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

Anton Setzer

Swansea University, Swansea UK

Lisbon, 9 September 2015

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** IntWe **solve the goal** using a

$$
\begin{array}{lcl}
a & : & \mathrm{Int} \\
a & = & \cdots \\
\\
f & : & \mathrm{Int} \to \mathrm{String} \\
f & = & \cdots \\
\\
p & : & \mathrm{String} \\
p & = & \{!\ \ !\}f\ \{!\ \ !\}f\ \{!\ \ !\}f\ a
\end{array}
$$

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 $x == 0$ 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**
$$(x : A) \to B(x)$$
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 $\mathrm{Mat}(n, m)$ of $n \times m$ matrices depends on n, m.
- Matrix multiplication has type

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

- In simply typed languages we can only have

$$\mathrm{matmult} : \mathrm{Mat} \to \mathrm{Mat} \to \mathrm{Mat}$$

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

$$\begin{array}{l}
\mathrm{numberOfComponents} : \mathrm{Lecture} \to \mathbb{N} \\
\mathrm{numberOfComponents}\ l = \cdots \\
\\
\mathrm{Marks} : (l : \mathrm{Lecture}) \to \mathrm{Set} \\
\mathrm{Marks}\ l = \mathrm{Mark}^{\mathrm{numberOfComponents}\ l} \\
\\
\mathrm{Weighting} : (l : \mathrm{Lecture}) \to \mathrm{Set} \\
\mathrm{Weighting}\ l = \mathrm{Percentage}^{\mathrm{numberOfComponents}\ l} \\
\\
\mathrm{finalMark} : (l : \mathrm{Lecture}) \to \mathrm{Marks}\ l \to \mathrm{Weighting}\ l \to \mathrm{Mark} \\
\mathrm{finalMark}\ l\ m\ w = \cdots
\end{array}$$

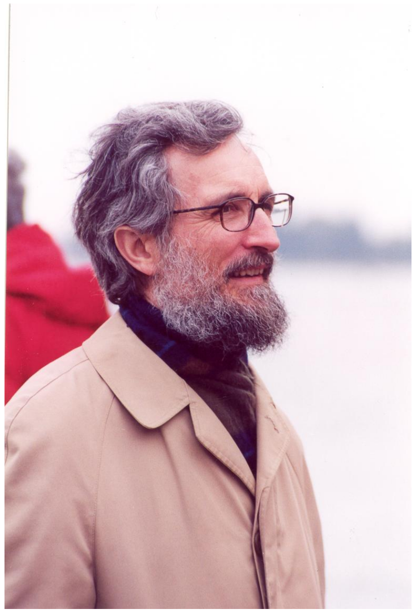
Generative Programming

- 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.

Per Martin-Löf (Stockholm)



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.

Key Philosophical Principles of Martin-Löf Type Theory

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:

zero	
$\mathrm{suc\ zero}$	
$\mathrm{suc\ zero} + \mathrm{suc\ zero}$	
$[]$	(empty list)
$\mathrm{cons\ zero}\ []$	(result of adding in front of the empty list zero)

Key Philosophical Principles of Martin-Löf Type Theory

Objects of Type Theory

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

Key Philosophical Principles of Martin-Löf Type Theory

Beyond Finitism

- 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**.

Types

- 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

$$\begin{array}{l} \mathrm{zero} : \mathbb{N} \\ \text{if } n : \mathbb{N} \text{ then } \mathrm{suc}\ n : \mathbb{N} \end{array}$$

- This is written as rules

$$\mathrm{zero} : \mathbb{N} \qquad \frac{n : \mathbb{N}}{\mathrm{suc}\ n : \mathbb{N}}$$

- We can conclude for instance

$$\mathrm{suc}\ (\mathrm{suc}\ \mathrm{zero}) : \mathbb{N}$$

Example: Natural Numbers

- Furthermore if we have another type B, i.e.

$$B : \mathrm{Set}$$

  and if we have

$$\begin{array}{l} b : B \\ g : B \to B \end{array}$$

  we can form

$$\begin{array}{lcl} h : \mathbb{N} \to B & & \\ h\ \mathrm{zero} & = & b \\ h\ (\mathrm{suc}\ n) & = & g\ (h\ n) \end{array}$$

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

$$h\ n = g^n\ b$$

- 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)

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

  which contains a code for the set of natural numbers

$$\widehat{\mathbb{N}} : \mathrm{U}$$

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

$$\frac{u : \mathrm{U}}{\mathrm{T}\ u : \mathrm{Set}}$$

- Then we can add a rule

$$\mathrm{T}\ \widehat{\mathbb{N}} = \mathbb{N} : \mathrm{Set}$$

- $\widehat{\mathbb{N}}$ 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
$$a : A \text{ s.t. } B(a) \text{ is true}$$

Constructive Mathematics

- Similarly from a proof
$$p : A \vee B$$
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
$$A \vee \neg A$$
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
$$A \wedge B$$
is given by a
$$\text{proof of } A \text{ and a proof of } B$$
- A proof of
$$A \vee B$$
is given by
$$\text{a proof of } A \text{ or a proof of } B$$
plus the information which of the two holds.

BHK-Interpretation of Logical Connectives

- A proof of
$$A \to B$$
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 $B(a)$

BHK-Interpretation of Logical Connectives

- There is no proof of falsity written as
$$\perp$$
- We define
$$\neg A := A \to \perp$$
so a proof of
$$\neg A$$
is a function which
converts a proof of A into a (non-existent) proof of $\perp$

Intuitionistic Logic

- We don't obtain stability
$$\neg\neg A \to A$$
- 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 $\perp$.
  - 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 intuitioniscally provable.
- Double negation interpretation (not part of this course).

Double Negation Interpretation

- Easy to see with $\vee$:
  Intuitionistically we have
  $$\neg(\neg(A \vee B)) \leftrightarrow \neg(\neg A \wedge \neg B)$$
  If we replace
  $$A \vee B$$
  by
  $$A \vee^{\mathrm{int}} B := \neg(\neg A \wedge \neg B)$$
  then
  $$A \vee^{\mathrm{int}} B$$
  behaves intuitionistically (with double negated formulas) like classical $\vee$.
- Especially tertium non datur is provable
  $$A \vee^{\mathrm{int}} \neg A = \neg(\neg A \wedge \neg\neg A)$$

Conclusion (Key Philosophical Principles of MLTT)

- This concludes the introduction into the philosophical principles of Martin-Löf Type Theory.
- We will in the next section go through the setup of Martin-Löf Type Theory with the terminology by Martin-Löf.

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
    $$A : \mathrm{Set}$$
  - A and B are equal types written as
    $$A = B : \mathrm{Set}$$
  - a is an element of type A written as
    $$a : A$$
  - a, b are equal elements of type A written as
    $$a = b : A$$

$s \longrightarrow t$ vs $s = t$

- The notion of reduction
$$s \longrightarrow t$$
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 $\longrightarrow$ or equivalence relation containing $\longrightarrow$.
- In most rules when concluding
$$s = t : A$$
it is actually the case that we have a reduction
$$s \longrightarrow t$$

$s \longrightarrow t$ vs $s = t$

- The notion
$$s \longrightarrow t$$
doesn't occur in the formal theory of Martin-Löf Type Theory, but only when implementing it.

Dependent Judgements

- We have as well **dependent judgements**, for instance for expressing
$$\text{if } x : \mathbb{N} \text{ then } \mathrm{suc}\ x : \mathbb{N}$$
which we write
$$x : \mathbb{N} \Rightarrow \mathrm{suc}\ x : \mathbb{N}$$
- Examples:
$$\begin{array}{lcl} x : \mathbb{N}, y : \mathbb{N} & \Rightarrow & x + y : \mathbb{N} \\ x : \mathbb{N} & \Rightarrow & x + \mathrm{zero} = x : \mathbb{N} \\ x : \mathrm{List} & \Rightarrow & \mathrm{Sorted}\ x : \mathrm{Set} \\ & \Rightarrow & \mathrm{Sorted}\ [] = \mathrm{True} : \mathrm{Set} \end{array}$$

Examples of Dependent Judgements

- 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
$$\begin{array}{lclcl} B(\vec{x}) : \mathrm{Set} & \text{or} & B(\vec{x}) = B'(\vec{x}) : \mathrm{Set} & \text{or} \\ b(\vec{x}) : B(\vec{x}) & \text{or} & b(\vec{x}) = b'(\vec{x}) : B(\vec{x}) \end{array}$$

Judgements in Agda

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

$$n : \mathbb{N}$$
$$n = \mathrm{zero}$$

$$f : \mathbb{N} \to \mathbb{N}$$
$$f \ \mathrm{zero} = \mathrm{suc}\ \mathrm{zero}$$
$$f \ (\mathrm{suc}\ m) = \mathrm{suc}\ (\mathrm{suc}(f \ m))$$

- $=$ 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

$$\mathrm{suc}\ \mathrm{zero}$$

Judgements in Agda

- 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.

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

- **Introduction Rules:**
  They introduce elements of a type, e.g.

$$\mathrm{zero} : \mathbb{N} \qquad \frac{n : \mathbb{N}}{\mathrm{suc}\ n : \mathbb{N}}$$

Four Kinds of Rules for each Type

- **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

$$\frac{B : \mathrm{Set} \qquad b : B \qquad g : B \to B \qquad n : \mathbb{N}}{h \ n : B}$$

  where

$$h := \mathrm{iter}\ B\ b\ g$$

Four Kinds of Rules for each Type

- **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)

$$\frac{B : \mathrm{Set} \qquad b : B \qquad g : B \to B}{h\ \mathrm{zero} = b : B}$$

$$\frac{B : \mathrm{Set} \qquad b : B \qquad g : B \to B \qquad n : \mathbb{N}}{h\ (\mathrm{suc}\ n) = g\ (h\ n) : B}$$

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

$$\frac{n : \mathbb{N}}{\mathrm{suc}\ n : \mathbb{N}}$$

is

$$\frac{n = m : \mathbb{N}}{\mathrm{suc}\ n = \mathrm{suc}\ m : \mathbb{N}}$$

Canonical vs Non-Canonical Elements

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

$$\mathrm{zero} \quad \text{and} \quad \mathrm{suc}$$

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

$$\mathrm{zero} \qquad \mathrm{suc}\ (\mathrm{zero} + \mathrm{zero})$$

    where $+$ is defined using elimination rules.
- Elements introduced by an elimination rule are **non-canonical** elements. For instance

$$\mathrm{zero} + \mathrm{zero}$$

- 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

$$\mathrm{suc}\ (\mathrm{zero} + \mathrm{zero}) \longrightarrow \mathrm{suc}\ \mathrm{zero}$$

- In case of a function type $\lambda x.t$ is considered to be canonical.
- Note that in

$$\lambda x.x : \mathbb{N} \to \mathbb{N}$$

  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 $s \longrightarrow \mathrm{zero}$, $t \longrightarrow \mathrm{suc}\ t'$ there is no need to evaluate t'.
  - If they have the same constructor, e.g. $s \longrightarrow \mathrm{suc}\ s'$ $t \longrightarrow \mathrm{suc}\ t'$ then we compare s' and t'.

The Type of Booleans

- One of the Simples types is the type of Booleans.
- **Formation rule:**
  $$\mathbb{B} : \mathrm{Set}$$
- **Introduction rules:**
  $$\mathrm{tt} : \mathbb{B} \qquad \mathrm{ff} : \mathbb{B}$$
- **Elimination rule:**

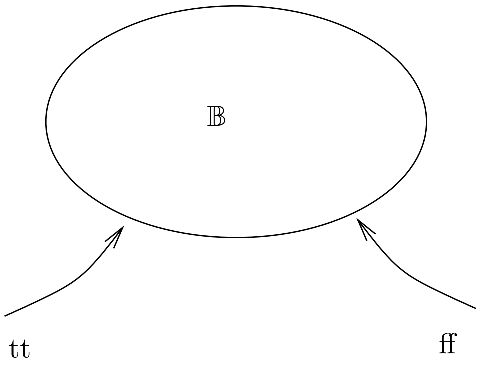
$$\frac{x : \mathbb{B} \Rightarrow C(x) : \mathrm{Set} \qquad \mathit{step}_{\mathrm{tt}} : C(\mathrm{tt}) \qquad \mathit{step}_{\mathrm{ff}} : C(\mathrm{ff}) \qquad b : \mathbb{B}}{\mathrm{elim}_{\mathbb{B}}(\mathit{step}_{\mathrm{tt}}, \mathit{step}_{\mathrm{ff}}, b) : C(b)}$$

Basic Types: Type of Booleans

- **Equality rules:**

$$\begin{array}{rcl} \mathrm{elim}_{\mathbb{B}}(\mathit{step}_{\mathrm{tt}}, \mathit{step}_{\mathrm{ff}}, \mathrm{tt}) & = & \mathit{step}_{\mathrm{tt}} : C(\mathrm{tt}) \\ \mathrm{elim}_{\mathbb{B}}(\mathit{step}_{\mathrm{tt}}, \mathit{step}_{\mathrm{ff}}, \mathrm{ff}) & = & \mathit{step}_{\mathrm{ff}} : C(\mathrm{ff}) \end{array}$$

Visualisation (Booleans)



2 Constructors, both no arguments.

Booleans in Agda

$$
\begin{array}{l}
\mathrm{data}\ \mathbb{B} : \mathrm{Set}\ \mathrm{where} \\
\quad \mathrm{tt} \ : \ \mathbb{B} \\
\quad \mathrm{ff} \ : \ \mathbb{B} \\
\\
\neg : \mathbb{B} \to \mathbb{B} \\
\neg\, \mathrm{tt} \ = \ \mathrm{ff} \\
\neg\, \mathrm{ff} \ = \ \mathrm{tt}
\end{array}
$$

Finite Types

- Similar versions for types with $0, 1, 3, 4, \ldots$ elements.
- Special case $\emptyset$.

Empty Type

- **Formation rule:**
$$\emptyset : \mathrm{Set}$$
- **Introduction rules:**
  There is no introduction rule.
- **Elimination rule:**
$$\frac{x : \emptyset \Rightarrow C(x) : \mathrm{Set} \qquad e : \emptyset}{\mathrm{efq}(e) : C(e)}$$
- **Equality rules:**
  There is no equality rule.

$\emptyset$ in Agda

$\text{data } \emptyset : \text{Set where}$

$\text{efq} : \emptyset \to A$
$\text{efq } ()$

- - () stands for the empty case distinction
- - and - - starts a comment

The Logical Framework (LF)

- When writing elimination rules we need to deal with notions such as
  - $C(x)$ is a set depending on $x : \mathbb{B}$.
  - instantiate $x = \text{tt}$ and get $C(\text{tt})$.
- Idea of the logical framework (LF) is
  - Instead of saying
    $$x : \mathbb{B} \Rightarrow C(x) : \text{Set}$$
    we write
    $$C : \mathbb{B} \to \text{Set}$$
  - Then we can apply C to 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:**
  $$\mathbb{B} : \text{Set}$$
- **Introduction rules:**
  $$\text{tt} : \mathbb{B} \qquad \text{ff} : \mathbb{B}$$
- **Elimination rule:**
  $$\frac{C : \mathbb{B} \to \text{Set} \qquad step_{\text{tt}} : C\ \text{tt} \qquad step_{\text{ff}} : C\ \text{ff} \qquad b : \mathbb{B}}{\text{elim}_{\mathbb{B}}\ C\ step_{\text{tt}}\ step_{\text{ff}}\ b : C\ b}$$
- **Equality rules:**
  $$\begin{aligned} \text{elim}_{\mathbb{B}}\ C\ step_{\text{tt}}\ step_{\text{ff}}\ \text{tt} &= step_{\text{tt}} : C\ \text{tt} \\ \text{elim}_{\mathbb{B}}\ C\ step_{\text{tt}}\ step_{\text{ff}}\ \text{ff} &= step_{\text{ff}} : C\ \text{ff} \end{aligned}$$

Rules for Booleans Using the LF

- We can even write

$$\begin{aligned}\mathrm{elim}_{\mathbb{B}} : (C : \mathbb{B} \to \mathrm{Set})\\ &\to C\ \mathrm{tt}\\ &\to C\ \mathrm{ff}\\ &\to \mathbb{B}\\ &\to \mathrm{Set}\end{aligned}$$

The Disjoint Union

- **Formation rule:**

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

- **Introduction rules:**

$$\frac{a : A}{\mathrm{inl}\ a : A + B} \qquad \frac{b : B}{\mathrm{inr}\ b : A + B}$$

The Disjoint Union

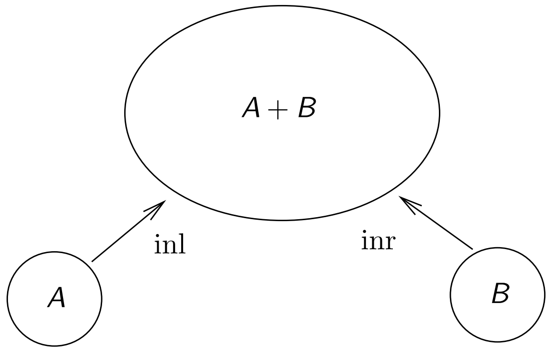
- **Elimination rule:**

$$\frac{\begin{array}{c}C : A + B \to \mathrm{Set}\\ step_{\mathrm{inl}} : (x : A) \to C\ (\mathrm{inl}\ x)\\ step_{\mathrm{inr}} : (x : B) \to C(\mathrm{inr}\ x)\\ c : A + B\end{array}}{\mathrm{elim}_{+}\ C\ step_{\mathrm{inl}}\ step_{\mathrm{inr}}\ c : C\ c}$$

- **Equality rules:**

$$\begin{aligned}\mathrm{elim}_{+}\ C\ step_{\mathrm{inl}}\ step_{\mathrm{inr}}\ (\mathrm{inl}\ a) &= step_{\mathrm{inl}}\ a : C\ (\mathrm{inl}\ a)\\ \mathrm{elim}_{+}\ C\ step_{\mathrm{inl}}\ step_{\mathrm{inr}}\ (\mathrm{inr}\ b) &= step_{\mathrm{inr}}\ b : C\ (\mathrm{inr}\ b)\end{aligned}$$

Visualisation $(A + B)$



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

$+$ as $\vee$

- A proof of $A \vee B$ is a proof of A or a proof of B.
- So $A \vee B$ is just $A + B$.

$A \vee B$ in Agda

$$
\begin{array}{l}
\mathrm{data}\ \_\vee\_\ (A\ B : \mathrm{Set}) : \mathrm{Set\ where} \\
\quad \mathrm{inl} \quad : \quad A \to A \vee B \\
\quad \mathrm{inr} \quad : \quad B \to A \vee B
\end{array}
$$

-- $\_\vee\_$ denotes infix operator
-- We postulate (i.e. assume) some sets

$$
\begin{array}{l}
\mathrm{postulate}\ A : \mathrm{Set} \\
\mathrm{postulate}\ B : \mathrm{Set}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{lemma} : A \vee B \to B \vee A & & \\
\mathrm{lemma}\ (\mathrm{inl}\ a) & = & \mathrm{inr}\ a \\
\mathrm{lemma}\ (\mathrm{inr}\ b) & = & \mathrm{inl}\ b
\end{array}
$$

The Σ-Type

- **Formation rule:**

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

- **Introduction rule:**

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

The Σ-Type

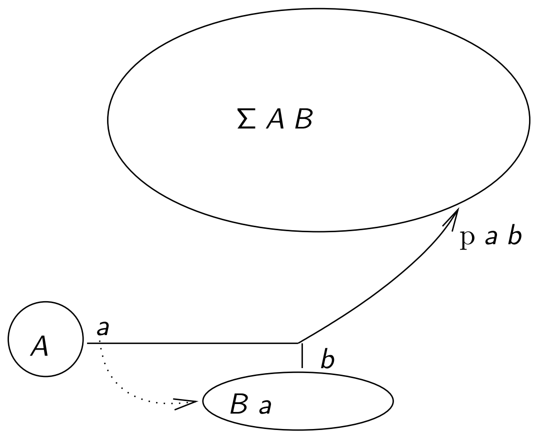
- **Elimination rule:**

$$\frac{\begin{array}{c} C : \Sigma\ A\ B \to \mathrm{Set} \\ step : (a : A, b : B\ a) \to C\ (\mathrm{p}\ a\ b) \\ c : \Sigma\ A\ B \end{array}}{\mathrm{elim}_{\Sigma}\ C\ step\ c : C\ c}$$

- **Equality rule:**

$$\mathrm{elim}_{\Sigma}\ C\ step\ (\mathrm{p}\ a\ b) = step\ a\ b : C\ (\mathrm{p}\ a\ b)$$

Visualisation ($\Sigma(A, B)$)



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

$\exists$ as Σ

- With the LF, a formula depending on $x : A$ is a

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

- A proof of $\exists x : A.B\ x$ is
  - an $a : A$
  - together with a $b : B\ a$
- That's just an element of

$$\Sigma\ A\ B$$

$\Sigma\ A\ B$ in Agda

$$\begin{array}{l} \mathrm{data}\ \Sigma\ (A : \mathrm{Set})\ (B : A \to \mathrm{Set}) : \mathrm{Set}\ \mathrm{where} \\ \quad \mathrm{p} : (a : A) \to B\ a \to \Sigma\ A\ B \end{array}$$

$$\begin{array}{l} \mathrm{postulate}\ A : \mathrm{Set} \\ \mathrm{postulate}\ B : A \to \mathrm{Set} \end{array}$$

$$\begin{array}{l} \pi_0 : \Sigma\ A\ B \to A \\ \pi_0\ (\mathrm{p}\ a\ b) = a \end{array}$$

$$\begin{array}{l} \pi_1 : (x : \Sigma\ A\ B) \to B\ (\pi_0\ x) \\ \pi_1\ (\mathrm{p}\ a\ b) = b \end{array}$$

Natural numbers

- **Formation rule:**

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

- **Introduction rules:**

$$\mathrm{zero} : \mathbb{N} \qquad \frac{n : \mathbb{N}}{\mathrm{S}\ n : \mathbb{N}}$$

- **Elimination rule:**

$$\frac{\begin{array}{c} C : \mathbb{N} \to \mathrm{Set} \\ step_{\mathrm{zero}} : C\ \mathrm{zero} \qquad step_{\mathrm{S}} : (n : \mathbb{N}, C\ n) \to C\ (\mathrm{S}\ n) \qquad n : \mathbb{N} \end{array}}{\mathrm{elim}_{\mathbb{N}}\ C\ step_{\mathrm{zero}}\ step_{\mathrm{S}}\ n : C\ n}$$

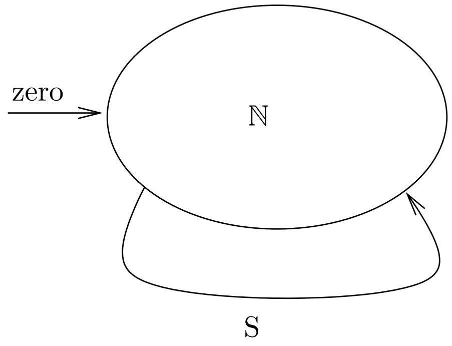
Natural numbers

- **Equality rules:**

$$\mathrm{elim}_{\mathbb{N}}\ C\ step_{\mathrm{zero}}\ step_{\mathrm{S}}\ \mathrm{zero} = step_{\mathrm{zero}} : C\ \mathrm{zero}$$

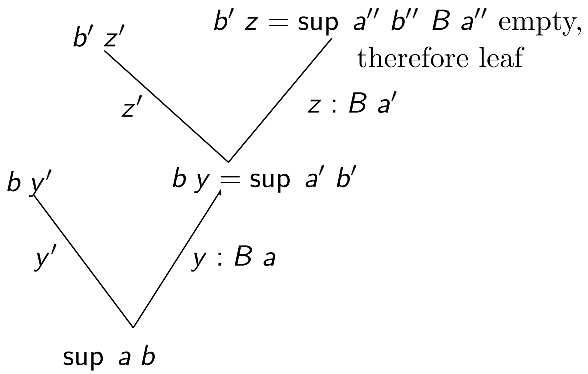
$$\begin{aligned}&\mathrm{elim}_{\mathbb{N}}\ C\ step_{\mathrm{zero}}\ step_{\mathrm{S}}\ (\mathrm{S}\ n)\\ &\quad = step_{\mathrm{S}}\ n\ (\mathrm{elim}_{\mathbb{N}}\ C\ step_{\mathrm{zero}}\ step_{\mathrm{S}}\ n) : C\ (\mathrm{S}\ n)\end{aligned}$$

Visualisation ($\mathbb{N}$)



- zero 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:**

$$\frac{A : \mathrm{Set} \qquad B : A \to \mathrm{Set}}{\mathrm{W}\ A\ B : \mathrm{Set}}$$

- **Introduction rule:**

$$\frac{a : A \qquad b : B\ a \to \mathrm{W}\ A\ B}{\mathrm{sup}\ a\ b : \mathrm{W}\ A\ B}$$

The W-Type

- **Elimination rule:**

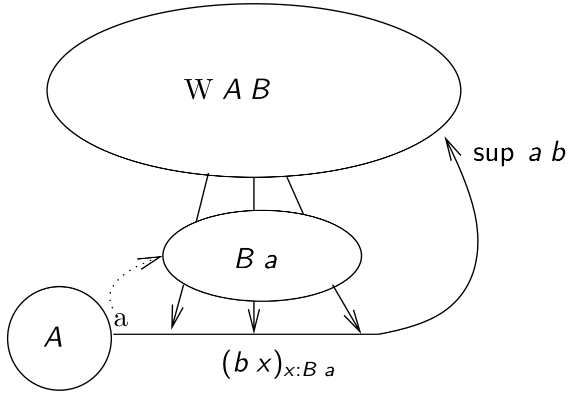
$$\frac{\begin{array}{l} C : \mathrm{W}\ A\ B \to \mathrm{Set} \\ \mathit{step} : (a : A) \\ \qquad \to (b : B\ a \to \mathrm{W}\ A\ B) \\ \qquad \to (\mathit{ih} : (x : B\ a) \to C\ (b\ x)) \\ \qquad \to C\ (\mathrm{sup}\ \ a\ b) \\ c : \mathrm{W}\ A\ B \end{array}}{\mathrm{elim}_{\mathrm{W}}\ C\ \mathit{step}\ c : C\ c}$$

- **Equality rule:**

$$\begin{array}{l} \mathrm{elim}_{\mathrm{W}}\ C\ \mathit{step}\ (\mathrm{sup}\ \ a\ b) \\ \qquad = \mathit{step}\ a\ b\ (\lambda x.\mathrm{elim}_{\mathrm{W}}\ C\ \mathit{step}\ (b\ x)) : C\ (\mathrm{sup}\ \ a\ b) \end{array}$$

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

Visualisation ($\mathrm{W}\ A\ B$)



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

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

Universes

- **Formation rules:**

$$\mathrm{U} : \mathrm{Set} \qquad \mathrm{T} : \mathrm{U} \to \mathrm{Set}$$

- **Introduction and Equality rules:**

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

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

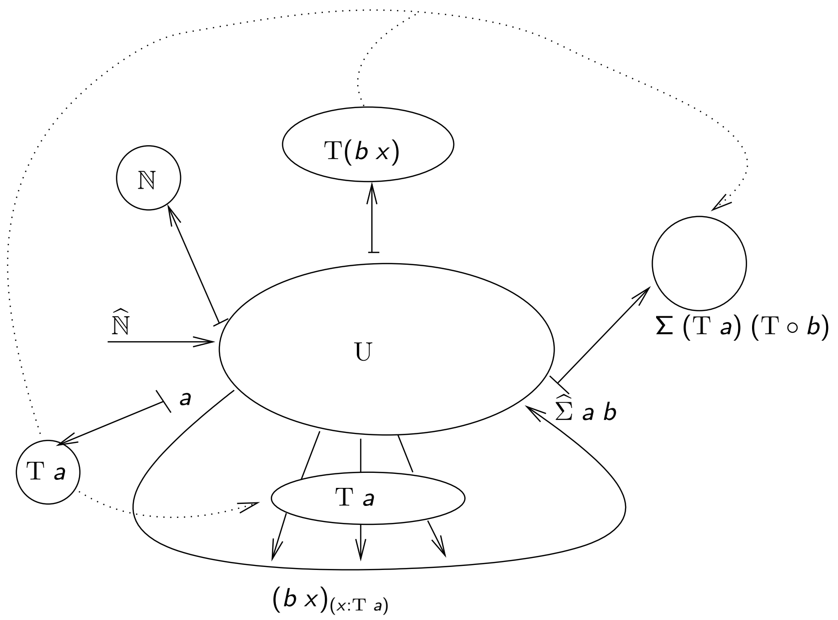
$$\mathrm{T}(\widehat{\Sigma}\ a\ b) = \Sigma\ (\mathrm{T}\ a)\ (\mathrm{T} \circ b)$$

Similarly for other type formers (except for U).

Elimination 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).

Visualisation (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 is indexed over $(\mathrm{T}\ a)$.
    - Index set $(\mathrm{T}\ a)$ for second argument depends on the 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. $\mathrm{Set} \to \mathrm{Set}$.

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:

$$\begin{aligned}\{f \mid\ & f \text{ function} \\ & \wedge \mathrm{dom}(f) = A \\ & \wedge \forall a \in A. f(a) \in B[x := a]\}\end{aligned}$$

Rules of the Dependent Funct. Set

Formation Rule

$$\frac{A : \mathrm{Set} \qquad x : A \Rightarrow B : \mathrm{Set}}{(x : A) \to B : \mathrm{Set}} (\to\text{-F})$$

Introduction Rule

$$\frac{x : A \Rightarrow b : B}{(\lambda x : A.b) : (x : A) \to B} (\to\text{-I})$$

Rules of the Dependent Function Set

Elimination Rule

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

Equality Rule

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

The η-Rule

The η-rule has a special status:

η-Rule

$$\frac{f : (x : A) \to B}{f = (\lambda x : A.f\ x) : (x : A) \to B} (\to\text{-}\eta)$$

- The η-rule expresses that every element of $(x : A) \to B$ is of the form $\lambda x : A.\mathrm{something}$.
- The η-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

$$(x : A) \to B$$

- Elements of $(x : A) \to B$ are introduced by using
  - either λ-abstraction, i.e. we can define

$$\begin{array}{lcl} f & : & (x : A) \to B \\ f & = & \lambda x \to b \end{array}$$

    - Requires that $b : B$ depending on $x : A$.
    - Note that the type B of b depends on $x : A$.
  - or by writing

$$\begin{array}{lcl} f & : & (x : A) \to B \\ f\ x & = & b \end{array}$$

The Dependent Function Set in Agda

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

Implication

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

Example

$$\begin{array}{l} \text{lemma} : A \to A \\ \text{lemma}\ a = a \end{array}$$

$$\begin{array}{l} \text{lemma2} : (A \to B) \to (B \to C) \to A \to C \\ \text{lemma2}\ f\ g\ a = g\ (f\ a) \end{array}$$

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** $(x : A) \to B$.
- Therefore we can **identify** $\forall \mathbf{x} : \mathbf{A}.\mathbf{B}$ with $(\mathbf{x} : \mathbf{A}) \to \mathbf{B}$.

$\forall$ in Agda

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

Example: Equality on $\mathbb{N}$

- We define equality on $\mathbb{N}$.
- First we introduce the true and false formulas:

- - $\perp$ is defined as $\emptyset$
data $\perp$: Set where

- - $\top$ has one proof, namely the trivial proof triv
data $\top$: Set where
  triv : $\top$

$$
\begin{array}{l}
\_ == \_ : \mathbb{N} \to \mathbb{N} \to \mathrm{Set} \\
\begin{array}{lllll}
\mathrm{zero} & == & \mathrm{zero} & = & \top \\
\mathrm{zero} & == & \mathrm{S}\ m & = & \perp \\
\mathrm{S}\ n & == & \mathrm{zero} & = & \perp \\
\mathrm{S}\ n & == & \mathrm{S}\ m & = & n == m
\end{array}
\end{array}
$$

Example Proof of Reflexivity of ==

$$
\begin{array}{l}
\mathrm{refl} : (n : \mathbb{N}) \to n == n \\
\begin{array}{lll}
\mathrm{refl\ zero} & = & \mathrm{triv} \\
\mathrm{refl}\ (\mathrm{S}\ n) & = & \mathrm{refl}\ n
\end{array}
\end{array}
$$

The Full Logical Framework

- Above we were already using types such as

$$C : \mathbb{B} \to \mathrm{Set}$$

- This doesn't type check yet, since we would need

$$\mathbb{B} \to \mathrm{Set} : \mathrm{Set}$$

and our rules allow this only if we had

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

Set

- Adding

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

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**.

Jean-Yves Girard

Set (Cont.)

- Instead we introduce a **new level on top of Set called Type.**
  - So besides judgements $A : \mathrm{Set}$ we have as well judgements of the form

$$A : \mathrm{Type}$$

  - One rule will especially express

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

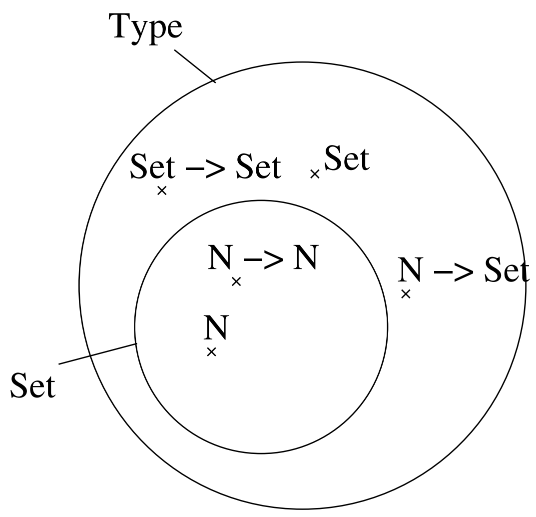
  - Elements of Type are **types**, elements of Set are **small types**.

Set (Cont.)

- We add rules asserting that **if A: Set then A: Type**.
- Further we add rules asserting that Type is closed under the elements of Set and the function type
- Since $\mathrm{Set} : \mathrm{Type}$ we get therefore (by closure under the function type)

$$\mathbb{B} \to \mathrm{Set} : \mathrm{Type}$$

Set and Type



Rules for Set (as an Element of Type)

Formation Rule for Set

$$\mathrm{Set} : \mathrm{Type} \qquad (\mathrm{SetIsType})$$

Every Set is a Type

$$\frac{A : \mathrm{Set}}{A : \mathrm{Type}}\ (\mathrm{Set2Type})$$

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

$$\frac{A : \mathrm{Type} \qquad x : A \Rightarrow B : \mathrm{Type}}{(x : A) \to B : \mathrm{Type}}\ (\to\text{-}\mathrm{F}^{\mathrm{Type}})$$

Algebraic Types

- 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.

$$
\begin{array}{l}
\mathrm{data\ A : Set\ where} \\
\quad \mathrm{C}_1 \ : (a_1 : \mathrm{A}^1_1) \to (a_2 : \mathrm{A}^1_2) \to \cdots \ (a_{n_1} : \mathrm{A}^1_{\mathrm{n}_1}) \to A \\
\quad \mathrm{C}_2 \ : (a_1 : \mathrm{A}^2_1) \to (a_2 : \mathrm{A}^2_2) \to \cdots (a_{n_2} : \mathrm{A}^2_{\mathrm{n}_2}) \ \to A \\
\quad \cdots \\
\quad \mathrm{C}_m : (a_1 : \mathrm{A}^\mathrm{m}_1) \to (a_2 : \mathrm{A}^\mathrm{m}_2) \to \cdots (a_{n_m} : \mathrm{A}^\mathrm{m}_{\mathrm{n}_\mathrm{m}}) \to A
\end{array}
$$

Meaning of “data”

- The idea is that A as before is the least set A s.t. we have constructors:

  $$
  \begin{array}{l}
  \mathrm{C_i} : (a_{i1} : \mathrm{A_{i1}}) \\
  \quad \to \cdots \\
  \quad \to (a_{in_i} : \mathrm{A_{in_i}}) \\
  \quad \to \mathrm{A}
  \end{array}
  $$

  where a constructor always constructs new elements.
- In other words the elements of A are exactly those constructed by those constructors.

Strictly Positive Algebraic Types

- In the types $\mathrm{A_{ij}}$ we can make use of A.
  - However, it is difficult to understand A, if we have **negative** occurrences of A.
  - Example:

    $$
    \begin{array}{l}
    \mathrm{data\ A : Set\ where} \\
    \quad \mathrm{C : (A \to A) \to A}
    \end{array}
    $$
  - What is the least set A having a constructor

    $$\mathrm{C : (A \to A) \to A} \qquad ?$$

Strictly Positive Algebraic Types

- If we
  - have constructed some elements of A already,
  - find a function $f : \mathrm{A} \to \mathrm{A}$, and
  - add $\mathrm{C\ f}$ to A,

  then f might no longer be a function $\mathrm{A} \to \mathrm{A}$.
  (f applied to the new element $\mathrm{C}\ f$ might not be defined).
- In fact, the termination checker issues a warning, if we define A as above.
- We shouldn't make use of such definitions.

Strictly Positive Algebraic Types

- A "good" definition is the set of lists of natural numbers, defined as follows:

$$\begin{array}{l}\mathrm{data}\ \mathbb{N}\mathrm{List} : \mathrm{Set}\ \mathrm{where} \\ \quad \begin{array}{lcl} [\,] & : & \mathbb{N}\mathrm{List} \\ \_::\_ & : & \mathbb{N} \to \mathbb{N}\mathrm{List} \to \mathbb{N}\mathrm{List}\end{array}\end{array}$$

- The constructor $\_::\_$ of $\mathbb{N}\mathrm{List}$ refers to $\mathbb{N}\mathrm{List}$, but in a positive way: We have: if $a : \mathbb{N}$ and $l : \mathbb{N}\mathrm{List}$, then

$$(a :: l) : \mathbb{N}\mathrm{List} \ .$$

Strictly Positive Algebraic Types

- If we add $a :: l$ to $\mathbb{N}\mathrm{List}$, the reason for adding it (namely $l : \mathbb{N}\mathrm{List}$) is not destroyed by this addition.
- So we can "construct" the set $\mathbb{N}\mathrm{List}$ by
  - starting with the empty set,
  - adding [] and
  - closing it under $\_::\_$ whenever possible.
- Because we can "construct" $\mathbb{N}\mathrm{List}$, the above is an acceptable definition.

Strictly Positive Algebraic Types

- In general:

$$\begin{array}{l}\mathrm{data}\ \mathrm{A} : \mathrm{Set}\ \mathrm{where} \\ \quad \mathrm{C}_1 : (a_1 : \mathrm{A}^1_1) \to (a_2 : \mathrm{A}^1_2) \to \cdots\ (a_{n_1} : \mathrm{A}^1_{\mathrm{n}_1}) \to A \\ \quad \mathrm{C}_2 : (a_1 : \mathrm{A}^2_1) \to (a_2 : \mathrm{A}^2_2) \to \cdots (a_{n_2} : \mathrm{A}^2_{\mathrm{n}_2}) \to A \\ \quad \cdots \\ \quad \mathrm{C}_m : (a_1 : \mathrm{A}^\mathrm{m}_1) \to (a_2 : \mathrm{A}^\mathrm{m}_2) \to \cdots (a_{n_m} : \mathrm{A}^\mathrm{m}_{\mathrm{n}_\mathrm{m}}) \to A\end{array}$$

  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.

Strictly Positive Algebraic Types

- The definitions of finite sets, $\Sigma\ A\ B$, $A + B$ and $\mathbb{N}$ were strictly positive algebraic types.

One further Example

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

$$\begin{array}{lll} \multicolumn{3}{l}{\mathrm{data\ BinTree : Set\ where}} \\ \quad \mathrm{leaf} & : & \mathrm{BinTree} \\ \quad \mathrm{branch} & : & \mathrm{Bintree} \to \mathrm{Bintree} \to \mathrm{Bintree} \end{array}$$

- 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:

$$\begin{array}{l} \mathrm{mutual} \\ \quad \begin{array}{lll} \multicolumn{3}{l}{\mathrm{data\ Even : Set\ where}} \\ \quad \mathrm{Z} & : & \mathrm{Even} \\ \quad \mathrm{S} & : & \mathrm{Odd} \to \mathrm{Even} \\ \\ \multicolumn{3}{l}{\mathrm{data\ Odd : Set\ where}} \\ \quad \mathrm{S'} & : & \mathrm{Even} \to \mathrm{Odd} \end{array} \end{array}$$

- 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 $\mathrm{A}_{ij} = B_1 \to A$ or even $\mathrm{A}_{ij} = B_1 \to \cdots \to B_l \to A$, where A is one of the types introduced simultaneously.
- Example (called "Kleene's O"):

$$\begin{array}{lll} \multicolumn{3}{l}{\mathrm{data\ O : Set\ where}} \\ \quad \mathrm{leaf} & : & \mathrm{O} \\ \quad \mathrm{succ} & : & \mathrm{O} \to \mathrm{O} \\ \quad \mathrm{lim} & : & (\mathbb{N} \to \mathrm{O}) \to \mathrm{O} \end{array}$$

- The last definition is unproblematic, since, if we have $f : \mathbb{N} \to \mathrm{O}$ and construct $\mathrm{lim}\ f$ 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".

Elimination Rules for data

- 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 $f\ (\mathrm{C}\ a_1\ \cdots\ a_n)$, we can refer only to f applied to elements used in $\mathrm{C}\ a_1\ \cdots\ a_n$.

Examples

- For instance
  - in the Bintree example, when defining

$$\mathrm{f} : \mathrm{Bintree} \to A$$

by case-distinction, then the definition of

$$\mathrm{f}\ (\mathrm{branch}\ l\ r)$$

can make use of $\mathrm{f}\ l$ and $\mathrm{f}\ r$.

Examples

- In the example of O, when defining

$$\mathrm{g} : \mathrm{O} \to A$$

by case-distinction, then the definition of

$$\mathrm{g}\ (\lim f)$$

can make use of $\mathrm{g}\ (f\ n)$ for all $n : \mathbb{N}$.

Codata Type

- Idea of Codata Types non-well-founded versions of inductive data types:

$$\begin{array}{l} \text{codata Stream} : \text{Set where} \\ \quad \text{cons} \;\; : \;\; \mathbb{N} \to \text{Stream} \to \text{Stream} \end{array}$$

- 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.

Coinductive Data Types (Coalgebras) in Type Theory

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.

Coinductive Data Types (Coalgebras) in Type Theory

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.

Coinductive Data Types (Coalgebras) in Type Theory

Algebras and Coalgebras

- Algebraic data types correspond to initial algebras.
  - $\mathbb{N}$ as an algebra can be represented as introduction rules for $\mathbb{N}$:

$$\begin{array}{lcl} \text{zero} & : & \mathbb{N} \\ \text{S} & : & \mathbb{N} \to \mathbb{N} \end{array}$$

- Coalgebra obtained by "reversing the arrows".
  - Stream as a coalgebra can be expressed as as elimination rules for it:

$$\begin{array}{lcl} \text{head} & : & \text{Stream} \to \mathbb{N} \\ \text{tail} & : & \text{Stream} \to \text{Stream} \end{array}$$

Weakly Initial Algebras and Final Coalgebras

- $\mathbb{N}$ as a weakly initial algebra corresponds to iteration (elimination rule): For $A : \mathrm{Set}$, $a : A$, $f : A \to A$ there exists

$$\begin{array}{lcl} g : \mathbb{N} \to A & & \\ g\ \mathrm{zero} & = & a \\ g\ (\mathrm{S}\ n) & = & f\ (g\ n) \end{array}$$

(or $g\ n = f^n\ a$).

- Stream as a weakly final coalgebra corresponds to coiteration or guarded iteration (introduction rule):
For $A : \mathrm{Set}$, $f_0 : A \to \mathbb{N}$, $f_1 : A \to A$ there exists g s.t.

$$\begin{array}{lcl} g : A \to \mathrm{Stream} & & \\ \mathrm{head}\ (g\ a) & = & f_0\ a \\ \mathrm{tail}\ (g\ a) & = & g\ (f_1\ a) \end{array}$$

Example

- Using coiteration we can define

$$\begin{array}{lcl} \mathrm{inc} : \mathbb{N} \to \mathrm{Stream} & & \\ \mathrm{head}\ (\mathrm{inc}\ n) & = & n \\ \mathrm{tail}\ \ (\mathrm{inc}\ n) & = & \mathrm{inc}\ (n+1) \end{array}$$

Recursion and Corecursion

- $\mathbb{N}$ as an initial algebra corresponds to uniqueness of g above.
  - Allows to derive primitive recursion:
  For $A : \mathrm{Set}$, $a : A$, $f : (\mathbb{N} \times A) \to A$ there exists

$$\begin{array}{lcl} g : \mathbb{N} \to A & & \\ g\ \mathrm{zero} & = & a \\ g\ (\mathrm{S}\ n) & = & f\ \langle n, (g\ n)\rangle \end{array}$$

- Stream as a final coalgebra corresponds to uniqueness of h.
  - Allows to derive primitive corecursion:
  For $A : \mathrm{Set}$, $f_0 : A \to \mathbb{N}$, $f_1 : A \to (\mathrm{Stream} + A)$ there exists

$$\begin{array}{lcll} g : A \to \mathrm{Stream} & & & \\ \mathrm{head}\ (g\ a) & = & f_0\ a & \\ \mathrm{tail}\ (g\ a) & = & s & \text{if } f_1\ a = \mathrm{inl}\ s \\ \mathrm{tail}\ (g\ a) & = & g\ a' & \text{if } f_1\ a = \mathrm{inr}\ a' \end{array}$$

Recursion vs Iteration

- Using recursion we can define inverse case of the constructors of $\mathbb{N}$ as follows:

$$\begin{array}{lcl} \mathrm{case} : \mathbb{N} \to (1 + \mathbb{N}) & & \\ \mathrm{case}\ \mathrm{zero} & = & \mathrm{inl} \\ \mathrm{case}\ (\mathrm{S}\ n) & = & \mathrm{inr}\ n \end{array}$$

- Using iteration, we cannot make use of n and therefore case is defined inefficiently:

$$\begin{array}{lcl} \mathrm{case} : \mathbb{N} \to (1 + \mathbb{N}) & & \\ \mathrm{case}\ \mathrm{zero} & = & \mathrm{inl} \\ \mathrm{case}\ (\mathrm{S}\ n) & = & \mathrm{caseaux}\ (\mathrm{case}\ n) \end{array}$$

$$\begin{array}{lcl} \mathrm{caseaux} : (1 + \mathbb{N}) \to (1 + \mathbb{N}) & & \\ \mathrm{caseaux}\ \mathrm{inl} & = & \mathrm{inr}\ \mathrm{zero} \\ \mathrm{caseaux}\ (\mathrm{inr}\ n) & = & \mathrm{inr}\ (\mathrm{S}\ n) \end{array}$$

Definition of pred

- One way of defining pred by iteration is by defining first case and then to define

$$\begin{array}{lcl} \text{predaux} : (1 + \mathbb{N}) \to \mathbb{N} & & \\ \text{predaux inl} & = & \text{zero} \\ \text{predaux } (\text{inr } n) & = & n \end{array}$$

$$\begin{array}{l} \text{pred} : \mathbb{N} \to \mathbb{N} \\ \text{pred } n = \text{predaux } (\text{case } n) \end{array}$$

Coinductive Data Types (Coalgebras) in Type Theory

Corecursion vs Coiteration

- Definition of cons (inverse of the destructors) using coiteration inefficient:

$$\begin{array}{lcl} \text{cons} : \mathbb{N} \to \text{Stream} \to \text{Stream} & & \\ \text{head } (\text{cons } n\ s) & = & n \\ \text{tail } (\text{cons } n\ s) & = & \text{cons } (\text{head } s)\ (\text{tail } s) \end{array}$$

  - Using primitive corecursion we can define more easily

$$\begin{array}{lcl} \text{cons} : \mathbb{N} \to \text{Stream} \to \text{Stream} & & \\ \text{head } (\text{cons } n\ s) & = & n \\ \text{tail } (\text{cons } n\ s) & = & s \end{array}$$

Coinductive Data Types (Coalgebras) in Type Theory

Induction - Coinduction?

- Induction is dependent primitive recursion:
  For $A : \mathbb{N} \to \text{Set}$, $a : A\ \text{zero}$, $f : (n : \mathbb{N}) \to A\ n \to A\ (\text{S}\ n)$ there exists

$$\begin{array}{lcl} g : (n : \mathbb{N}) \to A\ n & & \\ g\ \text{zero} & = & a \\ g\ (\text{S}\ n) & = & f\ n\ (g\ n) \end{array}$$

  - 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 **dependent** primitive recursion?

Coinductive Data Types (Coalgebras) in Type Theory

Weakly Final Coalgebra

- Equality for final coalgebras is undecidable:
  Two streams

$$\begin{array}{lclccccccc} s & = & (a_0 & , & a_1 & , & a_2 & , & \ldots \\ t & = & (b_0 & , & b_1 & , & b_2 & , & \ldots \end{array}$$

  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.

Definition of Coalgebras by Observations

- We see now that elements of coalgebras are defined by their observations:
  An element s of Stream is anything for which we can define

$$\begin{array}{lll} \mathrm{head} & s & : & \mathbb{N} \\ \mathrm{tail} & s & : & \mathrm{Stream} \end{array}$$

- This generalises the function type.
  Functions are as well determined by observations.
  - An $f : A \to B$ is any program which if applied to $a : A$ returns some $b : B$.
- **Inductive data types** are defined by **construction**
  coalgebraic data types and **functions** by **observations**.

Relationship to Objects in Object-Oriented Programming

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

Patterns and Copatterns

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

Patterns and Copatterns

$f\ n = n, n, n-1, n-1, \ldots 0, 0, N, N, N-1, N-1, \ldots 0, 0, N, N, N-1, N-1,$

$$\begin{array}{lll} f : \mathbb{N} \to \mathrm{Stream} & & \\ f & = & ? \end{array}$$

$$\begin{array}{lll} f : \mathbb{N} \to \mathrm{Stream} & & \\ f & = & ? \end{array}$$

Copattern matching on $f : \mathbb{N} \to \mathrm{Stream}$:

$$\begin{array}{lll} f : \mathbb{N} \to \mathrm{Stream} & & \\ f\ n & = & ? \end{array}$$

$$\begin{array}{lll} f : \mathbb{N} \to \mathrm{Stream} & & \\ f\ n & = & ? \end{array}$$

Copattern matching on $f\ n : \mathrm{Stream}$:

$$\begin{array}{lll} f : \mathbb{N} \to \mathrm{Stream} & & \\ \mathrm{head}\ (f\ n) & = & ? \\ \mathrm{tail}\ (f\ n) & = & ? \end{array}$$