Lecture Notes for ICL 2026

Gert Smolka, Saarland University

April 17, 2026

These notes accompany my teaching of the course ICL in 2026 at Saarland University. The course is about interactive theorem proving and computational type theory. We use the proof assistant Rocq and follow the textbook MPCTT (covering substantially more material). I have been teaching variants of the course since 2010. As it turns out, my presentation of the basic material still changes every year. Compared to MPCTT, I skip and reorder material, omit and postpone technical details, and experiment with a simplified, lecture-oriented presentation of the material.

The plan is to write a section for each lecture and making it available soon after the lecture. Lectures take 90 minutes and happen twice a week. Altogether we have 28 lectures.

There are accompanying Rocq files (look for `ln_*.v` in the MPCTT git repo) covering all definitions and proofs in the notes. The details missing in the text can all be found in the Rocq files. Once we have mechanized proofs we can step through and experiment with, giving detailed proofs on paper is not helpful

There are weekly assignment sheets (14 in total) accompanying the course.

The notes should be useful for teachers and students working with MPCTT.

Contents

1 Inductive Definition of Numbers 2

2 Iteration, Dependent Function Types, and Pairs 3

3 Proposition as Types 5

4 Conjunction and Disjunction 7

1 Inductive Definition of Numbers

Numbers $0, 1, 2, \ldots$ can be defined in computational type theory. They are predefined in Rocq.

- Inductive definition of numbers introduces three typed constructors:
  $\mathsf{N} : \mathbb{T},\ 0 : \mathsf{N},\ \mathsf{S} : \mathsf{N} \to \mathsf{N}$
  $\mathbb{T}$ is the type of all types. We assume $\mathbb{T} : \mathbb{T}$ (constraints apply).
- Definition of addition $x + y$ and multiplication $x \cdot y$ as functions $\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ with defining equations discriminating on first argument. Exactly one equation per constructor (0 and S). Recursion must be terminating. We shall only use structural recursion. We speak of inductive function definitions.
- Inductive definition of numbers and their operations go back to Dedekind and Peano.
- Subtraction is defined as truncating (e.g., $3-4=0$). Definition is by discrimination on first argument and a secondary discrimination on second argument in the successor case.
- The inductive definitions of numbers, addition and subtraction we are using also underly their definition in Rocq:

$$\mathsf{N} : \mathbb{T} \ ::= \ 0 \mid \mathsf{S}(\mathsf{N})$$

$$\begin{aligned} + &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ 0 + y &:= y \\ \mathsf{S}x + y &:= \mathsf{S}(x + y) \end{aligned} \qquad \begin{aligned} - &: \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ 0 - y &:= 0 \\ \mathsf{S}x - 0 &:= \mathsf{S}x \\ \mathsf{S}x - \mathsf{S}y &:= x - y \end{aligned}$$

- Order can be defined with truncating subtraction: $x \le y \ := \ (x - y = 0)$
- A functions for minimum can be defined following the discrimination scheme of subtraction.
- In Rocq defining equations must be expressed with a single equation and a match construct familiar from ML-style functional programming. Not nice from a user's perspective. Recursion must be announced by writing Fixpoint in place of Definition. Rocq's type checker enforces a liberal form of structural recursion. We will only use strict structural recursion.
- Left-to-right rewriting with defining equations is terminating and yields unique normal forms. We say that an equation holds by computational equality if both sides have the same normal form. Rocq's tactic (i.e., proof command) for computational equality is reflexivity.

Proof by induction

- A (proof) goal consists of $n \ge 0$ assumptions and a claim. The assumptions are typed variables, and the claim is a proposition (a logical statement). Propositions are syntactic objects and are subject to type checking like everything else. A proof step reduces a goal G to $n \ge 0$ so-called subgoals. If $n = 0$, the proof step proves the goal G. If $n \ge 1$, proofs of the subgoals yield a proof of the goal G.
- The inductive definition of numbers yields the induction rule
  $\mathsf{I}_\mathsf{N} : \ \forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to (\forall n.\, pn \to p(\mathsf{S}n)) \to \forall n.pn$

$\mathbb{P}$ is the type of propositions. p is a predicate $\mathsf{N} \to \mathbb{P}$ on numbers. If we apply the induction rule to a goal with the claim $\forall n.pn$, we get two subgoals, one for $p0$ and one for the universally quantified implication $\forall n.\ pn \to p(\mathsf{S}n)$.

- The functional interpretation of the induction rule says that I_N is a function taking a predicate p, a proof of $p0$, and a function $\forall n.\ pn \to p(\mathsf{S}n)$ as arguments and returning a proof $\forall n.pn$. Moreover, a function $\forall n.\ pn \to p(\mathsf{S}n)$ takes a number n and a proof of pn as arguments and returns a proof of $p(\mathsf{S}n)$.
- The function $\forall n.\ pn \to p(\mathsf{S}n)$ represents the so-called inductive step, and the argument pn the so-called inductive hypothesis.
- The induction rule also provides for discrimination (case analysis)
  $\forall p^{\mathsf{N}\to\mathbb{P}}.\ p0 \to (\forall n.\ p(\mathsf{S}n)) \to \forall n.pn$
  on the constructor of a number, one just ignores the inductive hypothesis.
- Universal quantifications and implications in the claim of a goal can be made assumptions of the goal (tactic intros). Conversely, assumptions can be moved into the claim (tactic revert).
- While $0 + y = y$ follows by computational equality, $x + 0 = x$ does not. An inductive proof for $x + 0 = x$ works. The inductive step rewrites with the inductive hypothesis.
- Proof of commutativity $x + y = y + x$ with induction uses rewriting with lemmas for $x + 0 = x$ and $x + \mathsf{S}y = \mathsf{S}(x + y)$. We will mostly use the command Fact to establish lemmas in Rocq. Unnamed facts can be stated and proved with the command Goal.
- $x - x = 0$ follows by induction on x and discrimination in the successor case.
- $(x + y) - x = y$ follows by induction on x and discrimination on y in the zero case.
- $(x + y) - x = y$ follows from $(x + y) - x = y$ and commutativity of $+$.
- $\min xy = \min yx$ can be shown by induction on x with y is quantified in inductive hypothesis (induction rule applied to $\forall x\, \forall y.\ \min xy = \min yx$ rather than to $\forall x.\ \min xy = \min yx$). Commutativity of maximum and distance are similar.
- Do Booleans and their operations as exercise.
- Do all proofs with Rocq to understand them in detail. Exploring a proof with Rocq is more helpful than doing it on paper.
- Rocq comes with arithmetic provers that can prove facts like commutativity of addition and multiplication and the subtraction laws automatically. They provers are provided through the tactics lia and nia.

Explore Chapter 1 of MPCTT and `ln_gs.v`.

2 Iteration, Dependent Function Types, and Pairs

- We use the notation $f^n x$ to say that f is applied n-times to x; for instance, $f^3 x = f(f(fx))$ and $f^0 x = x$; we speak of iteration.
- We have $n = \mathsf{S}^n 0$, $n + y = \mathsf{S}^n y$, $n \cdot y = (+y)^n 0$.
- Iteration can be defined recursively: $f^0 x := x$, $f^{\mathsf{S}n} = f(f^n x)$
- Iteration can be modeled with a family of functions $\mathsf{iter}_X : (X \to X) \to \mathsf{N} \to X \to X$

where X ranges over all types. The iterations in the examples above use $\mathsf{iter}_{\mathsf{N}}$.

- Computational type theory accommodates the type argument X of iter with a dependent function type $\mathsf{iter} : \forall X^{\mathbb{T}}.\ (X \to X) \to \mathsf{N} \to X \to X$.
  For instance, $\mathsf{iter}_{\mathsf{N}} : (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \to \mathsf{N}$.
- The full inductive definition of iter we are using looks as follows:

$$\begin{aligned} &\mathsf{iter} : \forall X^{\mathbb{T}}.\ (X \to X) \to \mathsf{N} \to X \to X \\ &\quad \mathsf{iter}\, X\, f\, 0\, x := x \\ &\mathsf{iter}\, X\, f\, (\mathsf{S}n)\, x := f(\mathsf{iter}\, X\, f\, n\, x) \end{aligned}$$

- Simple function types $s \to t$ are dependent function types $\forall x^s.\ t$ where the variable x does not occur in t. Rocq displays a dependent function type $\forall x^s.\ t$ as $s \to t$ whenever x does not occur in t.
- In Rocq, an argument may be written as an underline "_". Rocq will try to infer underline arguments. For instance, iter _ S will elaborate to iter N S while iter _ by itself will fail to elaborate.
- In Rocq, arguments can be declared as implicit arguments. During parsing, implicit arguments are put in as underlines so that elaboration can derive them. With the first argument of iter declared implicit, we can write $n + y = \mathsf{iter}\,\mathsf{S}\,n y$.
- The equation $n + y = \mathsf{iter}\,\mathsf{S}\,n y$ has a straightforward inductive proof. So has the shift law $\mathsf{iter}\, f (\mathsf{S}n) x = \mathsf{iter}\, f n (\mathsf{S}x)$.

Pairs and pair types

- Pairs and pair types (similar to $(x, y) \in X \times Y$ with sets) can be obtained with an inductive definition

$$\mathsf{Pair}(X : \mathbb{T}, Y : \mathbb{T}) : \mathbb{T} \ ::= \ \mathsf{pair}(X, Y)$$

  introducing a type constructor and a value constructor

$$\begin{aligned} &\mathsf{Pair} : \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ &\mathsf{pair} : \forall XY^{\mathbb{T}}.\ X \to Y \to \mathsf{pair}\, XY \end{aligned}$$

- We shall use the familiar notations

$$\begin{aligned} X \times Y \quad &\rightsquigarrow \quad \mathsf{Pair}\, XY \\ (x, y) \quad &\rightsquigarrow \quad \mathsf{pair}\, \_\, \_\, xy \end{aligned}$$

- Projection functions for pairs can be defined inductively with the equations

$$\begin{aligned} &\pi_1 : \forall XY^{\mathbb{T}}.\ X \times Y \to X && \pi_2 : \forall XY^{\mathbb{T}}.\ X \times Y \to Y \\ &\pi_1\, XY(x, y) := x && \pi_2\, XY(x, y) := y \end{aligned}$$

- The discrimination rule for pairs coming with their inductive definition is

$$\forall XY^{\mathbb{T}}\, \forall p^{X \times Y \to \mathbb{P}}.\ (\forall xy.\ p(x, y)) \to \forall a.\, pa$$

  Applying the discrimination rule for pair to the claim of a goal $\forall a.\, pa$ has the effect that the claim is replaced by $\forall xy.\ p(x, y)$ The so-called eta law $\forall a^{X \times Y}.\ (\pi_1 a, \pi_2 a) = a$ follows with the discrimination rule and computational equality.

Fibonacci sequence and plain function definitions

- The Fibonacci sequence $0, 1, 1, 2, 3, 5, \ldots$ can be computed by iterating a step function σ on pairs:

$$\sigma : \mathsf{N} \times \mathsf{N} \to \mathsf{N} \times \mathsf{N} \qquad \mathsf{fib} : \mathsf{N} \to \mathsf{N}$$
$$\sigma\, a := (\pi_2 a,\ \pi_1 a + \pi_2 a) \qquad \mathsf{fib}\, n := \pi_1(\mathsf{iter}\, \sigma\, n\, (0, 1))$$

  The functions σ and fib are defined as a so-called plain functions. Plain functions must not discriminate on inductive arguments and must not be recursive. The defining equations of plain functions are used for reduction and contribute to computational equality, similar to the defining equations of inductive functions. The function application $\mathsf{fib}\, n$ computes the nth Fibonacci number. The equation $\mathsf{fib}(\mathsf{S}(\mathsf{S}n)) = \mathsf{fib}\, n + \mathsf{fib}(\mathsf{S}n)$ follows by computational equality.

Explore Chapter 1 of MPCTT and `ln_gs.v`.

3 Proposition as Types

- We take the view that a proof of an implication $s \to t$ is a function mapping a proof of the proposition s to a proof of the proposition t.
- Moreover, we see a proof of a universal quantification $\forall x^u.\ s$ as a function mapping an element a of the type u to a proof of the proposition s where the value a replaces the variable x.
- Following this functional interpretation, computational type theory represents implications $s \to t$ and universal quantifications $\forall x^u.\ s$ as function types. More generally, all propositions are represented as types. One speaks of the propositions as types approach.
- To distinguish computational types (e.g. N) from propositional types, computational type theory has a second universe $\mathbb{P}$ of propositional types.
- Given a propositional type u, a proof term for u is a term whose type is u.
- It turns out that all propositions we want to consider can be represented as propositional types, and that proofs of propositions can always be represented as proof terms. To simplify our language, we may refer propositional types as propositions and to proof terms as proofs.
- With the propositions as types approach, proof checking amounts to type checking. There are type checking algorithms that work well in practice as one can see with the proof assistant Rocq.
- Here are propositional types that have simple proof terms:

$$\forall X^{\mathbb{P}}.\ X \to X$$
$$\forall XYZ^{\mathbb{P}}.\ (X \to Y) \to (Y \to Z) \to (X \to Z)$$
$$\forall XYZ^{\mathbb{P}}.\ (X \to Y \to Z) \to (X \to Y) \to (X \to Z)$$

  The proof terms are easily obtained as plain functions defined for the given types.

Lambda Terms

- A lambda term $\lambda x^u.s$ describes a function taking an argument x of type u and returning the value described by the term s. For instance, $\lambda x^{\mathsf{N}}.\, x - 5$ describes a function subtracting 5 from its argument. To ease our language, we may refer to lambda terms jus as lambdas.
- Using lambdas, we can write proof terms for the above propositional types not using defined functions:

$$\lambda X^{\mathbb{P}}\, \lambda x^X.\, x$$
$$\lambda XYZ^{\mathbb{P}}\, \lambda f^{X\to Y}\, \lambda g^{Y\to Z}\, \lambda x^X.\, g(fx)$$
$$\lambda XYZ^{\mathbb{P}}\, \lambda f^{X\to Y\to Z}\, \lambda g^{X\to Y}\, \lambda x^X.\, fx(gx)$$

  The above lambdas are nested and omit the dots where we find it convenient. Given the propositions from above as claims, we can omit the variable types since they can be inferred from the claims:

$$\begin{aligned}
&\lambda Xx.\, x && : \forall X^{\mathbb{P}}.\, X \to X\\
&\lambda XYZfgx.\, g(fx) && : \forall XYZ^{\mathbb{P}}.\, (X \to Y) \to (Y \to Z) \to (X \to Z)\\
&\lambda XYZfgx.\, fx(gx) && : \forall XYZ^{\mathbb{P}}.\, (X \to Y \to Z) \to (X \to Y) \to (X \to Z)
\end{aligned}$$

  This way we get a compact notation for proof terms. It is best explored with the proof assistant.
- Proof term construction is best done with the proof assistant, which provides type inference and type checking in addition to recording the types of the variables (as assumptions) and the claims still to prove. If one constructs a proof by hand, one may follow a top down stretey a use a proof table (see MPCTT) to record the assumptions and claims. With Rocq, one may use the commands Definition, Fact and Goal together with the tactics intros, exact, and refine.

Falsity and Negation

- To support the propositions as types approach, computational type theory is designed such that there are empty propositional types. Given this design, we can pick the propositional type

$$\bot : \mathbb{P} \;:=\; \forall X^{\mathbb{P}}.X$$

  as a proposition that has no proof. From the logical perspective, $\bot$ represents falsity. Expressing falsity as a function immediately gives us a logical principle known as explosion rule or as ex falso quodlibet: From a (hypothetical) proof of falsity, we can get a proof of everything.
- With falsity in place, we can express negation:

$$\neg s \quad \rightsquigarrow \quad s \to \bot$$

  With this representation, a proof of a negation $\neg s$ is a function mapping a proof of s to a proof of falsity. Thus a proof of a negation $\neg s$ says that the proposition s has no proof.

- We say that a proposition s is provable if it has a proof term. Moreover, we say that a proposition s is disprovable if its negation $\neg s$ is provable. As it comes to intuition, provable propositions are true and disprovable propositions are false.
- Here are propositional types with negations that have proof terms:

$$X \to \neg X \to \bot$$
$$X \to \neg\neg X$$
$$X \to \neg X \to Y$$
$$(X \to Y) \to (\neg Y \to \neg X)$$
$$(X \to \neg X) \to (\neg X \to X) \to \bot$$

  The first two propositions are identical up to notation, so they have the same proof terms. The proof of the third proposition uses the explosion rule. We describe the proof of the final proposition.

  > Assume $f : X \to \neg X$ and $g : \neg X \to X$. We prove $\bot$. From f we obtain $h : \neg X$. Now $h(g(h))$ proves the claim.

  The described proof translates into a proof term using a let term:

$$\lambda f g.\ \text{LET}\ h : \neg X\ =\ \lambda x.\ f x x\ \ \text{IN}\ \ h(gh)$$

Explore Chapter 3 of MPCTT and `ln_pat.v`.

4 Conjunction and Disjunction

Conjunction

- In the propositions as types approach we need three constants for conjunctions:

$\wedge\ :\ \mathbb{P} \to \mathbb{P} \to \mathbb{P}$	formation
$\mathsf{C}\ :\ \forall XY^{\mathbb{P}}.\ X \to Y \to X \wedge Y$	introduction
$\mathsf{E}_\wedge\ :\ \forall XY^{\mathbb{P}}.\ X \wedge Y \to \forall Z^{\mathbb{P}}.\ (X \to Y \to Z) \to Z$	elimination

- With the formation constant we can write conjunctions $X \wedge Y$.
- With the introduction constant we can prove a conjunction $X \wedge Y$ by proving X and Y.
- The elimination constant says that given a proof of $X \wedge Y$, we can prove a proposition Z by proving the implication $X \to Y \to Z$.
- Assuming the constants for conjunction, we can prove
  - $X \wedge Y \to Y \wedge X$ saying that conjunction is commutative.
  - the converse of the elimination rule: $\forall XY^{\mathbb{P}}.\ (\forall Z^{\mathbb{P}}.\ (X \to Y \to Z) \to Z) \to X \wedge Y$.
- The last fact tells us that we can define the constants for conjunction as plain constants or plain functions:

$\wedge\ :\ \mathbb{P} \to \mathbb{P} \to \mathbb{P}$	$:=\ \lambda XY^{\mathbb{P}}.\ \forall Z^{\mathbb{P}}.\ (X \to Y \to Z) \to Z$
$\mathsf{C}\ :\ \forall XY^{\mathbb{P}}.\ X \to Y \to X \wedge Y$	$:=\ \lambda XYxy.\ \lambda Zf.\ fxy$
$\mathsf{E}_\wedge\ :\ \forall XY^{\mathbb{P}}.\ X \wedge Y \to \forall Z^{\mathbb{P}}.\ (X \to Y \to Z) \to Z$	$:=\ \lambda XYa.a$

Disjunction

- In the propositions as types approach we need four constants for disjunctions:

$$\begin{array}{rll}
\vee & : \mathbb{P} \to \mathbb{P} \to \mathbb{P} & \text{formation} \\
\mathsf{L} & : \forall XY^{\mathbb{P}}.\ X \to X \vee Y & \text{introduction left} \\
\mathsf{R} & : \forall XY^{\mathbb{P}}.\ Y \to X \vee Y & \text{introduction right} \\
\mathsf{E}_{\vee} & : \forall XY^{\mathbb{P}}.\ X \vee Y \to \forall Z^{\mathbb{P}}(X \to Z) \to (Y \to Z) \to Z & \text{elimination}
\end{array}$$

- With the formation constant we can write disjunctions $X \vee Y$.
- With the introduction constants we can prove $X \vee Y$ by proving either X or Y.
- The elimination constant says that given a proof of $X \vee Y$, we can prove a proposition Z by proving the implications $X \to Z$ and $Y \to Z$.
- Assuming the constants for Disjunction, we can prove
  - $X \vee Y \to Y \vee X$ saying that disjunction is commutative.
  - the converse of the elimination rule:
    $\forall XY^{\mathbb{P}}.\ (\forall Z^{\mathbb{P}}.\ ((X \to Z) \to (Y \to Z) \to Z) \to X \vee Y$.
- The last fact tells us that we can define the constants for disjunction as plain constants or plain functions:

$$\begin{array}{rll}
\vee & : \mathbb{P} \to \mathbb{P} \to \mathbb{P} & := \lambda XY^{\mathbb{P}}.\ \forall Z^{\mathbb{P}}.\ (X \to Z) \to (Y \to Z) \to Z \\
\mathsf{L} & : \forall XY^{\mathbb{P}}.\ X \to X \vee Y & := \lambda XYx.\lambda Zfg.\ fx \\
\mathsf{R} & : \forall XY^{\mathbb{P}}.\ Y \to X \vee Y & := \lambda XYy.\lambda Zfg.\ gy \\
\mathsf{E}_{\vee} & : \forall XY^{\mathbb{P}}.\ X \vee Y \to \forall Z^{\mathbb{P}}.\ (X \to Y \to Z) \to Z & := \lambda XYa.a
\end{array}$$

Inductive definition of conjunction and disjunction

Computational type theory admits inductive definitions in the propositional universe $\mathbb{P}$. We can write inductive definitions providing the formation and the introduction constants for conjunction and disjunction:

$$\begin{array}{rcl}
\wedge(X : \mathbb{P}, Y : \mathbb{P}) & ::= & \mathsf{C}(X, Y) \\
\vee(X : \mathbb{P}, Y : \mathbb{P}) & ::= & \mathsf{L}(X) \mid \mathsf{R}(Y)
\end{array}$$

Note that inductive definition of conjunctions in $\mathbb{P}$ mirrors the inductive definition of pairs in $\mathbb{T}$. The eliminators for conjunctions and disjunctions can now easily be obtained as inductive functions:

$$\begin{array}{rcl}
\mathsf{E}_{\wedge} & : & \forall XY^{\mathbb{P}}.\ X \wedge Y \to \forall Z^{\mathbb{P}}.\ (X \to Y \to Z) \to Z \\
\mathsf{E}_{\wedge}\, XY\, (\mathsf{C}__xy)\, Zf & := & fxy
\end{array}$$

$$\begin{array}{rcl}
\mathsf{E}_{\vee} & : & \forall XY^{\mathbb{P}}.\ X \vee Y \to \forall Z^{\mathbb{P}}.\ (X \to Y \to Z) \to Z \\
\mathsf{E}_{\vee}\, XY\, (\mathsf{L}__x)\, Zfg & := & fx \\
\mathsf{E}_{\vee}\, XY\, (\mathsf{R}__y)\, Zfg & := & gy
\end{array}$$

We may refer to the value constructors of inductive type definitions in $\mathbb{P}$ as proof constructors. For instance, the proposition constructor $\vee$ has two proof constructors L and R.

We can also give inductive definitions of the formation and elimination constants for falsity:

$$\bot : \mathbb{P} \ ::= \ []$$
$$\mathsf{E}_\bot \ : \ \bot \to \forall Z^{\mathbb{P}}.\, Z$$

The trick is to not have a proof constructor for $\bot$. This way inductive functions taking a argument of type $\bot$ can be established without defining equations (the general rule says there must be one equation for every constructor). In the logical literature one speaks of vacuous truth.

We remark that Rocq uses inductive definitions of falsity, conjunction, and disjunction.

We express propositional equivalence with a notation:

$$s \longleftrightarrow t \quad \rightsquigarrow \quad (s \to t) \wedge (t \to s)$$

A proof of an equivalence $s \longleftrightarrow t$ provides two functions translating between proofs of s and t.

The following equivalences are provable for all propositions X and Y:

$$\begin{aligned} X \wedge Y &\longleftrightarrow \forall Z^{\mathbb{P}}.\, (X \to Y \to Z) \to Z \\ X \vee Y &\longleftrightarrow \forall Z^{\mathbb{P}}.\, (X \to Z) \to (Y \to Z) \to Z \\ \bot &\longleftrightarrow \forall Z^{\mathbb{P}}.\, Z \end{aligned}$$

They are know as functional or impredicative characterizations of the logical constants $\wedge$, $\vee$, and $\bot$.

Here are more provable equivalences:

$$\begin{aligned} \neg(X \vee Y) &\longleftrightarrow \neg X \wedge \neg Y \\ (X \to Y \to Z) &\longleftrightarrow (X \wedge Y \to Z) \end{aligned}$$

Explore Chapter 3 of MPCTT and `ln_pat.v`.