Introduction to Type Theory in Agda

Lecture 1 – Introduction

Todd Waugh Ambridge

8 April 2024

Lecture Outline

Lecture Outline

Motivation

The first question we shall try to answer in this course is

What are types?

Types can be understood from two perspectives:

- Their use in foundational theories of mathematics,
- Their use in typed programming languages.

The main goal of this course is to show you that when we work with Type Theory in Agda, these two perspectives become one and the same.

Motivation

The first question we shall try to answer in this course is

What are types?

Types can be understood from two perspectives:

- Their use in foundational theories of mathematics,
- Their use in typed programming languages.

The main goal of this course is to show you that when we work with Type Theory in Agda, these two perspectives become one and the same.

Motivation

The first question we shall try to answer in this course is

What are types?

Types can be understood from two perspectives:

- Their use in foundational theories of mathematics,
- Their use in typed programming languages.

The main goal of this course is to show you that when we work with Type Theory in Agda, these two perspectives become one and the same.

Motivation

The first question we shall try to answer in this course is

What are types?

Types can be understood from two perspectives:

- **Their use in foundational theories of mathematics,**
- Their use in typed programming languages.

The main goal of this course is to show you that when we work with Type Theory in Agda, these two perspectives become one and the same.

Motivation – Types in Mathematics

In traditional *set-theoretic* foundations of mathematics, objects (such as numbers, the Booleans, shapes, etc.) are collected into *sets*.

For example, the number 5 is an element of

- the set of natural numbers

$$\mathbb{N} := \{0, 1, 2, 3, 4, 5, ...\},$$

- the set of odd numbers

$$\mathbb{N}_o := \{n \in \mathbb{N} \mid \exists a. 2a + 1 = n\},$$

- the set of prime numbers

$$\mathbb{N}_p := \{n \in \mathbb{N} \mid n \text{ is a prime number}\},$$

- etc.

Motivation – Types in Mathematics

In traditional *set-theoretic* foundations of mathematics, objects (such as numbers, the Booleans, shapes, etc.) are collected into *sets*.

For example, the number 5 is an element of

- the set of natural numbers

$$\mathbb{N} := \{0, 1, 2, 3, 4, 5, \ldots\},$$

- the set of odd numbers

$$\mathbb{N}_o := \{n \in \mathbb{N} \mid \exists a.2a + 1 = n\},$$

- the set of prime numbers

$$\mathbb{N}_p := \{n \in \mathbb{N} \mid n \text{ is a prime number}\},$$

- etc.

Motivation – Types in Mathematics

In traditional *set-theoretic* foundations of mathematics, objects (such as numbers, the Booleans, shapes, etc.) are collected into *sets*.

For example, the number 5 is an element of

- the set of natural numbers

$$\mathbb{N} := \{0, 1, 2, 3, 4, 5, ...\},$$

- the set of odd numbers

$$\mathbb{N}_o := \{n \in \mathbb{N} \mid \exists a. 2a + 1 = n\},$$

- the set of prime numbers

$$\mathbb{N}_p := \{n \in \mathbb{N} \mid n \text{ is a prime number}\},$$

- etc.

Motivation – Types in Mathematics

In traditional *set-theoretic* foundations of mathematics, objects (such as numbers, the Booleans, shapes, etc.) are collected into *sets*.

For example, the number 5 is an element of

- the set of natural numbers

$$\mathbb{N} := \{0, 1, 2, 3, 4, 5, ...\},$$

- the set of odd numbers

$$\mathbb{N}_o := \{n \in \mathbb{N} \mid \exists a. 2a + 1 = n\},$$

- the set of prime numbers

$$\mathbb{N}_p := \{n \in \mathbb{N} \mid n \text{ is a prime number}\},$$

- etc.

Motivation – Types in Mathematics

This differs to the type-theoretic perspective, where objects instead arise from *rules of construction*. These rules are unique and, hence, objects have a *unique* type.

For example, the number 5 is a term of

- the type of natural numbers $\mathbb{N}$;

5 *paired with a proof* that $\exists a.2a + 1 = 5$ is a term of

- the type of odd numbers $\mathbb{N}_o$;

5 *paired with a proof* that it is a prime number is a term of

- the type of prime numbers $\mathbb{N}_p$,

Note that, in type theory, *logical statements* and *proofs* themselves are objects on the same level as (for example) numbers.

Motivation – Types in Mathematics

This differs to the type-theoretic perspective, where objects instead arise from *rules of construction*. These rules are unique and, hence, objects have a *unique* type.

For example, the number 5 is a term of

- the type of natural numbers $\mathbb{N}$;

5 *paired with a proof* that $\exists a.2a + 1 = 5$ is a term of

- the type of odd numbers $\mathbb{N}_o$;

5 *paired with a proof* that it is a prime number is a term of

- the type of prime numbers $\mathbb{N}_p$,

Note that, in type theory, *logical statements* and *proofs* themselves are objects on the same level as (for example) numbers.

Motivation – Types in Mathematics

This differs to the type-theoretic perspective, where objects instead arise from *rules of construction*. These rules are unique and, hence, objects have a *unique* type.

For example, the number 5 is a term of

- the type of natural numbers $\mathbb{N}$;

5 *paired with a proof* that $\exists a.2a + 1 = 5$ is a term of

- the type of odd numbers $\mathbb{N}_o$;

5 *paired with a proof* that it is a prime number is a term of

- the type of prime numbers $\mathbb{N}_p$,

Note that, in type theory, *logical statements* and *proofs* themselves are objects on the same level as (for example) numbers.

Motivation – Types in Mathematics

This differs to the type-theoretic perspective, where objects instead arise from *rules of construction*. These rules are unique and, hence, objects have a *unique* type.

For example, the number 5 is a term of

- the type of natural numbers $\mathbb{N}$;

5 *paired with a proof* that $\exists a.2a + 1 = 5$ is a term of

- the type of odd numbers $\mathbb{N}_o$;

5 *paired with a proof* that it is a prime number is a term of

- the type of prime numbers $\mathbb{N}_p$,

Note that, in type theory, *logical statements* and *proofs* themselves are objects on the same level as (for example) numbers.

Motivation – Types in Mathematics

This differs to the type-theoretic perspective, where objects instead arise from *rules of construction*. These rules are unique and, hence, objects have a *unique* type.

For example, the number 5 is a term of

- the type of natural numbers $\mathbb{N}$;

5 *paired with a proof* that $\exists a.2a + 1 = 5$ is a term of

- the type of odd numbers $\mathbb{N}_o$;

5 *paired with a proof* that it is a prime number is a term of

- the type of prime numbers $\mathbb{N}_p$,

Note that, in type theory, *logical statements* and *proofs* themselves are objects on the same level as (for example) numbers.

Motivation

The first question we shall try to answer in this course is

What are types?

Types can be understood from two perspectives:

- Their use in foundational theories of mathematics,
- **Their use in typed programming languages.**

The main goal of this course is to show you that when we work with Type Theory in Agda, these two perspectives become one and the same.

Motivation – Types in Programming

You may have already come across types in programming, because many programming languages are *typed*. For example, in Java:

- The terms `true` and `false` have type `Boolean`,
- The terms 5 and 3 has type `int`,
- The expression `5 + 3` has type `int`,
- The operator + *(effectively)* has type `(int , int) -> int`.

How do types help us as programmers?

Motivation – Types in Programming

You may have already come across types in programming, because many programming languages are *typed*. For example, in Java:

- The terms `true` and `false` have type `Boolean`,
- The terms 5 and 3 has type `int`,
- The expression `5 + 3` has type `int`,
- The operator + *(effectively)* has type `(int , int) -> int`.

How do types help us as programmers?

Motivation – Types in Programming

You may have already come across types in programming, because many programming languages are *typed*. For example, in Java:

- The terms `true` and `false` have type `Boolean`,
- The terms 5 and 3 has type `int`,
- The expression `5 + 3` has type `int`,
- The operator + *(effectively)* has type `(int , int) -> int`.

How do types help us as programmers?

Motivation – Types in Programming

You may have already come across types in programming, because many programming languages are *typed*. For example, in Java:

- The terms `true` and `false` have type `Boolean`,
- The terms `5` and `3` has type `int`,
- The expression `5 + 3` has type `int`,
- The operator `+` *(effectively)* has type `(int , int) -> int`.

How do types help us as programmers?

Motivation – Types in Programming

You may have already come across types in programming, because many programming languages are *typed*. For example, in Java:

- The terms `true` and `false` have type `Boolean`,
- The terms `5` and `3` has type `int`,
- The expression `5 + 3` has type `int`,
- The operator `+` *(effectively)* has type `(int , int) -> int`.

How do types help us as programmers?

Motivation – Types in Programming

You may have already come across types in programming, because many programming languages are *typed*. For example, in Java:

- The terms `true` and `false` have type `Boolean`,
- The terms `5` and `3` has type `int`,
- The expression `5 + 3` has type `int`,
- The operator `+` *(effectively)* has type `(int , int) -> int`.

How do types help us as programmers?

Motivation – Types in Programming

Types stop us from writing nonsense in our programs!

For example, the expression `true + 5`, which adds a `Boolean` and an `int`, doesn't make any sense. In an untyped language, at runtime, trying to evaluate this expression will cause a *run-time error* or unpredictable behaviour.

But Java's type system prevents us writing this in the first place.

- The + operator *(effectively)* has type `(int , int) -> int`.
- If we pretend it has type `(Boolean , int) -> int`, then we get an expresison (e.g. `true + 5`) that is not *well-typed*...
- This results in a *compile-time* error, which is much more helpful to a programmer than a runtime error.

Motivation – Types in Programming

Types stop us from writing nonsense in our programs!

For example, the expression `true + 5`, which adds a `Boolean` and an `int`, doesn't make any sense. In an untyped language, at runtime, trying to evaluate this expression will cause a *run-time error* or unpredictable behaviour.

But Java's type system prevents us writing this in the first place.

- The + operator *(effectively)* has type `(int , int) -> int`.
- If we pretend it has type `(Boolean , int) -> int`, then we get an expresison (e.g. `true + 5`) that is not *well-typed*...
- This results in a *compile-time* error, which is much more helpful to a programmer than a runtime error.

Motivation – Types in Programming

Types stop us from writing nonsense in our programs!

For example, the expression `true + 5`, which adds a `Boolean` and an `int`, doesn't make any sense. In an untyped language, at runtime, trying to evaluate this expression will cause a *run-time error* or unpredictable behaviour.

But Java's type system prevents us writing this in the first place.

- The + operator *(effectively)* has type `(int , int) -> int`.
- If we pretend it has type `(Boolean , int) -> int`, then we get an expresison (e.g. `true + 5`) that is not *well-typed*...
- This results in a *compile-time* error, which is much more helpful to a programmer than a runtime error.

Motivation – Types in Programming

Types stop us from writing nonsense in our programs!

For example, the expression `true + 5`, which adds a `Boolean` and an `int`, doesn't make any sense. In an untyped language, at runtime, trying to evaluate this expression will cause a *run-time error* or unpredictable behaviour.

But Java's type system prevents us writing this in the first place.

- The + operator *(effectively)* has type `(int , int) -> int`.
- If we pretend it has type `(Boolean , int) -> int`, then we get an expresison (e.g. `true + 5`) that is not *well-typed*...
- This results in a *compile-time* error, which is much more helpful to a programmer than a runtime error.

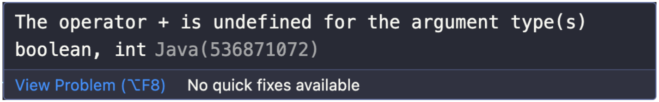
Motivation – Types in Programming

Types stop us from writing nonsense in our programs!

For example, the expression `true + 5`, which adds a `Boolean` and an `int`, doesn't make any sense. In an untyped language, at runtime, trying to evaluate this expression will cause a *run-time error* or unpredictable behaviour.

But Java's type system prevents us writing this in the first place.

- The + operator *(effectively)* has type `(int , int) -> int`.
- If we pretend it has type `(Boolean , int) -> int`, then we get an expresison (e.g. `true + 5`) that is not *well-typed*...



```
The operator + is undefined for the argument type(s)
boolean, int Java(536871072)
```

- This results in a *compile-time* error, which is much more helpful to a programmer than a runtime error.

Motivation

So, what are types?

- Types are a way of organising mathematical objects and terms of a programming language by *rules of construction*.
- The type of an object or term is *unique*.
- The type of an object is used to determine its *behaviour*, such as which operations can manipulate it.
- Types are even used to construct *logical statements and proofs*.

While these two quick perspectives can help us to understand types, type theory is a very deep and broad field of mathematics and theoretical computer science.

One that stretches back to the early 1900s...

Motivation

So, what are types?

- Types are a way of organising mathematical objects and terms of a programming language by *rules of construction*.
- The type of an object or term is *unique*.
- The type of an object is used to determine its *behaviour*, such as which operations can manipulate it.
- Types are even used to construct *logical statements and proofs*.

While these two quick perspectives can help us to understand types, type theory is a very deep and broad field of mathematics and theoretical computer science.

One that stretches back to the early 1900s...

Motivation

So, what are types?

- Types are a way of organising mathematical objects and terms of a programming language by *rules of construction*.
- The type of an object or term is *unique*.
- The type of an object is used to determine its *behaviour*, such as which operations can manipulate it.
- Types are even used to construct *logical statements and proofs*.

While these two quick perspectives can help us to understand types, type theory is a very deep and broad field of mathematics and theoretical computer science.

One that stretches back to the early 1900s...

Motivation

So, what are types?

- Types are a way of organising mathematical objects and terms of a programming language by *rules of construction*.
- The type of an object or term is *unique*.
- The type of an object is used to determine its *behaviour*, such as which operations can manipulate it.
- Types are even used to construct *logical statements and proofs*.

While these two quick perspectives can help us to understand types, type theory is a very deep and broad field of mathematics and theoretical computer science.

One that stretches back to the early 1900s...

Motivation

So, what are types?

- Types are a way of organising mathematical objects and terms of a programming language by *rules of construction*.
- The type of an object or term is *unique*.
- The type of an object is used to determine its *behaviour*, such as which operations can manipulate it.
- Types are even used to construct *logical statements and proofs*.

While these two quick perspectives can help us to understand types, type theory is a very deep and broad field of mathematics and theoretical computer science.

One that stretches back to the early 1900s...

Motivation

So, what are types?

- Types are a way of organising mathematical objects and terms of a programming language by *rules of construction*.
- The type of an object or term is *unique*.
- The type of an object is used to determine its *behaviour*, such as which operations can manipulate it.
- Types are even used to construct *logical statements and proofs*.

While these two quick perspectives can help us to understand types, type theory is a very deep and broad field of mathematics and theoretical computer science.

One that stretches back to the early 1900s...

Lecture Outline

1. Motivation
2. **A Brief History of Type Theory**
3. Course Outline
4. Formal Notation of Type Theory
5. Church's simply typed λ-calculus
6. Martin-Lof Constructive Type Theory (MLTT)
7. Introducing Agda!

A Brief History of Type Theory

1908: Russell introduces type theory as a foundation of mathematics that avoids the inconsistencies of naive set theory.

1934: Curry discovers that types can be viewed as (constructive) logical axioms.

1936: Church develops the *λ-calculus*, a foundational 'programming language' used to model functions and application. Infinite loops can be defined, which are inconsistent when the calculus is viewed as logic.

1940: Type theory comes to the rescue again! Church introduces the *simply typed λ-calculus*, whose typing rules prevent nonterminating terms from being derived.

1969: Howard, following Curry's perspective, discovers the simply type λ-calculus corresponds exactly to (constructive) natural deduction.

And thus began the quest for new type theories...

A Brief History of Type Theory

1908: Russell introduces type theory as a foundation of mathematics that avoids the inconsistencies of naive set theory.

1934: Curry discovers that types can be viewed as (constructive) logical axioms.

1936: Church develops the *λ-calculus*, a foundational ‘programming language’ used to model functions and application. Infinite loops can be defined, which are inconsistent when the calculus is viewed as logic.

1940: Type theory comes to the rescue again! Church introduces the *simply typed λ-calculus*, whose typing rules prevent nonterminating terms from being derived.

1969: Howard, following Curry’s perspective, discovers the simply type λ-calculus corresponds exactly to (constructive) natural deduction.

And thus began the quest for new type theories...

A Brief History of Type Theory

1908: Russell introduces type theory as a foundation of mathematics that avoids the inconsistencies of naive set theory.

1934: Curry discovers that types can be viewed as (constructive) logical axioms.

1936: Church develops the *λ-calculus*, a foundational 'programming language' used to model functions and application. Infinite loops can be defined, which are inconsistent when the calculus is viewed as logic.

1940: Type theory comes to the rescue again! Church introduces the *simply typed λ-calculus*, whose typing rules prevent nonterminating terms from being derived.

1969: Howard, following Curry's perspective, discovers the simply type λ-calculus corresponds exactly to (constructive) natural deduction.

And thus began the quest for new type theories...

A Brief History of Type Theory

1908: Russell introduces type theory as a foundation of mathematics that avoids the inconsistencies of naive set theory.

1934: Curry discovers that types can be viewed as (constructive) logical axioms.

1936: Church develops the *λ-calculus*, a foundational 'programming language' used to model functions and application. Infinite loops can be defined, which are inconsistent when the calculus is viewed as logic.

1940: Type theory comes to the rescue again! Church introduces the *simply typed λ-calculus*, whose typing rules prevent nonterminating terms from being derived.

1969: Howard, following Curry's perspective, discovers the simply type λ-calculus corresponds exactly to (constructive) natural deduction.

And thus began the quest for new type theories...

A Brief History of Type Theory

Following this Curry-Howard correspondence, also known as the *propositions-as-types interpretation* because types are viewed as mathematical propositions, Per Martin-Lof introduced his *constructive type theory* in 1972.

By utilising *dependent types* — types that are dependent on terms of other types — Martin-Lof Type Theory (MLTT) corresponds to to (constructive) predicate logic, and is thus a very rich foundation of mathematics. It has been used to formalise a wide variety of proofs in constructive mathematics, and is the basis of the more recent *univalent type theory* (2012).

From now on in this course, when we say "Type Theory" we mean dependent type theories and systems based on MLTT, such as Agda's type system.

A Brief History of Type Theory

Following this Curry-Howard correspondence, also known as the *propositions-as-types interpretation* because types are viewed as mathematical propositions, Per Martin-Lof introduced his *constructive type theory* in 1972.

By utilising *dependent types* — types that are dependent on terms of other types — Martin-Lof Type Theory (MLTT) corresponds to to (constructive) predicate logic, and is thus a very rich foundation of mathematics. It has been used to formalise a wide variety of proofs in constructive mathematics, and is the basis of the more recent *univalent type theory* (2012).

From now on in this course, when we say "Type Theory" we mean dependent type theories and systems based on MLTT, such as Agda's type system.

A Brief History of Type Theory

Following this Curry-Howard correspondence, also known as the *propositions-as-types interpretation* because types are viewed as mathematical propositions, Per Martin-Lof introduced his *constructive type theory* in 1972.

By utilising *dependent types* — types that are dependent on terms of other types — Martin-Lof Type Theory (MLTT) corresponds to to (constructive) predicate logic, and is thus a very rich foundation of mathematics. It has been used to formalise a wide variety of proofs in constructive mathematics, and is the basis of the more recent *univalent type theory* (2012).

From now on in this course, when we say “Type Theory” we mean dependent type theories and systems based on MLTT, such as Agda's type system.

Lecture Outline

Course Outline

In this course, we will:

- Learn about type theory — at first formally, but then by using Agda.
- Gain experience programming in a dependently-typed programming language.
- Formalise mathematical proofs, and see how type theory lends itself to this task.

In the lectures, we will build up our framework for MLTT in Agda. In the exercise classes, we will use this framework to define various mathematical structures and prove things about them!

You **do not** need to install Agda locally! We will use Ingo Blechschmidt's Agdapad.

Course Outline

Lecture 1: Introduction

Lecture 2: Propositions as types

Lecture 3: Equality and equivalence

Lecture 4: Towards univalent type theory

Course Outline

Lecture Outline

Formal Notation of Type Theory

Formally, a type theory is made up of *typing rules*.
Typing rules are a deductive system of *typing judgements*.
And typing judgements allow us to reason about *type assignments*.

Before we can introduce MLTT, we first need to understand these three fundamental concepts that are common across different type theories.

Formal Notation of Type Theory

Formally, a type theory is made up of *typing rules*.
Typing rules are a deductive system of *typing judgements*.
And typing judgements allow us to reason about *type assignments*.

Before we can introduce MLTT, we first need to understand these three fundamental concepts that are common across different type theories.

Formal Notation of Type Theory

Formally, a type theory is made up of *typing rules*.
Typing rules are a deductive system of *typing judgements*.
And typing judgements allow us to reason about *type assignments*.

Before we can introduce MLTT, we first need to understand these three fundamental concepts that are common across different type theories.

Formal Notation of Type Theory

Formally, a type theory is made up of *typing rules*.
Typing rules are a deductive system of *typing judgements*.
And typing judgements allow us to reason about *type assignments*.

Before we can introduce MLTT, we first need to understand these three fundamental concepts that are common across different type theories.

Formal Notation of Type Theory – Type Assignments

A *type assignment* looks like this:

$$x : A$$

This says "the object x has type A".

Formal Notation of Type Theory – Type Assignments

Writing $x : A$ is fundamentally different to writing $x \in A$ in set theory (which means "the object x is a member of the set A").

In set theory, stating $x \in A$ is always a valid logical assertion, and asking whether $x \in A$ is always a valid question. But in type theory, terms just *have* types — it makes no sense to "ask" whether or not $x : A$.

To illustrate, the following is a valid logical statement in set theory:

$$((x \in \mathbb{Z}) \wedge (x \geq 0)) \to x \in \mathbb{N}$$

But in type theory, this *assertion* makes no sense.

Instead, the theory's *typing judgements* and *typing rules* are used to *assign* the correct types to the theory's objects.

Formal Notation of Type Theory – Type Assignments

Writing $x : A$ is fundamentally different to writing $x \in A$ in set theory (which means "the object x is a member of the set A").

In set theory, stating $x \in A$ is always a valid logical assertion, and asking whether $x \in A$ is always a valid question. But in type theory, terms just *have* types — it makes no sense to "ask" whether or not $x : A$.

To illustrate, the following is a valid logical statement in set theory:

$$((x \in \mathbb{Z}) \wedge (x \geq 0)) \rightarrow x \in \mathbb{N}$$

But in type theory, this *assertion* makes no sense.

Instead, the theory's *typing judgements* and *typing rules* are used to *assign* the correct types to the theory's objects.

Formal Notation of Type Theory – Type Assignments

Writing $x : A$ is fundamentally different to writing $x \in A$ in set theory (which means "the object x is a member of the set A").

In set theory, stating $x \in A$ is always a valid logical assertion, and asking whether $x \in A$ is always a valid question. But in type theory, terms just *have* types — it makes no sense to "ask" whether or not $x : A$.

To illustrate, the following is a valid logical statement in set theory:

$$((x \in \mathbb{Z}) \wedge (x \geq 0)) \rightarrow x \in \mathbb{N}$$

But in type theory, this *assertion* makes no sense.

Instead, the theory's *typing judgements* and *typing rules* are used to *assign* the correct types to the theory's objects.

Formal Notation of Type Theory – Type Assignments

Writing $x : A$ is fundamentally different to writing $x \in A$ in set theory (which means "the object x is a member of the set A").

In set theory, stating $x \in A$ is always a valid logical assertion, and asking whether $x \in A$ is always a valid question. But in type theory, terms just *have* types — it makes no sense to "ask" whether or not $x : A$.

To illustrate, the following is a valid logical statement in set theory:

$$((x \in \mathbb{Z}) \wedge (x \geq 0)) \to x \in \mathbb{N}$$

But in type theory, this *assertion* makes no sense.

Instead, the theory's *typing judgements* and *typing rules* are used to *assign* the correct types to the theory's objects.

Formal Notation of Type Theory – Typing Judgements

A *typing judgement* looks like this:

$$\Gamma \vdash x : A$$

This says "given a list of typing assignments $\Gamma := \{x_0 : A_0, ..., x_n : A_n\}$, it is the case that $x : A$".

The list of typing assignments Γ is often called *the context*.

Formal Notation of Type Theory – Typing Rules

A *typing rule* looks like this:

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash x : A}{\Gamma \vdash f(x) : B}$$

This says "*if* the object f has type $A \to B$ (i.e. it is a function that takes an A and returns a B) in the context Γ, and the object x has type A in the context Γ, *then* the object $f(x)$ has type B in the context Γ".

Formal Notation of Type Theory

A type theory is made up of typing rules, which are a deductive system of typing judgements, which are themselves a way of reasoning about type assignments.

In order to really understand this, we need to see an example type theory — let's choose something a bit simpler than MLTT: Church's *simply typed λ-calculus*.

Formal Notation of Type Theory

A type theory is made up of typing rules, which are a deductive system of typing judgements, which are themselves a way of reasoning about type assignments.

In order to really understand this, we need to see an example type theory — let's choose something a bit simpler than MLTT: Church's *simply typed λ-calculus*.

Lecture Outline

1. Motivation
2. A Brief History of Type Theory
3. Course Outline
4. Formal Notation of Type Theory
5. **Church's simply typed λ-calculus**
6. Martin-Lof Constructive Type Theory (MLTT)
7. Introducing Agda!

Church's simply typed λ-calculus

A type τ in the *simply typed λ-calculus* is either a *base type* or a *function type*. In this example, we give a single base type of natural numbers $\mathbb{N}$:

$$\tau := \mathbb{N} \mid \tau_1 \to \tau_2$$

A term M of the calculus is either:

- A constant of a base type (e.g. a number $0, 1, 2, ...$),
- A variable $x, y, z, \ldots$,
- A function $\lambda x.M_1$ that binds the argument variable x to the term M_1,
- An application of two terms $M_1 M_2$.

Additionally, *function applications* can be *reduced* by replacing every bound variable x in M_1 with M_2:

$$(\lambda x.M_1)M_2 \rightsquigarrow M_1[x/M_2]$$

Church's simply typed λ-calculus

A type τ in the *simply typed λ-calculus* is either a *base type* or a *function type*. In this example, we give a single base type of natural numbers $\mathbb{N}$:

$$\tau := \mathbb{N} \mid \tau_1 \to \tau_2$$

A term M of the calculus is either:

- A constant of a base type (e.g. a number $0, 1, 2, ...$),
- A variable $x, y, z, \ldots$,
- A function $\lambda x.M_1$ that binds the argument variable x to the term M_1,
- An application of two terms $M_1 M_2$.

Additionally, *function applications* can be *reduced* by replacing every bound variable x in M_1 with M_2:

$$(\lambda x.M_1)M_2 \rightsquigarrow M_1[x/M_2]$$

Church's simply typed λ-calculus

A type τ in the *simply typed λ-calculus* is either a *base type* or a *function type*. In this example, we give a single base type of natural numbers $\mathbb{N}$:

$$\tau := \mathbb{N} \mid \tau_1 \to \tau_2$$

A term M of the calculus is either:

- A constant of a base type (e.g. a number $0, 1, 2, ...$),
- A variable $x, y, z, \ldots$,
- A function $\lambda x.M_1$ that binds the argument variable x to the term M_1,
- An application of two terms $M_1 M_2$.

Additionally, *function applications* can be *reduced* by replacing every bound variable x in M_1 with M_2:

$$(\lambda x.M_1)M_2 \rightsquigarrow M_1[x/M_2]$$

Church's simply typed λ-calculus

Typing rules ensure we can only produce *well-typed* terms — terms that are nonsensical (like $5z$) and which don't fully *reduce* (like $(\lambda x.xx)(\lambda x.xx)$) will be invalid constructions.

The typing rules for constants are straightforward: in any context, we can introduce any constant. We introduce the natural numbers by the Peano axioms.

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}}\ \text{(Zero)} \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \text{succ } n : \mathbb{N}}\ \text{(Succ)}$$

The typing rule for variables is also clear: we can introduce a variable x of any type τ by simply adding it to our context.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{(Var)}$$

Church's simply typed λ-calculus

Typing rules ensure we can only produce *well-typed* terms — terms that are nonsensical (like $5z$) and which don't fully *reduce* (like $(\lambda x.xx)(\lambda x.xx)$) will be invalid constructions.

The typing rules for constants are straightforward: in any context, we can introduce any constant. We introduce the natural numbers by the Peano axioms.

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}}\ (\text{Zero}) \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \text{succ}\ n : \mathbb{N}}\ (\text{Succ})$$

The typing rule for variables is also clear: we can introduce a variable x of any type τ by simply adding it to our context.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\text{Var})$$

Church's simply typed λ-calculus

Typing rules ensure we can only produce *well-typed* terms — terms that are nonsensical (like $5z$) and which don't fully *reduce* (like $(\lambda x.xx)(\lambda x.xx)$) will be invalid constructions.

The typing rules for constants are straightforward: in any context, we can introduce any constant. We introduce the natural numbers by the Peano axioms.

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}} \text{ (Zero)} \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \text{succ } n : \mathbb{N}} \text{ (Succ)}$$

The typing rule for variables is also clear: we can introduce a variable x of any type τ by simply adding it to our context.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \text{ (Var)}$$

Church's simply typed λ-calculus

The latter two typing rules are more intricate.

The typing rule for functions says that if the type assignment $x : \tau$ is in the context, then given a term $M : \sigma$, we can build the function that binds this variable to this term

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda(x : \tau).M : (\tau \to \sigma)} \text{ (Fun)}$$

Finally, the typing rule for application ensures that the term M_1 that we are applying to M_2 is indeed a function — i.e. the type of M_1 must be a function type.

$$\frac{\Gamma \vdash M_1 : \tau \to \sigma \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 M_2 : \sigma} \text{ (App)}$$

Therefore, in the simply typed calculus, we cannot build nonsense terms such as $5z$ or nonterminating ones such as $(\lambda x.xx)(\lambda x.xx)$.

Church's simply typed λ-calculus

The latter two typing rules are more intricate.

The typing rule for functions says that if the type assignment $x : \tau$ is in the context, then given a term $M : \sigma$, we can build the function that binds this variable to this term

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda(x : \tau).M : (\tau \to \sigma)} \text{ (Fun)}$$

Finally, the typing rule for application ensures that the term M_1 that we are applying to M_2 is indeed a function — i.e. the type of M_1 must be a function type.

$$\frac{\Gamma \vdash M_1 : \tau \to \sigma \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 M_2 : \sigma} \text{ (App)}$$

Therefore, in the simply typed calculus, we cannot build nonsense terms such as $5z$ or nonterminating ones such as $(\lambda x.xx)(\lambda x.xx)$.

Church's simply typed λ-calculus

The latter two typing rules are more intricate.

The typing rule for functions says that if the type assignment $x : \tau$ is in the context, then given a term $M : \sigma$, we can build the function that binds this variable to this term

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda(x : \tau).M : (\tau \to \sigma)} \text{ (Fun)}$$

Finally, the typing rule for application ensures that the term M_1 that we are applying to M_2 is indeed a function — i.e. the type of M_1 must be a function type.

$$\frac{\Gamma \vdash M_1 : \tau \to \sigma \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 M_2 : \sigma} \text{ (App)}$$

Therefore, in the simply typed calculus, we cannot build nonsense terms such as $5z$ or nonterminating ones such as $(\lambda x.xx)\,(\lambda x.xx)$.

Lecture Outline

1. Motivation
2. A Brief History of Type Theory
3. Course Outline
4. Formal Notation of Type Theory
5. Church's simply typed λ-calculus
6. **Martin-Lof Constructive Type Theory (MLTT)**
7. Introducing Agda!

MLTT

In 1972, Per Martin-Lof introduced his type theory (MLTT), which we shall see corresponds to (constructive) predicate logic, and is thus a very rich foundation of mathematics. It has been used to formalise much of constructive mathematics, and is the basis of *univalent type theory*.

We will explore MLTT by first introducing its key types and type families via their typing rules. However, after some time we will get bored with this semi-formal mathematical approach, and dive into programming with MLTT in the proof assistant AGDA!

MLTT

In 1972, Per Martin-Lof introduced his type theory (MLTT), which we shall see corresponds to (constructive) predicate logic, and is thus a very rich foundation of mathematics. It has been used to formalise much of constructive mathematics, and is the basis of *univalent type theory*.

We will explore MLTT by first introducing its key types and type families via their typing rules. However, after some time we will get bored with this semi-formal mathematical approach, and dive into programming with MLTT in the proof assistant AGDA!

MLTT

Each type or family of types in MLTT usually has four kinds of typing rules:

- Formation rules, which tell us how to *form* those types,
- Introduction rules, which tell us how to *construct* terms of those types,
- Elimination rules, which tell us how to *destruct* terms of those types,
- Computation rules, which tell us how elimination rules are applied to *reduce* introduced terms.

There are thus quite a few more typing rules for MLTT than for simple type theory! We will focus on the formation, introduction, elimination and computation rules of the key types and type families of the theory. Namely...

MLTT

Each type or family of types in MLTT usually has four kinds of typing rules:

- Formation rules, which tell us how to *form* those types,
- Introduction rules, which tell us how to *construct* terms of those types,
- Elimination rules, which tell us how to *destruct* terms of those types,
- Computation rules, which tell us how elimination rules are applied to *reduce* introduced terms.

There are thus quite a few more typing rules for MLTT than for simple type theory! We will focus on the formation, introduction, elimination and computation rules of the key types and type families of the theory. Namely...

MLTT

Each type or family of types in MLTT usually has four kinds of typing rules:

- Formation rules, which tell us how to *form* those types,
- Introduction rules, which tell us how to *construct* terms of those types,
- Elimination rules, which tell us how to *destruct* terms of those types,
- Computation rules, which tell us how elimination rules are applied to *reduce* introduced terms.

There are thus quite a few more typing rules for MLTT than for simple type theory! We will focus on the formation, introduction, elimination and computation rules of the key types and type families of the theory. Namely...

MLTT

Each type or family of types in MLTT usually has four kinds of typing rules:

- Formation rules, which tell us how to *form* those types,
- Introduction rules, which tell us how to *construct* terms of those types,
- Elimination rules, which tell us how to *destruct* terms of those types,
- Computation rules, which tell us how elimination rules are applied to *reduce* introduced terms.

There are thus quite a few more typing rules for MLTT than for simple type theory! We will focus on the formation, introduction, elimination and computation rules of the key types and type families of the theory. Namely...

MLTT

Each type or family of types in MLTT usually has four kinds of typing rules:

- Formation rules, which tell us how to *form* those types,
- Introduction rules, which tell us how to *construct* terms of those types,
- Elimination rules, which tell us how to *destruct* terms of those types,
- Computation rules, which tell us how elimination rules are applied to *reduce* introduced terms.

There are thus quite a few more typing rules for MLTT than for simple type theory! We will focus on the formation, introduction, elimination and computation rules of the key types and type families of the theory. Namely...

MLTT

Each type or family of types in MLTT usually has four kinds of typing rules:

- Formation rules, which tell us how to *form* those types,
- Introduction rules, which tell us how to *construct* terms of those types,
- Elimination rules, which tell us how to *destruct* terms of those types,
- Computation rules, which tell us how elimination rules are applied to *reduce* introduced terms.

There are thus quite a few more typing rules for MLTT than for simple type theory! We will focus on the formation, introduction, elimination and computation rules of the key types and type families of the theory. Namely...

MLTT

Each type or family of types in MLTT usually has four kinds of typing rules:

- Formation rules, which tell us how to *form* those types,
- Introduction rules, which tell us how to *construct* terms of those types,
- Elimination rules, which tell us how to *destruct* terms of those types,
- Computation rules, which tell us how elimination rules are applied to *reduce* introduced terms.

There are thus quite a few more typing rules for MLTT than for simple type theory! We will focus on the formation, introduction, elimination and computation rules of the key types and type families of the theory. Namely...

MLTT

(a) Function types $\to$,
(b) Natural numbers $\mathbb{N}$,
(c) The unit $\mathbb{1}$ and empty $\mathbb{0}$ types, *(Lecture 2)*
(d) Disjoint union types $+$, *(Lecture 2)*
(e) Binary product types $\times$, *(Lecture 2)*
(f) Dependent function types Π, *(Lecture 2)*
(g) Dependent pair types Σ, *(Lecture 2)*
(h) Identity types $=$ *(Lecture 3)*,
(i) Type universes $\mathcal{U}_0, \mathcal{U}_1, \dots$ *(Lecture 3)*.

MLTT

(a) **Function types $\to$,**
(b) Natural numbers $\mathbb{N}$,
(c) The unit $\mathbb{1}$ and empty $\mathbb{0}$ types, *(Lecture 2)*
(d) Disjoint union types $+$, *(Lecture 2)*
(e) Binary product types $\times$, *(Lecture 2)*
(f) Dependent function types Π, *(Lecture 2)*
(g) Dependent pair types Σ, *(Lecture 2)*
(h) Identity types $=$ *(Lecture 3)*,
(i) Type universes $\mathcal{U}_0, \mathcal{U}_1, \ldots$ *(Lecture 3)*.

MLTT – Function Types

The formation rule for function types simply says that if A and B are types, then $A \to B$ is a type:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type}}{\Gamma \vdash (A \to B) : \text{Type}} \ (\to\text{-Form})$$

The introduction rule for function types is just the 'Fun' rule from the simply-typed λ-calculus:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : (A \to B)} \ (\to\text{-Intro})$$

Meanwhile, the elimination rule is also just the 'App' rule:

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \ (\to\text{-Elim})$$

MLTT – Function Types

The formation rule for function types simply says that if A and B are types, then $A \to B$ is a type:

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash (A \to B) : \mathsf{Type}} \ (\to\text{-Form})$$

The introduction rule for function types is just the 'Fun' rule from the simply-typed λ-calculus:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : (A \to B)} \ (\to\text{-Intro})$$

Meanwhile, the elimination rule is also just the 'App' rule:

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \ (\to\text{-Elim})$$

MLTT – Function Types

The formation rule for function types simply says that if A and B are types, then $A \to B$ is a type:

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash (A \to B) : \mathsf{Type}} \ (\to\text{-Form})$$

The introduction rule for function types is just the 'Fun' rule from the simply-typed λ-calculus:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : (A \to B)} \ (\to\text{-Intro})$$

Meanwhile, the elimination rule is also just the 'App' rule:

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \ (\to\text{-Elim})$$

MLTT – Function Types

The computation rule for function types is also just the computation rule of the λ-calculus: $(\lambda x.M_1)M_2 \rightsquigarrow M_1[x/M_2]$

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).b)\,(a) = b[a/x] : B}\ (\to\text{-Comp})$$

Computation rules use a new kind of typing judgement called *judgemental equality*.

The judgement $x_1 = x_2 : X$ says that the terms $x_1 : X$ and $x_2 : X$ are literally just different names for the same term.

Therefore, because $x_1 = x_2$ we have $x_1 \rightsquigarrow x_2$.

MLTT – Function Types

The computation rule for function types is also just the computation rule of the λ-calculus: $(\lambda x.M_1)M_2 \rightsquigarrow M_1[x/M_2]$

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).b)\,(a) = b[a/x] : B}\ (\to\text{-Comp})$$

Computation rules use a new kind of typing judgement called *judgemental equality*.

The judgement $x_1 = x_2 : X$ says that the terms $x_1 : X$ and $x_2 : X$ are literally just different names for the same term.

Therefore, because $x_1 = x_2$ we have $x_1 \rightsquigarrow x_2$.

MLTT – Function Types

- Formation rules tell us how to *form* those types,

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type}}{\Gamma \vdash (A \to B) : \text{Type}} \ (\to\text{-Form})$$

- Introduction rules tell us how to *construct* terms of those types,

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : (A \to B)} \ (\to\text{-Intro})$$

- Elimination rules tell us how to *destruct* terms of those types,

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \ (\to\text{-Elim})$$

- Computation rules tell us how elimination rules are applied to *reduce* introduced terms.

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).b)\,(a) = b[a/x] : B} \ (\to\text{-Comp})$$

MLTT

(a) Function types $\to$,

(b) **Natural numbers $\mathbb{N}$,**

(c) The unit $\mathbb{1}$ and empty $\mathbb{0}$ types, *(Lecture 2)*

(d) Disjoint union types $+$, *(Lecture 2)*

(e) Binary product types $\times$, *(Lecture 2)*

(f) Dependent function types Π, *(Lecture 2)*

(g) Dependent pair types Σ, *(Lecture 2)*

(h) Identity types $=$ *(Lecture 3)*,

(i) Type universes $\mathcal{U}_0, \mathcal{U}_1, \ldots$ *(Lecture 3)*.

MLTT – Natural Numbers

The formation rule for natural numbers simply says that there is a type of natural numbers:

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathsf{Type}} \ (\mathbb{N}\text{-Form})$$

The introduction rule for natural numbers are just the Peano axioms (which we saw previously as the 'Zero' and 'Succ' rules of our example simply-typed λ-calculus):

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}} \ (\mathbb{N}\text{-Intro}_0) \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{succ}\ n : \mathbb{N}} \ (\mathbb{N}\text{-Intro}_s)$$

MLTT – Natural Numbers

The formation rule for natural numbers simply says that there is a type of natural numbers:

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathsf{Type}}\ (\mathbb{N}\text{-Form})$$

The introduction rule for natural numbers are just the Peano axioms (which we saw previously as the 'Zero' and 'Succ' rules of our example simply-typed λ-calculus):

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}}\ (\mathbb{N}\text{-Intro}_0) \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{succ}\ n : \mathbb{N}}\ (\mathbb{N}\text{-Intro}_\mathsf{s})$$

MLTT – Natural Numbers

The elimination rule for natural numbers is as follows:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \genfrac{}{}{0pt}{}{\Gamma, n{:}\mathbb{N}, p_n{:}P(n)}{\vdash p_s(n,p_n){:}P(\text{succ } n)}}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, n) : P(n)} \ (\mathbb{N}\text{-Elim})$$

This looks quite complicated at first glance!

However, once we recognise that this is just the induction principle on natural numbers — that we recall from A-Level or equivalent mathematics — the rule should become clear: *If...*

- For every $n : \mathbb{N}$ there is a type $P(n)$: Type
- There is a term p_0 of type $P(0)$: Type,
- For every $n : \mathbb{N}$ and $p_n : P(n)$, there is a procedure p_s for constructing a term $p_s(n, p_n)$ of type $P(\text{succ } n)$: Type,

Then, we can construct a term of type $P(n)$.

MLTT – Natural Numbers

The elimination rule for natural numbers is as follows:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \begin{matrix}\Gamma, n{:}\mathbb{N}, p_n{:}P(n) \\ \vdash p_s(n, p_n){:}P(\text{succ } n)\end{matrix}}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, n) : P(n)} \ (\mathbb{N}\text{-Elim})$$

This looks quite complicated at first glance!

However, once we recognise that this is just the induction principle on natural numbers — that we recall from A-Level or equivalent mathematics — the rule should become clear: *If...*

- For every $n : \mathbb{N}$ there is a type $P(n)$: Type
- There is a term p_0 of type $P(0)$: Type,
- For every $n : \mathbb{N}$ and $p_n : P(n)$, there is a procedure p_s for constructing a term $p_s(n, p_n)$ of type $P(\text{succ } n)$: Type,

Then, we can construct a term of type $P(n)$.

MLTT – Natural Numbers

The elimination rule for natural numbers is as follows:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \genfrac{}{}{0pt}{}{\Gamma, n{:}\mathbb{N}, p_n{:}P(n)}{\vdash p_s(n,p_n){:}P(\text{succ } n)}}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, n) : P(n)} \ (\mathbb{N}\text{-Elim})$$

This looks quite complicated at first glance!

However, once we recognise that this is just the induction principle on natural numbers — that we recall from A-Level or equivalent mathematics — the rule should become clear: *If...*

- For every $n : \mathbb{N}$ there is a type $P(n)$: Type
- There is a term p_0 of type $P(0)$: Type,
- For every $n : \mathbb{N}$ and $p_n : P(n)$, there is a procedure p_s for constructing a term $p_s(n, p_n)$ of type $P(\text{succ } n)$: Type,

Then, we can construct a term of type $P(n)$.

MLTT – Natural Numbers

The elimination rule for natural numbers is as follows:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \begin{array}{c}\Gamma, n{:}\mathbb{N}, p_n{:}P(n) \\ \vdash p_s(n, p_n){:}P(\text{succ }\ n)\end{array}}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, n) : P(n)} \ (\mathbb{N}\text{-Elim})$$

This looks quite complicated at first glance!

However, once we recognise that this is just the induction principle on natural numbers — that we recall from A-Level or equivalent mathematics — the rule should become clear: *If...*

- For every $n : \mathbb{N}$ there is a type $P(n) : \text{Type}$
- There is a term p_0 of type $P(0) : \text{Type}$,
- For every $n : \mathbb{N}$ and $p_n : P(n)$, there is a procedure p_s for constructing a term $p_s(n, p_n)$ of type $P(\text{succ }\ n) : \text{Type}$,

Then, we can construct a term of type $P(n)$.

MLTT – Natural Numbers

The elimination rule for natural numbers is as follows:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \begin{array}{c}\Gamma, n{:}\mathbb{N}, p_n{:}P(n) \\ \vdash p_s(n,p_n){:}P(\text{succ } n)\end{array}}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, n) : P(n)} \ (\mathbb{N}\text{-Elim})$$

This looks quite complicated at first glance!

However, once we recognise that this is just the induction principle on natural numbers — that we recall from A-Level or equivalent mathematics — the rule should become clear: *If...*

- For every $n : \mathbb{N}$ there is a type $P(n)$: Type
- There is a term p_0 of type $P(0)$: Type,
- For every $n : \mathbb{N}$ and $p_n : P(n)$, there is a procedure p_s for constructing a term $p_s(n, p_n)$ of type $P(\text{succ } n)$: Type,

Then, we can construct a term of type $P(n)$.

MLTT – Natural Numbers

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathsf{Type}}\ (\mathbb{N}\text{-Form}) \qquad \frac{}{\Gamma \vdash 0 : \mathbb{N}}\ (\mathbb{N}\text{-Intro}_0) \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{succ}\ n : \mathbb{N}}\ (\mathbb{N}\text{-Intro}_s)$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \mathsf{Type} \quad \Gamma \vdash p_0 : P(0) \quad {}^{\Gamma, n:\mathbb{N}, p_n : P(n)}_{\vdash p_s(n, p_n) : P(\mathsf{succ}\ n)}}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, n) : P(n)}\ (\mathbb{N}\text{-Elim})$$

Only the computation rules remain, which tell us how the elimination rule should reduce terms arising from either the two introduction rules, i.e. zero and successors:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \mathsf{Type} \quad \Gamma \vdash p_0 : P(0) \quad {}^{\Gamma, n:\mathbb{N}, p_n : P(n)}_{\vdash p_s(n, p_n) : P(\mathsf{succ}\ n)}}{\Gamma \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, 0) = p_0 : P(0)}\ (\mathbb{N}\text{-Comp}_0)$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \mathsf{Type} \quad \Gamma \vdash p_0 : P(0) \quad {}^{\Gamma, n:\mathbb{N}, p_n : P(n)}_{\vdash p_s(n, p_n) : P(\mathsf{succ}\ n)}}{\Gamma, n : \mathbb{N} \vdash {}^{\mathbb{N}\text{-induction}(P, p_0, p_s, \mathsf{succ}\ n)}_{= p_s(n, \mathbb{N}\text{-}induction(P, p_0, p_s, n))} : P(\mathsf{succ}\ n)}\ (\mathbb{N}\text{-Comp}_s)$$

MLTT – Natural Numbers

$$\frac{}{\Gamma \vdash \mathbb{N} : \text{Type}} \; (\mathbb{N}\text{-Form}) \quad \frac{}{\Gamma \vdash 0 : \mathbb{N}} \; (\mathbb{N}\text{-Intro}_0) \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \text{succ } n : \mathbb{N}} \; (\mathbb{N}\text{-Intro}_s)$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n : P(n) \\ \vdash p_s(n, p_n) : P(\text{succ } n)}}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, n) : P(n)} \; (\mathbb{N}\text{-Elim})$$

Only the computation rules remain, which tell us how the elimination rule should reduce terms arising from either the two introduction rules, i.e. zero and successors:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n : P(n) \\ \vdash p_s(n, p_n) : P(\text{succ } n)}}{\Gamma \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, 0) = p_0 : P(0)} \; (\mathbb{N}\text{-Comp}_0)$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n : P(n) \\ \vdash p_s(n, p_n) : P(\text{succ } n)}}{\Gamma, n : \mathbb{N} \vdash \substack{\mathbb{N}\text{-induction}(P, p_0, p_s, \text{succ } n) \\ = p_s(n, \mathbb{N}\text{-}induction(P, p_0, p_s, n))} : P(\text{succ } n)} \; (\mathbb{N}\text{-Comp}_s)$$

MLTT – Natural Numbers

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathsf{Type}}\ (\mathbb{N}\text{-Form}) \quad \frac{}{\Gamma \vdash 0 : \mathbb{N}}\ (\mathbb{N}\text{-Intro}_0) \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{succ}\ n : \mathbb{N}}\ (\mathbb{N}\text{-Intro}_s)$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \mathsf{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n:P(n) \\ \vdash p_s(n,p_n):P(\mathsf{succ}\ n)}}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, n) : P(n)}\ (\mathbb{N}\text{-Elim})$$

Only the computation rules remain, which tell us how the elimination rule should reduce terms arising from either the two introduction rules, i.e. zero and successors:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \mathsf{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n:P(n) \\ \vdash p_s(n,p_n):P(\mathsf{succ}\ n)}}{\Gamma \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, 0) = p_0 : P(0)}\ (\mathbb{N}\text{-Comp}_0)$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \mathsf{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n:P(n) \\ \vdash p_s(n,p_n):P(\mathsf{succ}\ n)}}{\Gamma, n : \mathbb{N} \vdash \substack{\mathbb{N}\text{-induction}(P,p_0,p_s,\mathsf{succ}\ n) \\ = p_s(n, \mathbb{N}\text{-}induction(P,p_0,p_s,n))} : P(\mathsf{succ}\ n)}\ (\mathbb{N}\text{-Comp}_s)$$

MLTT – Natural Numbers

The natural numbers are an *inductive type*. Defining inductive types in MLTT follows a straightforward recipe:

- Form the type,
- Define a finite number of introduction rules for the type, with at least one of those rules as a 'base rule',
- Define a single inductive elimination rule,
- Define a computation rule for each introduction rule.

We could follow this recipe to define formal typing rules for more types, but defining operations and proofs would eventually become unwieldy in this formal framework.

So let's ditch it and learn about MLTT using Agda instead!

MLTT – Natural Numbers

The natural numbers are an *inductive type*. Defining inductive types in MLTT follows a straightforward recipe:

- Form the type,
- Define a finite number of introduction rules for the type, with at least one of those rules as a 'base rule',
- Define a single inductive elimination rule,
- Define a computation rule for each introduction rule.

We could follow this recipe to define formal typing rules for more types, but defining operations and proofs would eventually become unwieldy in this formal framework.

So let's ditch it and learn about MLTT using Agda instead!

MLTT – Natural Numbers

The natural numbers are an *inductive type*. Defining inductive types in MLTT follows a straightforward recipe:

- Form the type,
- Define a finite number of introduction rules for the type, with at least one of those rules as a ‘base rule’,
- Define a single inductive elimination rule,
- Define a computation rule for each introduction rule.

We could follow this recipe to define formal typing rules for more types, but defining operations and proofs would eventually become unwieldy in this formal framework.

So let’s ditch it and learn about MLTT using Agda instead!

MLTT – Natural Numbers

The natural numbers are an *inductive type*. Defining inductive types in MLTT follows a straightforward recipe:

- Form the type,
- Define a finite number of introduction rules for the type, with at least one of those rules as a 'base rule',
- Define a single inductive elimination rule,
- Define a computation rule for each introduction rule.

We could follow this recipe to define formal typing rules for more types, but defining operations and proofs would eventually become unwieldy in this formal framework.

So let's ditch it and learn about MLTT using Agda instead!

Lecture Outline

1. Motivation
2. A Brief History of Type Theory
3. Course Outline
4. Formal Notation of Type Theory
5. Church's simply typed λ-calculus
6. Martin-Lof Constructive Type Theory (MLTT)
7. **Introducing Agda!**

Introducing Agda!

Dependently-typed programming languages based on MLTT have been developed for the past fifty years (twice the age of MGS!).

These languages emphasise the computational aspects of MLTT and double as proof assistants, meaning we can run our proofs and compute meaningful information from them.

Introducing Agda!

Dependently-typed programming languages based on MLTT have been developed for the past fifty years (twice the age of MGS!).

These languages emphasise the computational aspects of MLTT and double as proof assistants, meaning we can run our proofs and compute meaningful information from them.

Introducing Agda!

Agda is the latest language in this long tradition. The first version of Agda (1999) was written by Catarina Coquand, while the second (2007) is by Ulf Norell and Andreas Abel.

We will not investigate Agda's type system formally, due to its complexity, but we will use a subset of Agda's features to continue our investigation of MLTT.

Introducing Agda!

Agda is the latest language in this long tradition. The first version of Agda (1999) was written by Catarina Coquand, while the second (2007) is by Ulf Norell and Andreas Abel.

We will not investigate Agda's type system formally, due to its complexity, but we will use a subset of Agda's features to continue our investigation of MLTT.

Introducing Agda!

In the final part of this lecture, we will take a look at defining some basic functions in Agda, and defining the natural numbers.

Agda infamously uses ‘Set‘ for the type of types (I believe many of the implementers now regret this). However, there is a quick fix...

Introducing Agda!

In the final part of this lecture, we will take a look at defining some basic functions in Agda, and defining the natural numbers.

Agda infamously uses ‘Set‘ for the type of types (I believe many of the implementers now regret this). However, there is a quick fix...

Introducing Agda!

Agda already has built-in support for defining functions. The syntax of Agda is similar to the syntax of Haskell. The : symbol starts the type definition, while the = symbol starts the definition of the term itself.

MLTT in Agda

(a) **Function types $\rightarrow$,**
(b) Natural numbers $\mathbb{N}$,
(c) The unit $\mathbb{1}$ and empty $\mathbb{0}$ types, *(Lecture 2)*
(d) Disjoint union types $+$, *(Lecture 2)*
(e) Binary product types $\times$, *(Lecture 2)*
(f) Dependent function types Π, *(Lecture 2)*
(g) Dependent pair types Σ, *(Lecture 2)*
(h) Identity types $=$ *(Lecture 3)*,
(i) Type universes $\mathcal{U}_0, \mathcal{U}_1, \ldots$ *(Lecture 3)*.

MLTT in Agda – Function Types

Despite Agda having built-in support for functions, we can actually re-define their typing rules.

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash (A \to B) : \mathsf{Type}}\ (\to\text{-Form})$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : (A \to B)}\ (\to\text{-Intro})$$

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}\ (\to\text{-Elim})$$

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).b)\,(a) = b[a/x] : B}\ (\to\text{-Comp})$$

Unfortunately, we cannot define the introduction rule in Agda; it is instead taken care of directly in Agda's type system. Note also that the computation rule is derived automatically, as with the `K` function.

MLTT in Agda – Function Types

Despite Agda having built-in support for functions, we can actually re-define their typing rules.

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma \vdash B : \mathrm{Type}}{\Gamma \vdash (A \to B) : \mathrm{Type}} \ (\to\text{-Form})$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : (A \to B)} \ (\to\text{-Intro})$$

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \ (\to\text{-Elim})$$

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).b)\,(a) = b[a/x] : B} \ (\to\text{-Comp})$$

Unfortunately, we cannot define the introduction rule in Agda; it is instead taken care of directly in Agda's type system. Note also that the computation rule is derived automatically, as with the `K` function.

MLTT in Agda

(a) Function types $\to$,
(b) **Natural numbers $\mathbb{N}$,**
(c) The unit $\mathbb{1}$ and empty $\mathbb{0}$ types, *(Lecture 2)*
(d) Disjoint union types $+$, *(Lecture 2)*
(e) Binary product types $\times$, *(Lecture 2)*
(f) Dependent function types Π, *(Lecture 2)*
(g) Dependent pair types Σ, *(Lecture 2)*
(h) Identity types $=$ *(Lecture 3)*,
(i) Type universes $\mathcal{U}_0, \mathcal{U}_1, \ldots$ *(Lecture 3)*.

MLTT in Agda – Natural Numbers

Agda has built-in support for defining inductive types via the recipe we discussed earlier, using the keyword `data`. The formation rule and introduction rules for natural numbers can be thus given in a `data` definition.

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathsf{Type}}\ (\mathbb{N}\text{-Form})$$

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}}\ (\mathbb{N}\text{-Intro}_0) \quad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{succ}\ n : \mathbb{N}}\ (\mathbb{N}\text{-Intro}_s)$$

MLTT in Agda – Natural Numbers

In order to formalise the elimination rule of natural numbers, we have to give both of its computation rules. We do this by *pattern matching* on $n : \mathbb{N}$.

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n:P(n) \\ \vdash p_s(n, p_n):P(\text{succ } n)}}{\Gamma, n : \mathbb{N} \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, n) : P(n)} \; (\mathbb{N}\text{-Elim})$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n:P(n) \\ \vdash p_s(n, p_n):P(\text{succ } n)}}{\Gamma \vdash \mathbb{N}\text{-induction}(P, p_0, p_s, 0) = p_0 : P(0)} \; (\mathbb{N}\text{-Comp}_0)$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) : \text{Type} \quad \Gamma \vdash p_0 : P(0) \quad \substack{\Gamma, n:\mathbb{N}, p_n:P(n) \\ \vdash p_s(n, p_n):P(\text{succ } n)}}{\Gamma, n : \mathbb{N} \vdash \substack{\mathbb{N}\text{-induction}(P, p_0, p_s, \text{succ } n) \\ = p_s(n, \mathbb{N}\textit{-induction}(P, p_0, p_s, n))} : P(\text{succ } n)} \; (\mathbb{N}\text{-Comp}_\text{s})$$

Next time...

Next lecture, we will look at formalising more types of MLTT in Agda. Furthermore, we will illuminate the *propositions-as-types* interpretation of type theory.

Please join me in the exercise classes, where you can get experience of programming Type Theory in Agda yourself!