakly initial algebra corresponds to iteration (elimination rule): For \( A : \text{Set}, \ a : A, \ f : A \rightarrow A \) there exists
  
  \[
  g : \mathbb{N} \rightarrow A \\
  g \ \text{zero} = a \\
  g \ (S \ n) = f \ (g \ n)
  \]
  
  (or \( g \ n = f^n a \)).

- Stream as a weakly final coalgebra corresponds to coiteration or guarded iteration (introduction rule):
  
  For \( A : \text{Set}, \ f_0 : A \rightarrow \mathbb{N}, \ f_1 : A \rightarrow A \) there exists \( g \) s.t.
  
  \[
  g : A \rightarrow \text{Stream} \\
  \text{head} \ (g \ a) = f_0 \ a \\
  \text{tail} \ (g \ a) = g \ (f_1 \ a)
  \]

Example

- Using coiteration we can define
  
  \[
  \text{inc} : \mathbb{N} \rightarrow \text{Stream} \\
  \text{head} \ (\text{inc} \ n) = n \\
  \text{tail} \ (\text{inc} \ n) = \text{inc} \ (n + 1)
  \]

Recursion and Corecursion

- \( \mathbb{N} \) as an initial algebra corresponds to uniqueness of \( g \) above.
  - Allows to derive primitive recursion:
    
    For \( A : \text{Set}, \ a : A, \ f : (\mathbb{N} \times A) \rightarrow A \) there exists
    
    \[
    g : \mathbb{N} \rightarrow A \\
    g \ \text{zero} = a \\
    g \ (S \ n) = f \ (n, (g \ n))
    \]
  - Stream as a final coalgebra corresponds to uniqueness of \( h \).
    - Allows to derive primitive corecursion:
      
      For \( A : \text{Set}, \ f_0 : A \rightarrow \mathbb{N}, \ f_1 : A \rightarrow (\text{Stream} + A) \) there exists
      
      \[
      g : A \rightarrow \text{Stream} \\
      \text{head} \ (g \ a) = f_0 \ a \\
      \text{tail} \ (g \ a) = s \quad \text{if} \ f_1 \ a = \text{inl} \ s \\
      \text{tail} \ (g \ a) = g \ a' \quad \text{if} \ f_1 \ a = \text{inr} \ a'
      \]

Recursion vs Iteration

- Using recursion we can define inverse case of the constructors of \( \mathbb{N} \) as follows:
  
  \[
  \text{case} : \mathbb{N} \rightarrow (1 + \mathbb{N}) \\
  \text{case zero} = \text{inl} \\
  \text{case} \ (S \ n) = \text{inr} \ n
  \]

- Using iteration, we cannot make use of \( n \) and therefore case is defined inefficiently:
  
  \[
  \text{case} : \mathbb{N} \rightarrow (1 + \mathbb{N}) \\
  \text{case zero} = \text{inl} \\
  \text{case} \ (S \ n) = \text{caseaux} \ (\text{case n})
  \]

  \[
  \text{caseaux} : (1 + \mathbb{N}) \rightarrow (1 + \mathbb{N}) \\
  \text{caseaux} \ \text{inl} = \text{inr} \ \text{zero} \\
  \text{caseaux} \ (\text{inr} \ n) = \text{inr} \ (S \ n)
  \]
Coinductive Data Types (Coalgebras) in Type Theory

Definition of \textit{pred}

- One way of defining \textit{pred} by iteration is by defining first \textit{case} and then to define

\[
\begin{align*}
predaux : (1 + \mathbb{N}) &\to \mathbb{N} \\
predaux \text{ inl} &= \text{zero} \\
predaux \text{ (inr } n) &= n \\
pred : \mathbb{N} &\to \mathbb{N} \\
pred n &= \text{predaux} \text{ (case } n) \\
\end{align*}
\]

Corecursion vs Coiteration

- Definition of \textit{cons} (inverse of the destructors) using coiteration inefficient:

\[
\begin{align*}
\text{cons} : \mathbb{N} &\to \text{Stream} \to \text{Stream} \\
\text{head} \text{ (cons } n s) &= n \\
\text{tail} \text{ (cons } n s) &= \text{cons} \text{ (head } s) \text{ (tail } s) \\
\end{align*}
\]

- Using primitive corecursion we can define more easily

\[
\begin{align*}
\text{cons} : \mathbb{N} &\to \text{Stream} \to \text{Stream} \\
\text{head} \text{ (cons } n s) &= n \\
\text{tail} \text{ (cons } n s) &= s \\
\end{align*}
\]

Induction - Coinduction?

- Induction is dependent primitive recursion:

For \( A : \mathbb{N} \to \text{Set} \), \( a : A \text{ zero} \), \( f : (n : \mathbb{N}) \to A \text{ n} \to A \text{ (S } n) \) there exists

\[
\begin{align*}
g : (n : \mathbb{N}) &\to A \text{ n} \\
g \text{ zero} &= a \\
g \text{ (S } n) &= f \text{ n} \text{ (g } n) \\
\end{align*}
\]

- Equivalent to uniqueness of arrows with respect to propositional equality and interpreting equality on arrows extensionally.

- Uniqueness of arrows in final coalgebras expresses that equality is bisimulation equality.

- How to dualise \textbf{dependent} primitive recursion?

Weakly Final Coalgebra

- Equality for final coalgebras is undecidable:

Two streams

\[
\begin{align*}
s &= (a_0 , a_1 , a_2 , \ldots) \\
t &= (b_0 , b_1 , b_2 , \ldots)
\end{align*}
\]

are equal iff \( a_i = b_i \) for all \( i \).

- Even the weak assumption

\[
\forall s. \exists n, s'. s = \text{cons } n s'
\]

results in an undecidable equality.

- Weakly final coalgebras obtained by omitting uniqueness of \( g \) in diagram for coalgebras.

- However, one can extend schema of coiteration as above, and still preserve decidability of equality.

- Those schemata are usually not derivable in weakly final coalgebras.
We see now that elements of coalgebras are defined by their observations:
An element \( s \) of Stream is anything for which we can define
\[
\text{head } s : \mathbb{N} \\
\text{tail } s : \text{Stream}
\]
This generalises the function type.
Functions are as well determined by observations.
\[
\text{An } f : A \rightarrow B \text{ is any program which if applied to } a : A \text{ returns some } b : B.
\]
Inductive data types are defined by construction
coalgebraic data types and functions by observations.

Objects in Object-Oriented Programming are types which are defined by their observations.
Therefore objects are coalgebraic types by nature.

We can define now functions by patterns and copatterns.
Example define stream:
\[
f n = \\
n, n, n-1, n-1, \ldots 0, 0, N, N, N-1, N-1, \ldots 0, 0, N, N, N-1, N-1,
\]

\[
f : \mathbb{N} \rightarrow \text{Stream} \\
f = ?
\]

Copattern matching on \( f : \mathbb{N} \rightarrow \text{Stream} \):
\[
f : \mathbb{N} \rightarrow \text{Stream} \\
f n = ?
\]

\[
f : \mathbb{N} \rightarrow \text{Stream} \\
f n = ?
\]

Copattern matching on \( f n : \text{Stream} \):
\[
f : \mathbb{N} \rightarrow \text{Stream} \\
\text{head } (f n) = ?
\]