Lean's logical foundations

David Loeffler

Charmey, January 2024

FernUni.ch
UniDistance.ch

1. **Foundations of mathematics**

2. The Curry–Howard correspondence

3. What it means in practice

FernUni.ch
UniDistance.ch

Set-theoretic foundations

- Cantor: “Everything is a set”

Set-theoretic foundations

- Cantor: “Everything is a set”
- *Zermelo–Fraenkel with choice* (ZFC): particular axiomatization, believed consistent

FernUni.ch
UniDistance.ch

Set-theoretic foundations

- Cantor: “Everything is a set”
- *Zermelo–Fraenkel with choice* (ZFC): particular axiomatization, believed consistent
- Sometimes include *large cardinal axioms* (equivalently: *Grothendieck universes*) – not provable from ZFC alone

FernUni.ch
UniDistance.ch

Alternative foundations

- The exact foundations we use make **very little practical difference** to “everyday” mathematics

Alternative foundations

- The exact foundations we use make **very little practical difference** to “everyday” mathematics
- (Who, in this room, knows what the axioms of ZFC are?)

Alternative foundations

- The exact foundations we use make **very little practical difference** to “everyday” mathematics
- (Who, in this room, knows what the axioms of ZFC are?)
- But if we want to **formalize** our proofs, we need to choose a foundation to build on.

Alternative foundations

- The exact foundations we use make **very little practical difference** to “everyday” mathematics
- (Who, in this room, knows what the axioms of ZFC are?)
- But if we want to **formalize** our proofs, we need to choose a foundation to build on.
- Some proof assistants (e.g. **Mizar**) do use ZFC, but there are other options.

FernUni.ch
UniDistance.ch

Lambda calculus

- **Alonzo Church**: pioneer of theoretical computer science

Lambda calculus

- **Alonzo Church**: pioneer of theoretical computer science
- λ-calculus: formal language describing **computation**, with *function application* and *abstraction* as basic operations

FernUni.ch
UniDistance.ch

Lambda calculus

- **Alonzo Church**: pioneer of theoretical computer science
- λ-calculus: formal language describing **computation**, with *function application* and *abstraction* as basic operations
  - Instead of “everything is a set” (Cantor), now “everything is a function”!

FernUni.ch
UniDistance.ch

Lambda calculus

- **Alonzo Church**: pioneer of theoretical computer science
- λ-calculus: formal language describing **computation**, with *function application* and *abstraction* as basic operations
  - Instead of "everything is a set" (Cantor), now "everything is a function"!
- Read $\lambda x, e$ as "the function $x \mapsto e$"

FernUni.ch
UniDistance.ch

Lambda calculus

- **Alonzo Church**: pioneer of theoretical computer science
- λ-calculus: formal language describing **computation**, with *function application* and *abstraction* as basic operations
  - Instead of “everything is a set” (Cantor), now “everything is a function”!
- Read $\lambda x, e$ as “the function $x \mapsto e$”
  - E.g. $\lambda x, \lambda y, x$ is “the function $(x, y) \mapsto x$”

Lambda calculus

- **Alonzo Church**: pioneer of theoretical computer science
- λ-calculus: formal language describing **computation**, with *function application* and *abstraction* as basic operations
  - Instead of “everything is a set” (Cantor), now “everything is a function”!
- Read $\lambda x, e$ as “the function $x \mapsto e$”
  - E.g. $\lambda x, \lambda y, x$ is “the function $(x, y) \mapsto x$”
- Reduction rules like $(\lambda x, e)\, x \rightsquigarrow e$

FernUni.ch
UniDistance.ch

Lambda calculus

- Flexible enough to **encode Peano arithmetic**

FernUni.ch
UniDistance.ch

Lambda calculus

- Flexible enough to **encode Peano arithmetic**
  - "2" corresponds to $\lambda f, \lambda x, f\ f\ x$ (the "iterate twice" map on functions)

Lambda calculus

- Flexible enough to **encode Peano arithmetic**
  - "2" corresponds to $\lambda f, \lambda x, f\ f\ x$ (the "iterate twice" map on functions)
- Problem: it's **too** flexible

FernUni.ch
UniDistance.ch

Lambda calculus

- Flexible enough to **encode Peano arithmetic**
  - "2" corresponds to $\lambda f, \lambda x, f\ f\ x$ (the "iterate twice" map on functions)
- Problem: it's **too** flexible
  - Computationally undecidable whether one λ-expression reduces to another

FernUni.ch
UniDistance.ch

Lambda calculus

- Flexible enough to **encode Peano arithmetic**
  - "2" corresponds to $\lambda f, \lambda x, f\ f\ x$ (the "iterate twice" map on functions)
- Problem: it's **too** flexible
  - Computationally undecidable whether one λ-expression reduces to another
- Lacks notion of *domain* of a function: can apply a function to anything, including itself

FernUni.ch
UniDistance.ch

Typed lambda calculus

- **Typed** λ-calculus: each variable has a *type*, intended to model data types on a computer (numbers, text, lists...)

FernUni.ch
UniDistance.ch

Typed lambda calculus

- **Typed** λ-calculus: each variable has a *type*, intended to model data types on a computer (numbers, text, lists...)

- If α, β are types, then $\alpha \to \beta$ ("functions from α to β") is a type

FernUni.ch
UniDistance.ch

Typed lambda calculus

- **Typed** λ-calculus: each variable has a *type*, intended to model data types on a computer (numbers, text, lists...)
- If α, β are types, then $\alpha \to \beta$ ("functions from α to β") is a type
  - If *f* has type $\alpha \to \beta$, and *e* has type α, then *f e* has type β

FernUni.ch
UniDistance.ch

Typed lambda calculus

- **Typed** λ-calculus: each variable has a *type*, intended to model data types on a computer (numbers, text, lists...)

- If α, β are types, then $\alpha \to \beta$ ("functions from α to β") is a type
  - If *f* has type $\alpha \to \beta$, and *e* has type α, then *f e* has type β
  - If *e* doesn't have type α, dismiss *f e* as **meaningless** ("not well-typed")

FernUni.ch
UniDistance.ch

Typed lambda calculus

- **Typed** λ-calculus: each variable has a *type*, intended to model data types on a computer (numbers, text, lists...)

- If α, β are types, then $\alpha \to \beta$ ("functions from α to β") is a type
  - If *f* has type $\alpha \to \beta$, and *e* has type α, then *f e* has type β
  - If *e* doesn't have type α, dismiss *f e* as **meaningless** ("not well-typed")

- Abstractions look like "$\lambda(x : \alpha)$, *e*" – interpret this as "the function sending *x* **of type** α to *e*"

FernUni.ch
UniDistance.ch

Typed lambda calculus

- **Typed** λ-calculus: each variable has a *type*, intended to model data types on a computer (numbers, text, lists...)

- If α, β are types, then $\alpha \to \beta$ ("functions from α to β") is a type
  - If *f* has type $\alpha \to \beta$, and *e* has type α, then *f e* has type β
  - If *e* doesn't have type α, dismiss *f e* as **meaningless** ("not well-typed")

- Abstractions look like "$\lambda(x : \alpha), e$" – interpret this as "the function sending *x* **of type** α to *e*"
  - If *e* has type β, then $\lambda(x : \alpha), e$ has type $\alpha \to \beta$

FernUni.ch
UniDistance.ch

Typed lambda calculus

- **Typed** λ-calculus: each variable has a *type*, intended to model data types on a computer (numbers, text, lists...)

- If α, β are types, then $\alpha \to \beta$ ("functions from α to β") is a type
  - If *f* has type $\alpha \to \beta$, and *e* has type α, then *f e* has type β
  - If *e* doesn't have type α, dismiss *f e* as **meaningless** ("not well-typed")

- Abstractions look like "$\lambda(x : \alpha), e$" – interpret this as "the function sending *x* **of type α** to *e*"
  - If *e* has type β, then $\lambda(x : \alpha), e$ has type $\alpha \to \beta$

- Only care about well-typed expressions (**terms**)

FernUni.ch
UniDistance.ch

Typed lambda calculus

- **Typed** λ-calculus: each variable has a *type*, intended to model data types on a computer (numbers, text, lists...)

- If α, β are types, then $\alpha \to \beta$ ("functions from α to β") is a type
  - If *f* has type $\alpha \to \beta$, and *e* has type α, then *f e* has type β
  - If *e* doesn't have type α, dismiss *f e* as **meaningless** ("not well-typed")

- Abstractions look like "$\lambda(x : \alpha), e$" – interpret this as "the function sending *x* **of type** α to *e*"
  - If *e* has type β, then $\lambda(x : \alpha), e$ has type $\alpha \to \beta$

- Only care about well-typed expressions (**terms**)

- Can algorithmically check if an expression is well-typed (and determine its type)

FernUni.ch
UniDistance.ch

Curry–Howard correspondence

Observation by Haskell Curry: there is a *correspondence* between **models of computation** and **formal systems for proofs**.

- Types $\leftrightarrow$ propositions

FernUni.ch
UniDistance.ch

Curry–Howard correspondence

Observation by Haskell Curry: there is a *correspondence* between **models of computation** and **formal systems for proofs**.

- Types $\leftrightarrow$ propositions
- Term of type $\alpha \leftrightarrow$ proof of proposition α

FernUni.ch
UniDistance.ch

Curry–Howard correspondence

Observation by Haskell Curry: there is a *correspondence* between **models of computation** and **formal systems for proofs**.

- Types $\leftrightarrow$ propositions
- Term of type $\alpha \leftrightarrow$ proof of proposition α
- Function $\alpha \to \beta \leftrightarrow$ implication $\alpha \Rightarrow \beta$

FernUni.ch
UniDistance.ch

Curry–Howard correspondence

Observation by Haskell Curry: there is a *correspondence* between **models of computation** and **formal systems for proofs**.

- Types $\leftrightarrow$ propositions
- Term of type $\alpha \leftrightarrow$ proof of proposition α
- Function $\alpha \to \beta \leftrightarrow$ implication $\alpha \Rightarrow \beta$
- Type is inhabited $\leftrightarrow$ proposition is proved

FernUni.ch
UniDistance.ch

Dependent types

- Church's typed λ-calculus $\leftrightarrow$ intuitionistic propositional logic

Dependent types

- Church's typed λ-calculus $\leftrightarrow$ intuitionistic propositional logic
  - Very weak logical system

FernUni.ch
UniDistance.ch

Dependent types

- Church's typed λ-calculus $\leftrightarrow$ intuitionistic propositional logic
  - Very weak logical system
  - No quantifiers, no law of excluded middle

FernUni.ch
UniDistance.ch

Dependent types

- Church's typed λ-calculus $\leftrightarrow$ intuitionistic propositional logic
  - Very weak logical system
  - No quantifiers, no law of excluded middle

- Howard, de Bruijn, Martin-Löf, ... : richer type theories

FernUni.ch
UniDistance.ch

Dependent types

- Church's typed λ-calculus $\leftrightarrow$ intuitionistic propositional logic
  - Very weak logical system
  - No quantifiers, no law of excluded middle
- Howard, de Bruijn, Martin-Löf, ... : richer type theories
  - Families of types with parameters (*dependent types*) – corresponds to $\forall$ quantifier

FernUni.ch
UniDistance.ch

Dependent types

- Church's typed λ-calculus $\leftrightarrow$ intuitionistic propositional logic
  - Very weak logical system
  - No quantifiers, no law of excluded middle
- Howard, de Bruijn, Martin-Löf, ... : richer type theories
  - Families of types with parameters (*dependent types*) – corresponds to $\forall$ quantifier
  - Pairs, disjoint unions, etc

Dependent types

- Church's typed λ-calculus $\leftrightarrow$ intuitionistic propositional logic
  - Very weak logical system
  - No quantifiers, no law of excluded middle
- Howard, de Bruijn, Martin-Löf, ... : richer type theories
  - Families of types with parameters (*dependent types*) – corresponds to $\forall$ quantifier
  - Pairs, disjoint unions, etc
  - special type `Nat` corresponding to $\mathbb{N}$

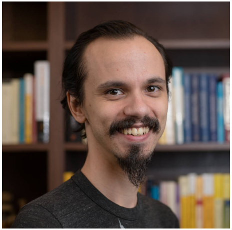
FernUni.ch
UniDistance.ch

Dependent types

- Church's typed λ-calculus $\leftrightarrow$ intuitionistic propositional logic
  - Very weak logical system
  - No quantifiers, no law of excluded middle
- Howard, de Bruijn, Martin-Löf, ... : richer type theories
  - Families of types with parameters (*dependent types*) – corresponds to $\forall$ quantifier
  - Pairs, disjoint unions, etc
  - special type `Nat` corresponding to $\mathbb{N}$
- More expressive power, but type-checking becomes harder

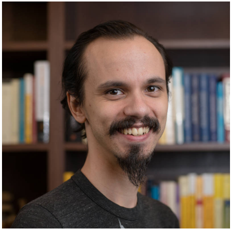
Lean's type theory

- *Calculus of Inductive Constructions* (CIC): flavour of type theory used in Coq and Lean



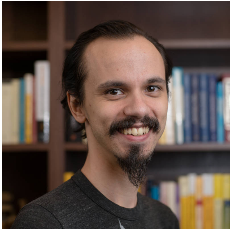
Lean's type theory

- *Calculus of Inductive Constructions* (CIC): flavour of type theory used in Coq and Lean
- Extremely flexible: allows dependent types and quantification over types



Lean’s type theory

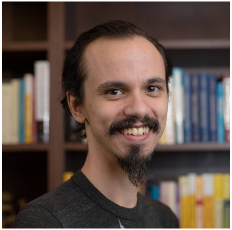
- *Calculus of Inductive Constructions* (CIC): flavour of type theory used in Coq and Lean
- Extremely flexible: allows dependent types and quantification over types
- To avoid paradoxes, need a hierarchy of *type universes*



FernUni.ch
UniDistance.ch

Lean’s type theory

- *Calculus of Inductive Constructions* (CIC): flavour of type theory used in Coq and Lean
- Extremely flexible: allows dependent types and quantification over types
- To avoid paradoxes, need a hierarchy of *type universes*
- **Theorem** (Carneiro): CIC has the same consistency strength as “ZFC with countably many inaccessible cardinals”.



FernUni.ch
UniDistance.ch

FernUni.ch
UniDistance.ch

Types and terms

- Anything you can construct in Lean is either a **type**, or a **term** of some type.

Types and terms

- Anything you can construct in Lean is either a **type**, or a **term** of some type.
- Roughly: *types are like sets*, and *terms are like elements*.

FernUni.ch
UniDistance.ch

Types and terms

- Anything you can construct in Lean is either a **type**, or a **term** of some type.
- Roughly: *types are like sets*, and *terms are like elements*.
- Key difference: each term has **one and only one type** (the elements know which set they belong to).

FernUni.ch
UniDistance.ch

Types and terms

- Anything you can construct in Lean is either a **type**, or a **term** of some type.
- Roughly: *types are like sets*, and *terms are like elements*.
- Key difference: each term has **one and only one type** (the elements know which set they belong to).
  - E.g. $\mathbb{Q}$ is a type

FernUni.ch
UniDistance.ch

Types and terms

- Anything you can construct in Lean is either a **type**, or a **term** of some type.
- Roughly: *types are like sets*, and *terms are like elements*.
- Key difference: each term has **one and only one type** (the elements know which set they belong to).
  - E.g. $\mathbb{Q}$ is a type
  - There is a term of type $\mathbb{Q}$ which stands for $\frac{3}{4}$

FernUni.ch
UniDistance.ch

Types and terms

- Anything you can construct in Lean is either a **type**, or a **term** of some type.
- Roughly: *types are like sets*, and *terms are like elements*.
- Key difference: each term has **one and only one type** (the elements know which set they belong to).
  - E.g. $\mathbb{Q}$ is a type
  - There is a term of type $\mathbb{Q}$ which stands for $\frac{3}{4}$
  - But "$\frac{3}{4}$ of type $\mathbb{Q}$", "$\frac{3}{4}$ of type $\mathbb{R}$" and "$\frac{3}{4}$ of type $\mathbb{C}$" are not the same object.

FernUni.ch
UniDistance.ch

Types and terms

- Anything you can construct in Lean is either a **type**, or a **term** of some type.
- Roughly: *types are like sets*, and *terms are like elements*.
- Key difference: each term has **one and only one type** (the elements know which set they belong to).
  - E.g. $\mathbb{Q}$ is a type
  - There is a term of type $\mathbb{Q}$ which stands for $\frac{3}{4}$
  - But "$\frac{3}{4}$ of type $\mathbb{Q}$", "$\frac{3}{4}$ of type $\mathbb{R}$" and "$\frac{3}{4}$ of type $\mathbb{C}$" are not the same object.
- The command `#check blah` will check whether the expression `blah` is well-typed, and if so, print its type.

Types and terms

- Anything you can construct in Lean is either a **type**, or a **term** of some type.
- Roughly: *types are like sets*, and *terms are like elements*.
- Key difference: each term has **one and only one type** (the elements know which set they belong to).
  - E.g. $\mathbb{Q}$ is a type
  - There is a term of type $\mathbb{Q}$ which stands for $\frac{3}{4}$
  - But "$\frac{3}{4}$ of type $\mathbb{Q}$", "$\frac{3}{4}$ of type $\mathbb{R}$" and "$\frac{3}{4}$ of type $\mathbb{C}$" are not the same object.
- The command `#check blah` will check whether the expression `blah` is well-typed, and if so, print its type.
- Conventionally: names of terms start with **small Latin letters**, names of types with **capitals** or **Greek letters**.

FernUni.ch
UniDistance.ch

Propositions and proofs

- Propositions (Props) are a special kind of type: they stand for logical statements that might or might not be true, i.e. **statements of theorems**.

FernUni.ch
UniDistance.ch

Propositions and proofs

- Propositions (Props) are a special kind of type: they stand for logical statements that might or might not be true, i.e. **statements of theorems**.
- If `P` is a Prop, then constructing a term of type `P` amounts to proving the corresponding theorem (Curry–Howard).

FernUni.ch
UniDistance.ch

Propositions and proofs

- Propositions (Props) are a special kind of type: they stand for logical statements that might or might not be true, i.e. **statements of theorems**.
- If `P` is a Prop, then constructing a term of type `P` amounts to proving the corresponding theorem (Curry–Howard).
- So terms whose type is a Prop are **proofs of theorems**.

FernUni.ch
UniDistance.ch

Propositions and proofs

- Propositions (Props) are a special kind of type: they stand for logical statements that might or might not be true, i.e. **statements of theorems**.
- If `P` is a Prop, then constructing a term of type `P` amounts to proving the corresponding theorem (Curry–Howard).
- So terms whose type is a Prop are **proofs of theorems**.
  - Lean's mathematics library contains propositions called `FermatLastTheorem` and `RiemannHypothesis`.

FernUni.ch
UniDistance.ch

Propositions and proofs

- Propositions (Props) are a special kind of type: they stand for logical statements that might or might not be true, i.e. **statements of theorems**.
- If `P` is a Prop, then constructing a term of type `P` amounts to proving the corresponding theorem (Curry–Howard).
- So terms whose type is a Prop are **proofs of theorems**.
  - Lean's mathematics library contains propositions called `FermatLastTheorem` and `RiemannHypothesis`.
  - Sadly it does not contain terms of those types.

Propositions and proofs

- Propositions (Props) are a special kind of type: they stand for logical statements that might or might not be true, i.e. **statements of theorems**.
- If `P` is a Prop, then constructing a term of type `P` amounts to proving the corresponding theorem (Curry–Howard).
- So terms whose type is a Prop are **proofs of theorems**.
  - Lean's mathematics library contains propositions called `FermatLastTheorem` and `RiemannHypothesis`.
  - Sadly it does not contain terms of those types.
  - It does contain a term `fermatLastTheoremFour` which is a proof of the $n = 4$ case of FLT.

FernUni.ch
UniDistance.ch

Propositions and proofs

- Propositions (Props) are a special kind of type: they stand for logical statements that might or might not be true, i.e. **statements of theorems**.
- If `P` is a Prop, then constructing a term of type `P` amounts to proving the corresponding theorem (Curry–Howard).
- So terms whose type is a Prop are **proofs of theorems**.
  - Lean's mathematics library contains propositions called `FermatLastTheorem` and `RiemannHypothesis`.
  - Sadly it does not contain terms of those types.
  - It does contain a term `fermatLastTheoremFour` which is a proof of the $n = 4$ case of FLT.
- "Theorems with hypotheses" **are functions**: a proof that $P \Rightarrow Q$ is a function from proofs of P to proofs of Q.

FernUni.ch
UniDistance.ch