ilyasergey.net

NUS Workshop on Formal Proofs and Lean 22 April 2024

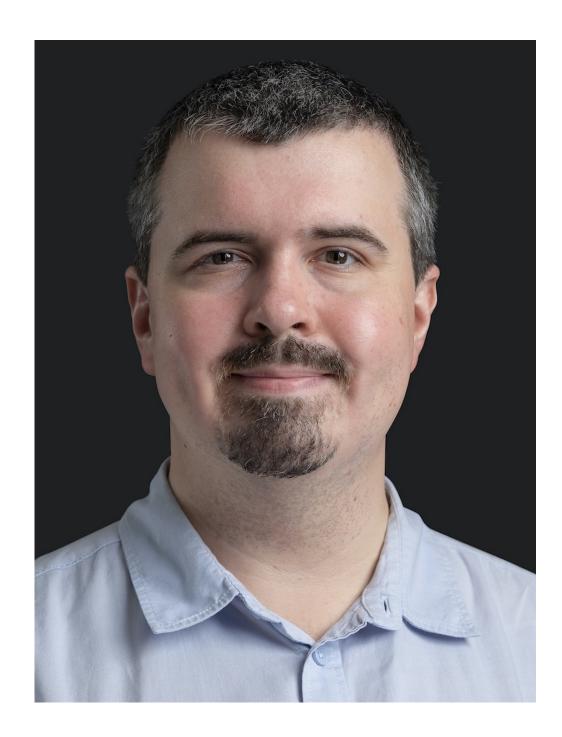
# Proving as Programming

### Ilya Sergey

# About myself: Ilya Sergey

- Associate Professor at NUS School of Computing
  - Running VERSE lab (<u>verse-lab.github.io</u>)
- Previously: Assistant Professor at UCL (2015-2018)
- Research Interests:
  - Design of Programming Languages
  - Computer-Aided Software Verification
  - Program Synthesis
- Using proof assistants since 2012
  - Wrote a textbook on Coq/SSreflect (<u>ilyasergey.net/pnp</u>)







slides of this talk

based on 2018/19 lectures by Xavier Leroy at Collège de France xavierleroy.org/CdF/2018-2019/

Proving as Programming Or

The Curry-Howard correspondence: a link between Computer Science and Logic



### Computer science and mathematical logic

from mathematical logic.

logicians or had beed trained in logic.)

This talk outlines the history of seminal interaction between logic and computer science, and more precisely between proof theory and programming languages.

- Since its inception, computer science has taken many ideas and many principles
- (Church, Turing, von Neumann, and many other founders of computer science were

# Programming with logic

- Many programming languages treat programs as formulas of a logic:
  - Logic programming (Prolog, Mercury, ...)
  - Constraint programming (Prolog III, CHIP, Oz, ...)
  - Queries in relational databases.

Executing the program amounts to logical formula.

Executing the program amounts to proving or refuting the corresponding

# An example in Prolog

A Prolog program:

append([], L, L).
append([H|T], L, [H|M]

append([H|T], L, [H|M]) :- append(T, L, M).
?- append(X, Y, [1,2,3]).

These Horn clauses and the final query define a logical formula:

 $(\forall L, append([], L, L)) \\ \land \quad (\forall H, T, L, M, append(T, L, M) \Rightarrow append([H|T], L, [H|M])) \\ \Rightarrow \quad \exists X, Y, append(X, Y, [1, 2, 3])$ 

Executing the program amounts to proving this proposition by finding appropriate X, Y that satisfy the  $\exists$ .

### Program = Proposition

Logic programming falls within a rather natural correspondence...

programming language mathematical logic proposition program proof execution

This is **not** the Curry-Howard correspondence.

# The Curry-Howard correspondence

An isomorphism between an intuitionistic (constructive) logic and the simply-typed  $\lambda$ -calculus (the core of a functional programming language).

simply-typed  $\lambda$ -calc

type

term (programme)

Complete change of viewpoint: programs are no longer **propositions**, but the **proofs** of propositions.



culus	intuitionistic logic
	proposition
)	proof

## The Curry-Howard correspondence

# summarized by the "PAT principle":

- Inspires a new perspective on logics and on programming languages,
  - **Propositions As Types** 
    - Proofs As Terms

### Curry-Howard today

A guiding principle to design, understand, and formalize programming languages (mostly, but not only, functional languages, such as OCaml and Haskell).

New ways to program, integrating formal verification (e.g., dependent types).

New ways to do mathematics, leveraging the power of the computer.

Powerful and versatile tools such as Coq and Lean, to help us explore this border between computer science and mathematics.

### Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and **Quantifying over Values**
- Curry-Howard for Practitioners: Program Synthesis and Proof Repair

### Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

### The lambda-notation

Circa 1930, Alonzo Church introduces the notation  $\lambda x$ . e meaning "the function that maps parameter x to expression e".

Today we write  $x \mapsto e$ . However, this notation, popularized by Bourbaki, was introduced later.

These notations  $\lambda x$ . e or  $x \mapsto e$  are useful to clarify the way mathematicians talk about functions...

- " the function  $x^2$  "
- "the function  $x^2 + y^2$ "
- " the family of functions cx<sup>2</sup>"

$$\begin{array}{l} x\mapsto x^2 \text{ or } \lambda x. \, x^2 \\ (x,y)\mapsto x^2+y^2 \text{ or } \lambda x. \lambda y. \, x^2+y^2 \\ \textbf{'} \quad c\mapsto (x\mapsto cx^2) \text{ or } \lambda c. \lambda x. \, cx^2 \end{array}$$

### Computing with the lambda-notation

Two main computation rules:

 $\alpha$ -conversion: renaming a bound variable.

function body.

 $(\lambda x. M)(N)$ 

In ordinary mathematics we write: Consider the function f(x) = x + 1. We have that f(2) = 3.

 $f(2) = (\lambda x.x+1)(2) =_{\beta} 2+1 = 3$ 

- $\lambda x. M =_{\alpha} \lambda y. M\{x \leftarrow y\}$  if y not free in M
- $\beta$ -conversion: replacing the formal parameter by the actual argument in a

$$) =_{\beta} M\{x \leftarrow N\}$$

- The lambda-notation decomposes this computation in multiple steps:

Why lambda?

Dana Scott tells that he once wrote Church Dear professor Church: why "lambda"? and received the following reply Eeny, meeny, miny, moe.

Rosser tells that Church progressively simplified the notation  $\hat{x}e$  that is used in Russell and Whitehead's *Principia Mathematica*:

$$\hat{x}e \rightarrow \wedge xe - \hat{x}e$$

 $\ \ \, \lambda \mathbf{x} \mathbf{e} \quad \rightarrow \quad \lambda \mathbf{x} [\mathbf{e}] \quad \rightarrow \quad \lambda \mathbf{x} . \mathbf{e}$ 

### A lambda to bind them all

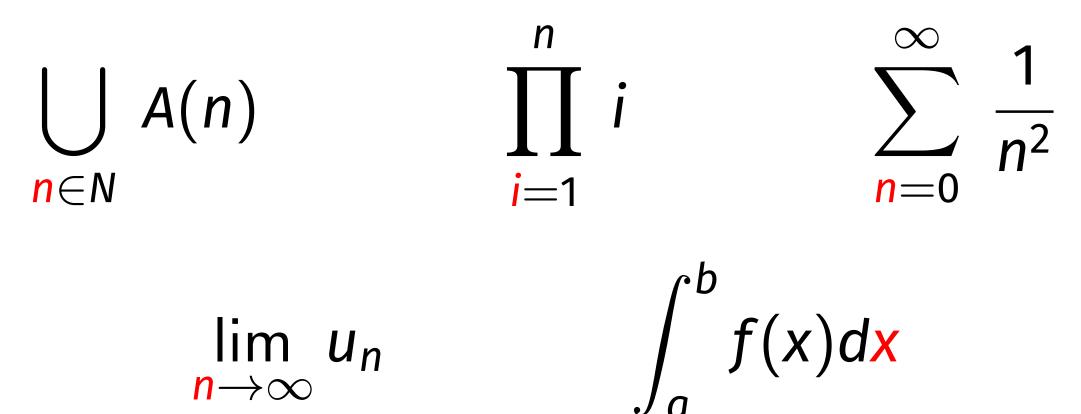
changing the meaning.

Bound variables appear in various mathematical notations:



### In $\lambda x.e$ , variable x is said to be "bound" and can be renamed without

 $\{\mathbf{X} \mid P(\mathbf{X})\} \qquad \forall \mathbf{X}, P(\mathbf{X}) \qquad \exists \mathbf{X}, P(\mathbf{X})$ 



Ja

### The lambda-calculus

- $\alpha$ -equivalence and  $\beta$ -reduction (= oriented conversion):

$$\lambda x.M =_{\alpha} \lambda y.M \{ x \leftarrow y \}$$
(y not free in M)

"normal form" N is reached.

$$M \rightarrow_{\beta} M_1 \rightarrow_{\beta} M_2 \rightarrow_{\beta} \cdots \rightarrow_{\beta} N \not\rightarrow_{\beta}$$

A minimalistic formal system to express algorithms as compositions of functions

 $M, N ::= x \mid \lambda x. M \mid M N$ 

$$(\lambda x.M) N \rightarrow_{\beta} M\{x \leftarrow N\}$$

A notion of computation appears: iterate  $\beta$ -reduction until an irreducible

# Lambda-calculus and computability

Church (1936) shows the first undecidability results (= cannot be computed by a general recursive function):

- Whether a term M has a normal form is undecidable.
- Corollary: Hilbert's decision problem (Entscheidungsproblem) has no solution.

Kleene (1935) and Turing (1937) show equivalences between

- general recursive functions (Herbrand, Gödel, Kleene);
- functions computable by a Turing machine;
- functions definable as a  $\lambda$ -term via Church's encoding of natural numbers (1933):

$$n \equiv \lambda f. \underbrace{f \circ \cdots \circ f}_{n \text{ times}} \equiv \lambda f. \lambda x. \underbrace{f (f (\cdots (f x)))}_{n \text{ times}}$$

# Lambda-calculus and functional languages

The pure lambda-calculus as the common ancestor and as the kernel of functional programming languages:

LISP 1.5 (1960) refers to Church and writes (LAMBDA ...) for anonymous functions. However, its semantics (dynamic scoping of bindings) does not follow  $\beta$ -reduction.

Landin (1965) proposes ISWIM, an extended lambda-calculus, as the basis for The next 700 programming languages.

Common Lisp, Scheme, SML, Caml, Haskell, ...

- functional language = pure lambda-calculus
  - reduction strategy ╋
  - data types +
  - type system (if applicable) +

- After 1970, all functional languages contain lambda-calculus as a kernel:

### Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

### Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

# Intuitionism and intuitionistic logic

Requires all mathematical objects to be accessible to intuition. Rejects non-constructive proofs.

**Intuitionistic logic:** family of logics studied by Heyting, Glivenko, Gödel, Kolmogorov. Formalizes the "only constructive proofs" aspect of intuitionism.

**Intuitionism:** a philosophy of mathematics developed by Brouwer (1881–1966).

### Constructive proofs vs. non-constructive proofs

- How to prove "there exists  $x \in A$  such that P(x)"?
  - Constructively: we define an element a of A from the
  - 2 By contradiction: we assume  $\exists x \in A$ . P(x) false, that is, we assume  $\forall x \in A. \neg P(x)$  true, and we deduce a contradiction.
- For an intuitionist, the first proof is the only valid way to show that  $\exists x \in A. P(x) \text{ is true.}$

The second proof only shows that  $\exists x \in A$ . P(x) is not absurd, which is a weaker result.

### An intuitionistic adventure (inspired by G. Dowek, Computation, Proof, Machine)

stop before the French border. Note: the train may make extra unplanned stops.

At which stop should I get off the train?

this is "my" stop.

criterion to decide at each stop whether to get off or stay...



- In the train back from the Netherlands, someone drops me a note:
  - The French classicist logicians want to kill you. Get off at the last
- The set A of all stops outside France is 1- nonempty (Brussels  $\in$  A), 2- totally ordered, and 3- finite; hence, it contains a maximal element;
- However, this proof is not constructive and fails to give me an effective

### Excluded middle

- The excluded middle principle (tertium non datur, tiers exclu):  $P \lor \neg P$  for all proposition P
- Equivalent to double negation elimination:
  - $\neg \neg P \Rightarrow P$  for all proposition P
- This is the principle for proofs by contradiction.

Intuitionistic logic rejects excluded middle (and the many equivalent rules).

### Truth vs provability

### **Classical logic:**

every proposition is *a priori* true or false. Mathematical proof establishes which of these two cases apply.

### Intuitionistic logic:

a proposition can be proved; or its negation can be proved; otherwise we do not know anything about the proposition.

### The BHK interpretation (Brouwer, Heyting, Kolmogorov)

Provability is characterized by the existence of a **construction** of the proposition.

- There exists a construction of  $\top$  ("true").
- There exists no construction of  $\perp$  ("false").
- A construction of  $P_1 \wedge P_2$  is a pair  $(c_1, c_2)$ where  $c_1$  is a construction of  $P_1$  and  $c_2$  a construction of  $P_2$ .
- A construction of  $P_1 \vee P_2$  is a pair (i, c)where  $i \in \{1, 2\}$  and c is a construction of  $P_i$ .

### The BHK interpretation (Brouwer, Heyting, Kolmogorov)

- A construction of  $P_1 \Rightarrow P_2$  is a "process" that, given a construction of  $P_1$ , produces a construction of  $P_2$ .
- A construction of  $\neg P$ , treated like  $P \Rightarrow \bot$ , is an algorithm that, from a (hypothetical) construction of P produces a non-existent object.
- A construction of  $\forall x, P(x)$  is an algorithm that, given a value for x, produces a construction of P(x).
- A construction of  $\exists x, P(x)$  is a pair (*a*, construction of P(a)).

Provability is characterized by the existence of a **construction** of the proposition.

### Back to excluded middle

Let TM be the set of all Turing machines. Consider

$$Q \stackrel{def}{=} \forall m \in TM$$
, termi

Q follows immediately from excluded middle ( $P \lor \neg P$  for all P).

Yet, a construction of Q would be an algorithm that, given a Turing machine *m*, returns (1, c) if *m* terminates and (2, c) otherwise.

Since the halting problem is undecidable, such an algorithm does not exist.

Excluded middle is therefore not constructive, since there is no construction that validates it.

(This construction would be a universal decision procedure!)

 $inates(m) \lor \neg terminates(m)$ 

### A formalization of intuitionistic logic

Manipulates intuitionistic sequents of the form  $\Gamma \vdash P$ (read: "under hypotheses Γ we can deduce P".)

Comprises axioms ("this sequent is true") and inference rules ("if the sequents above are true, then the sequent below is true"). For example:

D

$$\begin{array}{ccc} \Gamma \vdash P & \Gamma \vdash Q \\ \hline \Gamma \vdash P \land Q \end{array}$$

### Rules of intuitionistic logic

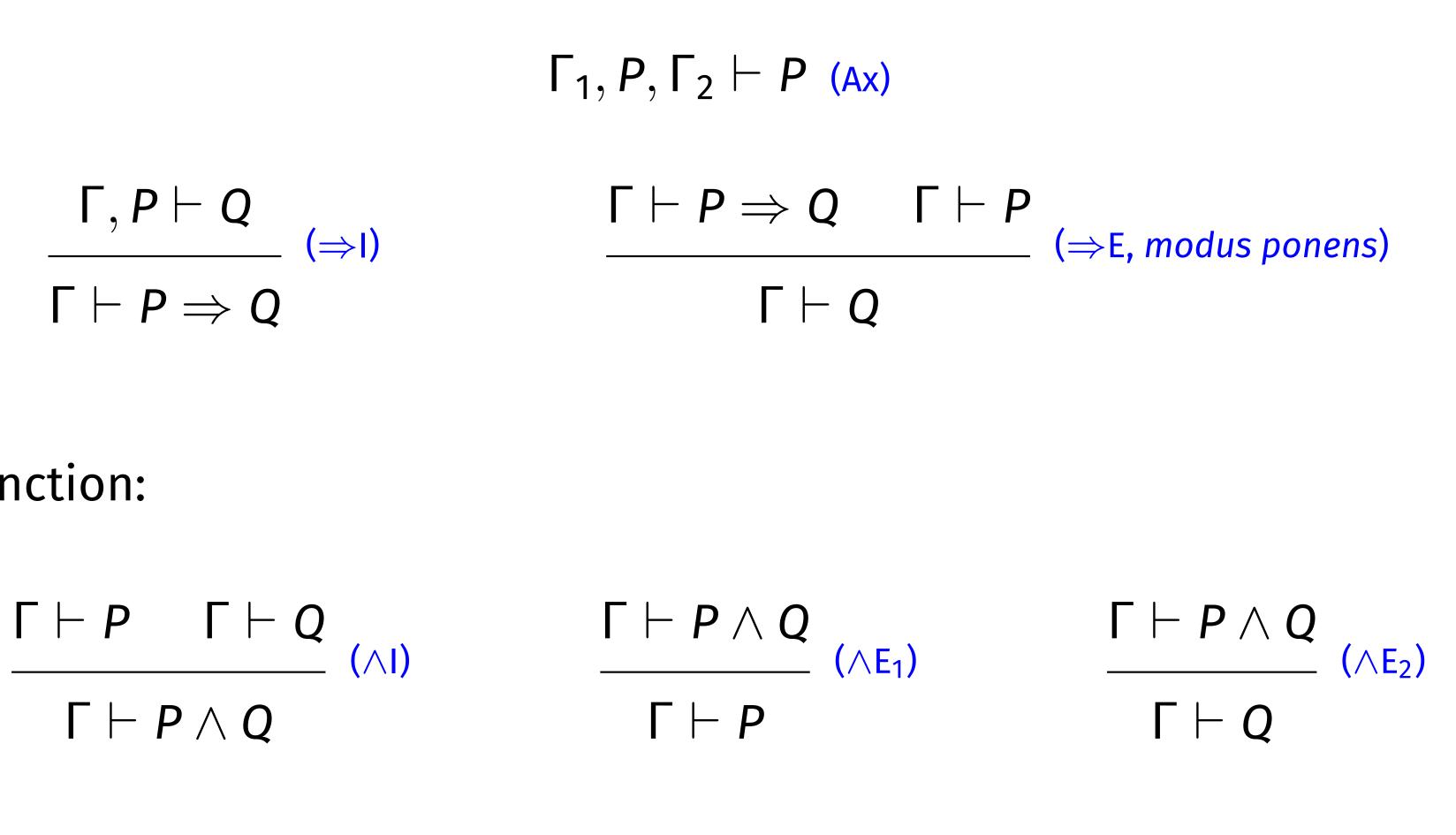
Implication:

 $\frac{\Gamma, P \vdash Q}{(\Rightarrow I)}$  $\Gamma \vdash P \Rightarrow Q$ 

Conjunction:

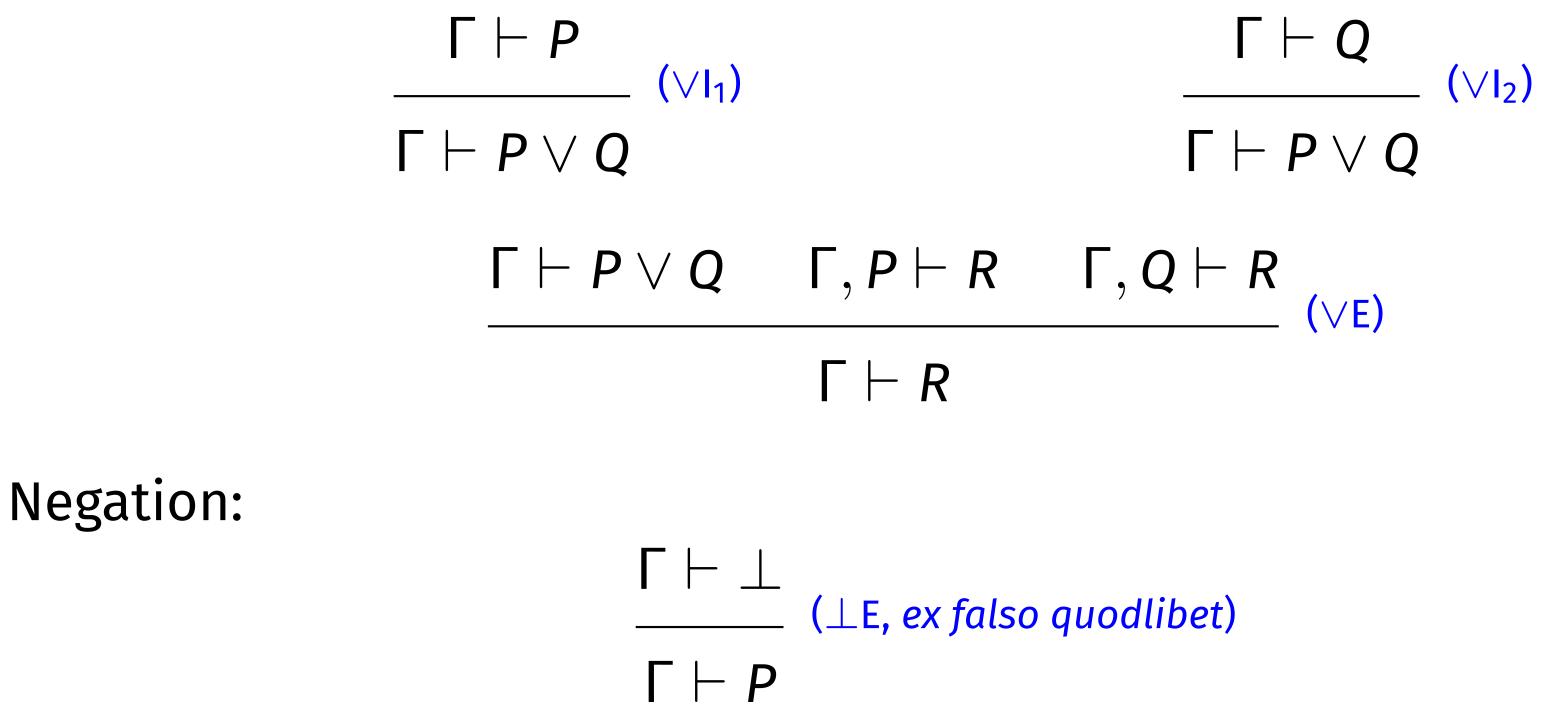
 $\Gamma \vdash P \land Q$ 

("I" stands for Introduction, "E" for Elimination)



### Rules of intuitionistic logic

Disjunction:



### Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

### Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

Why Types?

### Folk wisdom: Don't compare apples to oranges. On n'additionne pas des choux et des carottes.

Physical wisdom: dimensional analysis

- d = v.t dist = dist

homogeneous, possibly correct d = v/t dist  $\neq$  dist.time<sup>-2</sup> not homogeneous, always false

## Types in programming languages

To detect and reject some programming errors.

"foo" (12) (character string  $\neq$  function) cos(45, "degrees") (wrong number of arguments)

As partial specifications of data structures and module interfaces.

val assoc: 'a -> ('a \* 'b) list -> 'b

To clarify the meaning of programs and facilitate compilation.

integer			n				
re	ea.	L 3	ζ				
n	Ξ	n	*	2	+	1	(intege
X	=	X	*	2	+	1	(floati

- type 'a tree = Leaf of 'a | Node of 'a tree \* 'a tree

```
ger arithmetic)
ing-point arithmetic)
```

# Types in mathematical logic

In elementary mathematics: an intuitive notion. For example, in Euclidean geometry, a point is not a line, and we never talk about the intersection of two points.

In naive set theory: no types a priori.

But we get paradoxes such as Russell's:  $\{x \mid x \notin x\}$ .

Type theories: add types to a logic to rule out paradoxes. cest of the kind  $x \in x$ .



- For example, the encoding of natural numbers  $n + 1 \stackrel{def}{=} n \cup \{n\}$ .

  - Typing is a sort of superego forbidding certain forms of logical in-

(J.-Y. Girard, The blind spot)

# The simply-typed lambda-calculus

The modern presentation separates the syntax of terms from the typing judgment  $\Gamma \vdash M : \alpha$ (read: term *M* has type  $\alpha$  under hypotheses  $\Gamma$ ). Terms:  $M, N ::= x \mid \lambda x : \alpha . M \mid M N$ Typing rules:

 $\Gamma, \mathbf{x} : \alpha \vdash \mathbf{M} : \beta$  $\Gamma \vdash \lambda \mathbf{x}: \alpha \cdot \mathbf{M} : \alpha \to \beta$ 

### $\Gamma_1, \mathbf{x} : \alpha, \Gamma_2 \vdash \mathbf{x} : \alpha$ $\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha$ $\Gamma \vdash M N : \beta$

# Type safety in programming languages

**Theorem:** (type soundness of simply typed lambda calculus)

If  $\emptyset \vdash M : \tau$  then there exists a value  $v = \lambda x : \tau' \cdot N$  such that  $M \rightarrow_{\beta}^* v \cdot$ 

#### "Well typed programs do not go wrong." – Robin Milner, 1978

# Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

# Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

### Combinatory logic

As early as 1930, Curry studies combinatory logic: a variant of the  $\lambda$ -calculus without abstraction  $\lambda x.M$ , but with combinators predefined by their conversion rules. For example:

Terms: M, N ::= M N | S | K | IRules: I x = xK x y = xS x y z = x z (y z)

Every  $\lambda$ -term can be encoded in combinatory logic if a suitable combinator basis is provided, such as S, K, I.

Typing combinatory logic using simple types:

$$M: \alpha \to \beta \qquad N: \alpha \qquad I: \alpha$$

 $MN:\beta$ 

 $\alpha \to \alpha \qquad \qquad \mathbf{K} : \alpha \to \beta \to \alpha$ 

 $S: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$ 

### Curry's correspondence

 $P \Rightarrow Q$  between propositions,

implication in a Hilbert-style logic:

• The typing rule for function application is *modus ponens*:

$$\frac{M: \alpha \to \beta \quad N: \alpha}{MN: \beta} \qquad \qquad \frac{P \Rightarrow Q \quad P}{Q}$$

M N :  $\beta$ 

On pages 313–314 of their book *Combinatory logic* (1958), Curry and Feys remark that, if we read the arrow in type  $\tau_1 \rightarrow \tau_2$  like the implication

• The types for the combinators S, K, I are the axioms that define

### Curry's correspondence

Via Curry's correspondence, a proposition P is provable if and only if

#### the corresponding type $\tau$ is **inhabited**, that is, there exists a closed term with type $\tau$ .

### Howard's manuscript

W. A. Howard, The formulae-as-types notion of construction, 1969

Extends and modernize Curry's result:

- Lambda-calculus instead of combinatory logic.
- Intuitionistic sequents instead of Hilbert-style axioms.
- Treats the connectors  $\land$ ,  $\lor$ ,  $\neg$  in addition to  $\Rightarrow$ ; discusses the quantifiers  $\forall$  and  $\exists$ .
- over proofs.

Our Correspondence between reduction over terms and cut elimination

#### Curry-Howard: implication

#### $\Gamma, \mathbf{X} : \mathbf{A} \vdash \mathbf{M} : \mathbf{B}$

 $\Gamma \vdash \lambda x. M : A \rightarrow B$ 

#### $\Gamma_1, \mathbf{X} : \mathbf{A}, \Gamma_2 \vdash \mathbf{X} : \mathbf{A}$

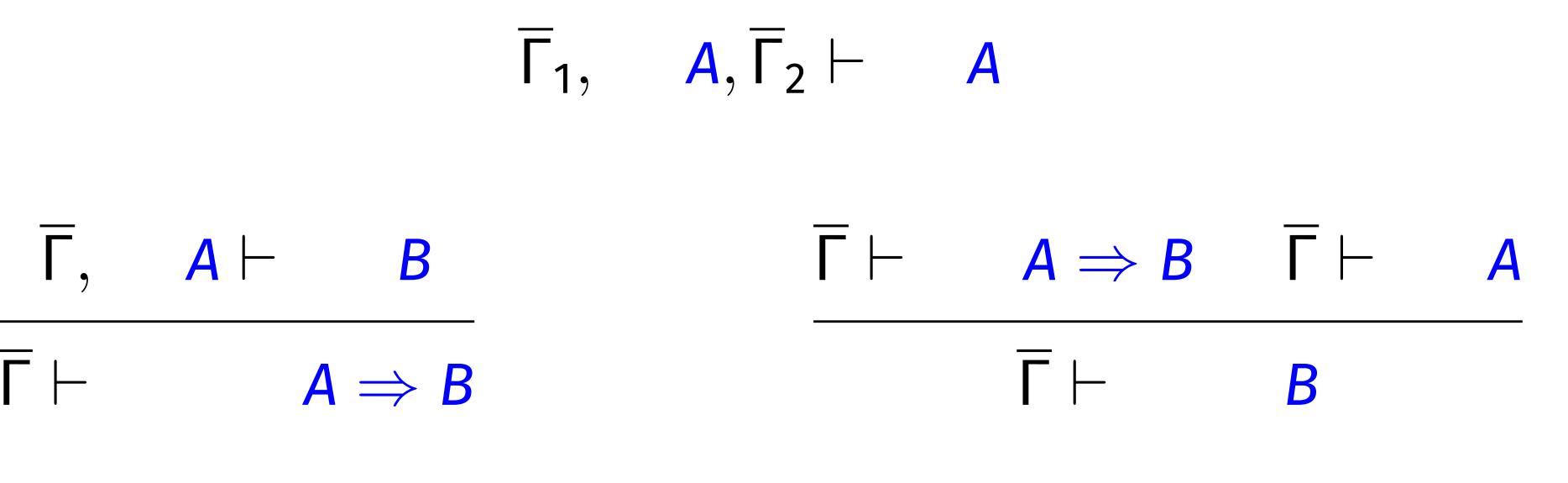
#### $\Gamma \vdash M : A \rightarrow B \qquad \Gamma \vdash N : A$

 $\Gamma \vdash M N : B$ 

### Curry-Howard: implication

# $\overline{\Gamma} \vdash A \Rightarrow B$

 $\Gamma$  is  $\Gamma$  without variable names, for example  $\overline{x}: A, y: A = A, A$ .



#### Curry-Howard: conjunction

Extend the lambda-calculus with pairs and projections:

- Types:  $A, B ::= \dots | A \times B$
- Terms:  $M, N ::= \dots |\langle M, N \rangle | \pi_1 M | \pi_2 M$

 $\Gamma \vdash M : A \qquad \Gamma \vdash N : B$  $\Gamma \vdash \langle M, N \rangle : \mathbf{A} \times \mathbf{B}$ 

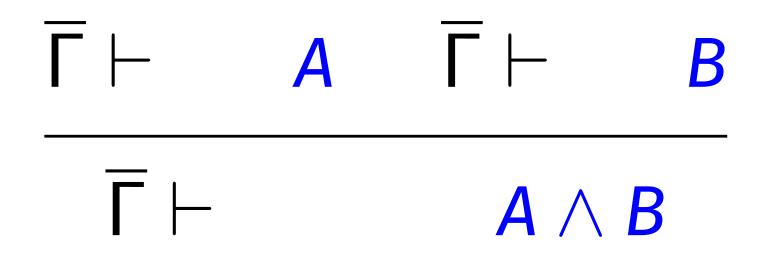


 $\Gamma \vdash M : A \times B$  $\Gamma \vdash M : A \times B$  $\Gamma \vdash \pi_1 M : A$  $\Gamma \vdash \pi_2 M : B$ 

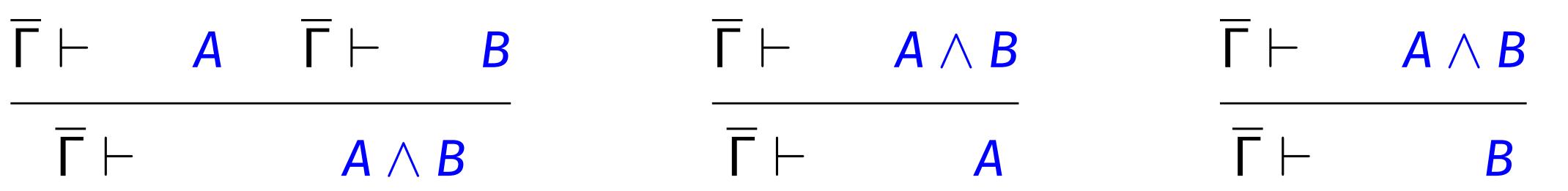
### Curry-Howard: conjunction

Extend the lambda-calculus with pairs and projections:

- Types:  $A, B ::= \dots | A \times B$
- Terms:  $M, N ::= \dots |\langle M, N \rangle | \pi_1 M | \pi_2 M$







### Curry-Howard: disjunction

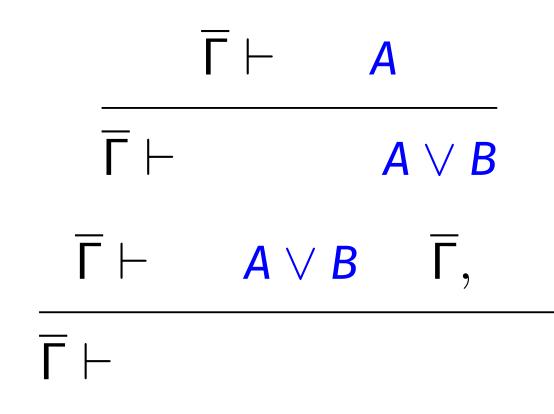
Extend the lambda-calculus a sum type, its constructors, and pattern matching

Types: A, B ::= ... | A + BTerms:  $M, N ::= \dots | \operatorname{inj}_1 M | \operatorname{inj}_2 M$ match M with  $\operatorname{inj}_1 X_1 \Rightarrow N_1 | \operatorname{inj}_2 X_2 \Rightarrow N_2$  end  $\Gamma \vdash N : B$  $\Gamma \vdash M : A$  $\Gamma \vdash \operatorname{inj}_2 N : A + B$  $\Gamma \vdash \operatorname{inj}_1 M : A + B$  $\Gamma \vdash M : A + B$   $\Gamma, x_1 : A \vdash N_1 : C$   $\Gamma, x_2 : B \vdash N_2 : C$  $\Gamma \vdash \text{match } M \text{ with inj}_1 x_1 \Rightarrow N_1 \mid \text{inj}_2 x_2 \Rightarrow N_2 \text{ end} : C$ 

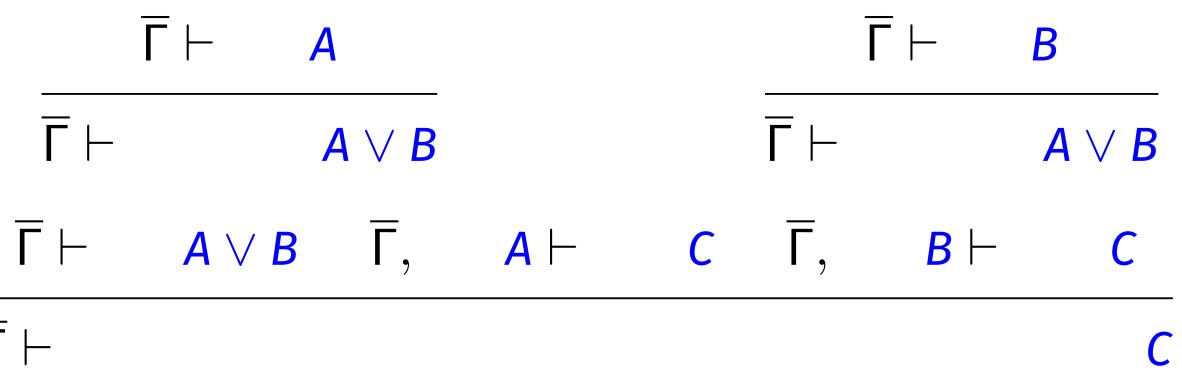
### Curry-Howard: disjunction

Extend the lambda-calculus a sum type, its constructors, and pattern matching

Types: A, B ::= ... | A + BTerms:  $M, N ::= \dots | \operatorname{inj}_1 M | \operatorname{inj}_1 M |$ match M with ir



$$\mathtt{j}_2 \ M$$
 $\mathtt{nj}_1 \ \mathtt{x}_1 \Rightarrow \mathtt{N}_1 \ | \ \mathtt{inj}_2 \ \mathtt{x}_2 \Rightarrow \mathtt{N}_2 \ \mathtt{end}$ 



# Curry-Howard: absurdity

Extend the lambda-calculus an empty type and "pattern matching" with zero branches

Types:  $A, B ::= \dots | \perp$ Terms:  $M, N ::= \dots \mid \text{match } M \text{ with end}$ 

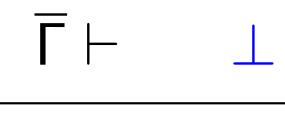
 $\[ \ \vdash match M with end : A \]$ 

 $\Gamma \vdash M : \bot$ 

# Curry-Howard: absurdity

Extend the lambda-calculus an empty type and "pattern matching" with zero branches

Types:  $A, B ::= ... \mid \bot$ Terms:  $M, N ::= ... \mid match M$  with end



Ē⊢

Α

# Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

# Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

# Static typing v. genericity

Static typing with simple types (as in simply-typed  $\lambda$ -calculus but also as in Algol, Pascal, etc) sometimes forces us to duplicate code.

#### Example

A sorting algorithm applies to any list list(t) of elements of type t, provided it also receives then function  $t \to t \to bool$  that compares two elements of type t.

With simple types, to sort lists of integers and lists of strings, we need two functions with different types, even if they implement the same algorithm:

 $\texttt{sort\_list\_int}: (\texttt{int} \to \texttt{int} \to \texttt{bool}) \to \texttt{list}(\texttt{int}) \to \texttt{list}(\texttt{int})$ 

 $\texttt{sort\_list\_string}: (\texttt{string} \rightarrow \texttt{string} \rightarrow \texttt{bool}) \rightarrow \texttt{list}(\texttt{string}) \rightarrow \texttt{list}(\texttt{string})$ 

# Static typing v. genericity

There is a tension between static typing on the one hand and reusable implementations of generic algorithms on the other hand.

Some languages elect to weaken static typing, e.g. by introducing a universal type "any" or "?" with run-time type checking, or even by turning typing off:

Instead, polymorphic typing extends the algebra of types and the typing rules so as to give a precise type to a generic function.

### The polymorphic lambda-calculus

(John C. Reynolds, Towards a theory of type structure, 1974)

syntactic checking of type correctness.

Extends simply-typed lambda-calculus with the ability to abstract over a type variable and to apply such an abstraction to a type:

Terms:
$$M, N ::= x \mid \lambda x:t. M \mid M N$$
 $\mid \Lambda X. M$ a $\mid M[t]$ ir

 $t ::= X \mid t_1 \to t_2 \mid \forall X. t$ Types:

We suggest that a solution to [the polymorphic sort function] problem is to permit types themselves to be passed as a special kind of parameter, whose usage is restricted in a way which permits the

> bstraction over type X nstantiation with type t

# The polymorphic lambda-calculus

Continuing the list sorting example, the generic sorting function can be given type

 $sort_list : \forall X. (X \rightarrow X \rightarrow bool)$ 

Its implementation is of the following shape:

 $sort\_list = \bigwedge X. \lambda cmp : X \rightarrow X \rightarrow bool. \lambda l : list(X). M$ 

The function can be used for integer lists as well as for string lists just by instantiation:

 $\texttt{sort_list[int]}: (\texttt{int} \to \texttt{int} \to \texttt{bool}) \to$  $\texttt{sort_list[string]}:(\texttt{string} \to \texttt{string} \to \texttt{bound})$ 

$$ightarrow \texttt{list}(X) 
ightarrow \texttt{list}(X)$$

$$ightarrow extsf{list(int)} 
ightarrow extsf{list(int)} 
ightarrow extsf{list(string)} 
ightarrow extsf{list(string)}$$

### Typing and reduction rules

The rules of simply-typed lambda calculus:

$$\Gamma_{1}, x: t, \Gamma_{2} \vdash x: t \qquad \frac{\Gamma, x: t \vdash M: t'}{\Gamma \vdash \lambda x: t. M: t \rightarrow t'} \qquad \frac{\Gamma \vdash M: t \rightarrow t' \quad \Gamma \vdash N: t}{\Gamma \vdash M N: t'}$$

Plus two rules for introduction and elimination of polymorphism:

 $\Gamma \vdash M : t$  X not free in  $\Gamma$ 

 $\Gamma \vdash \Lambda X. M : \forall X. t$ 

A new form of  $\beta$ -reduction:

$$(\Lambda X. M)[t]$$



$$\Gamma \vdash M : \forall X. t$$
$$\Gamma \vdash M[t'] : t\{X \leftarrow t'\}$$

$$\beta M\{X \leftarrow t\}$$

# The polymorphic lambda-calculus in practice

Second-class polymorphism ( $\approx$  polymorphic definitions, but monomorphic values):

- Generics in Ada, Java, C#.
- In the ML family languages (SML, OCaml, OCaml, SML, OCaml, OCA Haskell, etc), with inference of types, of  $\Lambda$  and of instantiations:

Type scheme:  $\sigma := \forall \alpha_1, \ldots, \alpha_n, \tau$  for let-bound variables

First-class polymorphism ( $\approx$  function parameters can have  $\forall$  types):

type poly\_id = { id : 'a. 'a -> 'a }

- Simple types:  $\tau := \alpha | \tau_1 \rightarrow \tau_2 | \ldots$  for all other variables and values

- Recent extensions of OCaml and of Haskell. An example in OCaml

# Polymorphism = Abstraction

Simply-typed lambda calculus: Functions: values can be abstracted over values (i.e., values depend on values)

**Polymorphic lambda calculus:** 

**Polymorphic lambda calculus with data types:** *Type constructors*: types can be abstracted over types (e.g., List<T> in Java)

What about *types depending on values*?

- Polymorphic functions: values can be abstracted over types (e.g., polymorphic sorting)
  - - cf. Barendregt's lambda-cube (1992)

# Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

# Today's Agenda

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and **Quantifying over Values**
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

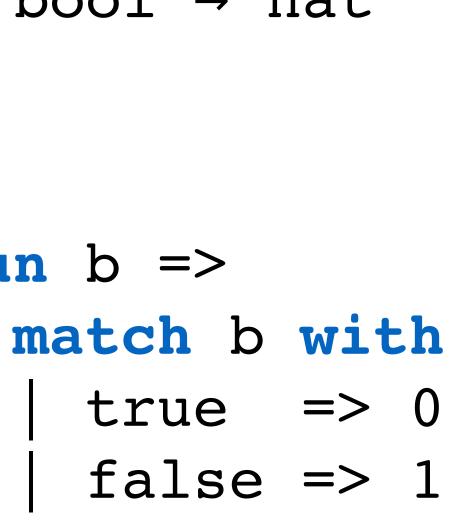
### Ordinary function type

bool  $\rightarrow$  nat

 $A \rightarrow B$ 

**fun** b =>

end



### Dependent function type

**fun** b => end

 $\Pi(x: A) \cdot B(x)$ 

n(b: bool). if b then nat else bool

- match b with

  - true => 0
    false => true

### Dependent types for programs

In Fortran, C or C++, the type of an array t[N] contains

- a type *t*: the type of the array elements
- a "term" (constant expression) N: the size of the array.

and a term N, and produces the type array(t, N).

Lifting the restriction that N is a constant expression, and allowing to array operations:

concat :  $\forall t. \forall N_1. \forall N_2. array(t, N_1)$ 

- In other words, the array type constructor takes two parameters, a type t
- ourselves to quantify over this N, we can give very precise dependent types

$$(1) \rightarrow \operatorname{array}(t, N_2) \rightarrow \operatorname{array}(t, N_1 + N_2)$$

### Curry-Howard for dependent types

#### $\Pi(x: A). P(x)$ $\forall x \in A, P(x)$ should be read as

#### Dependent function type corresponds to universal quantification!



### A familiar dependent type

- $\Pi(P : nat -> Prop).$ P(0) ->
  - $\Pi(n : nat)$ . P n

... and its inhabitant in Coq

```
Function nat ind (P : nat -> Prop)
             (f0 : P 0)
 fun (n : nat) =>
   match n with
     0 => f0
n'.+1 => fn n' (nat_ind P f0 fn n0)
   end.
```

 $(\Pi(n : nat). P(n) -> P(n.+1))$ 

 $(fn : \Pi(n : nat), P n -> P (n.+1)) :=$ 

#### Dependent types as theorems

**Theorem** counterexample (A: Type) (P:  $A \rightarrow Prop$ ) :  $(\exists x: A, \neg P x) \rightarrow \neg (\forall x, P x).$ **Proof. by case** => x H1 H2; apply : H1 (H2 x). **Qed.** 

#### Dependent types as theorems

**Theorem** counterexample (A: Type) (P:  $A \rightarrow Prop$ ) :  $(\exists x: A, \neg P x) \rightarrow \neg (\forall x, P x).$ **Proof. by case** => x H1 H2; apply : H1 (H2 x). Qed.

**Function** counterexample := fun (A : Type) (P : A -> Prop) (hyp : ∃x : A, ¬ P x) => (fun F :  $\forall$ (x : A) (p : (fun x0 : A => ¬ P x0) x), match hyp as e return ((fun :  $(\exists x : A, \neg P x) = \neg (\forall x : A, P x)) e$ ) with | ex x x0 => F x x0

#### Proof term in Coq

 $(fun : (\exists x0 : A, \neg P x0) => \neg (\forall x0 : A, P x0))$ (ex intro (fun x0 : A => ¬ P x0) x p) => end) (fun (x : A) (H1 :  $\neg$  P x) (H2 :  $\forall$ x0 : A, P x0) => H1 (H2 x)).



#### Dependent types as theorems

**theorem** counterexample { $\alpha$ : Type} (P:  $\alpha \rightarrow$  Prop) :  $(\exists x: A, \neg P x) \rightarrow \neg (\forall x, P x) := by$ sby scase => x H1 H2; apply H1

theorem counterexample :  $\forall \{\alpha : Type\}$  (P :  $\alpha \rightarrow Prop$ ),  $(\exists x, \neg P x) \rightarrow \neg \forall (x : \alpha), P x :=$ fun { $\alpha$ } P H => Exists.casesOn (motive := fun t => H = t  $\rightarrow \neg \forall$  (x :  $\alpha$ ), P x) H (fun w h h 1 H2 => h (Eq.mp ((congrArg Not ((fun x => eq true (H2 x)) w)).trans not\_true eq false) h).elim) (Eq.refl H)

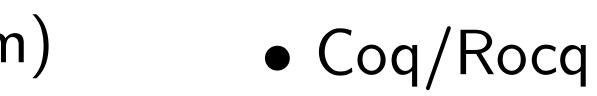
Proof term in Lean



# Programming with dependent types

- Scala (limited form)
- Agda

theorem reverse reverse {α : Type} :  $\forall xs : List \alpha$ , reverse (reverse xs) = xs => **by** rfl [] x :: xs => by simp [reverse, reverse\_append, reverse\_reverse xs]



### • Lean

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and **Quantifying over Values**
- Curry-Howard for Practitioners: **Program Synthesis** and **Proof Repair**

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: Program Synthesis and Proof Repair

# Exploiting the PAT principle

If propositions = types and proofs = terms, we can reuse principles for proofs construction to generate programs and vice versa.

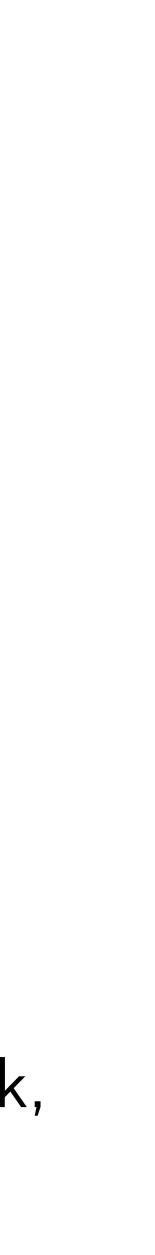
### **Program Synthesis**

Write a theorem specifying a program, build a constructive proof that such program exists, the obtained program is correct by construction.

### **Proof Repair**

Definitions involved into a theorem changed: Refactor the proof as a program working a new data type to it would type-check, the result is a new valid proof.



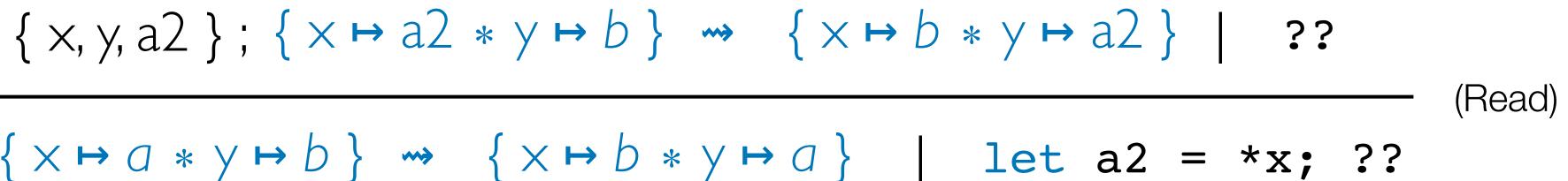


### Program Synthesis (Polikarpova & Sergey, 2019)

- $\{ \times \mapsto a * \vee \mapsto b \}$
- void swap(loc x, loc y)
  - $\{ \times \mapsto b \ast \vee \mapsto a \}$



 $\{x, y\}; \{x \mapsto a * y \mapsto b\} \Rightarrow \{x \mapsto b * y \mapsto a\} \mid \text{let a2} = *x; ??$ 



# $\{ x, y, a2, b2 \}; \{ x \mapsto a2 * y \mapsto b2 \}$ $\{ x, y, a2 \}; \{ x \mapsto a2 * y \mapsto b \} \Rightarrow \{ x \mapsto a2 * y \mapsto b \} \Rightarrow \{ x \mapsto a2 * y \mapsto b \} \Rightarrow \{ x \mapsto a2 * y \mapsto b \} \Rightarrow \{ x \mapsto a2 * y \mapsto b \} \Rightarrow \{ x \mapsto a2 * y \mapsto b \} \Rightarrow \{ x \mapsto a2 * y \mapsto b \} \Rightarrow \{ x \mapsto a2 * y \mapsto b \} \Rightarrow \{ x \mapsto b2 \} = \{ x \mapsto b2 \} =$

$$\begin{array}{c} & \twoheadrightarrow & \{ \times \mapsto b2 \ast y \mapsto a2 \} & | & ?? \\ \hline & \mapsto b \ast y \mapsto a2 \} & | & \text{let } b2 & = & *y; & ?? \\ \hline & \mapsto b \ast y \mapsto a \} & | & \text{let } a2 & = & *x; & ?? \end{array}$$
 (Read)

# 

$  \  \  \  \  \  \  \  \  \  \  \  \  \$	$- (\Lambda / rito)$
$\{ x \mapsto b2 * y \mapsto a2 \}$   *x = b2; ?	/ <b></b>
<pre> → b * y → a2 }   let b2 = *y; ?? </pre>	(Read)
$\{ \mapsto b * y \mapsto a \}$   let a2 = *x; ??	(Read)

{ x, y, a2, b2 }; { y ↦ b2  $\{x, y, a2, b2\}; \{x \mapsto b2 * y \mapsto b2\}$  $\{x, y, a2, b2\}; \{x \mapsto a2 * y \mapsto b2\}$  $\{x, y, a2\}; \{x \mapsto a2 * y \mapsto b\} \Rightarrow \{x \vdash a2 * y \mapsto b\}$  $\{x, y\}; \{x \mapsto a * y \mapsto b\} \twoheadrightarrow \{x\}$ 

{ x, y, a2, b2 }; { y ↦

{ x, y, a2, b2 }; { y ↦ b2

 $\{x, y, a2, b2\}; \{x \mapsto b2 * y \mapsto b2\}$ 

 $\{x, y, a2, b2\}; \{x \mapsto a2 * y \mapsto b2\}$ 

 $\{x, y, a2\}; \{x \mapsto a2 * y \mapsto b\} \twoheadrightarrow \{x \mapsto a2 * y \mapsto b\}$ 

 $\{x, y\}; \{x \mapsto a * y \mapsto b\} \twoheadrightarrow \{x$ 

$\rightarrow a2 \} \longrightarrow \{ y \mapsto a2 \}   ??$		
	- (Write)	
$\{ y \mapsto a2 \}   *y = a2;$	??	
$\} \implies \{ x \mapsto b2 * y \mapsto a2 \}$		(Frame)
$\{ \times \mapsto b2 \ast \gamma \mapsto a2 \} $ $*x =$	= b2;	(Write) ??
		— (Read)
$\mapsto b * y \mapsto a2 \}$   let b2 =	*y; ?	
$\mapsto b * y \mapsto a \}$   let a2 =	*x; ?	- (Read)

- { x, y, a2, b2 }; { e
- { x, y, a2, b2 }; { y ►
- { x, y, a2, b2 } ; { y ↦ b2
- $\{x, y, a2, b2\}; \{x \mapsto b2 * y \mapsto b2$

 $\{x, y, a2, b2\}; \{x \mapsto a2 * y \mapsto b2\}$ 

 $\{x, y, a2\}; \{x \mapsto a2 * y \mapsto b\} \twoheadrightarrow \{x\}$ 

 $\{x, y\}; \{x \mapsto a * y \mapsto b\} \twoheadrightarrow \{x$ 

emp } 🛶 { emp }   ??	Ŋ	
$\rightarrow a2 \} \rightarrow \{\gamma \mapsto a2 \}   ??$	ame) (\A/rito)	
2} $\rightarrow$ { $y \mapsto a2$ }   $*y = a2;$	(VVrite) ??	(Eromo)
	??	(Frame) — (Write)
$\{ \times \mapsto b2 * \gamma \mapsto a2 \}   *x =$		( , , , , , , , , , , , , , , , , , , ,
$\mapsto b * y \mapsto a2 \}  $ let $b2 = *$		?
$( \mapsto b * y \mapsto a \}   let a2 =$	*x; ?	- (Read) ?

$$\frac{\langle x, y, a2, b2 \rangle; \{emp\} \twoheadrightarrow \{emp\} | skip}{\langle x, y, a2, b2 \rangle; \{y \mapsto a2 \} \implies \{y \mapsto a2 \} | ??}$$
(Frame)  

$$\frac{\langle x, y, a2, b2 \rangle; \{y \mapsto a2 \} \implies \{y \mapsto a2 \} | ??$$
(Write)  

$$\frac{\langle x, b2 \rangle; \{y \mapsto b2 \} \implies \{y \mapsto a2 \} | [*y = a2; ??]$$
(Frame)  

$$\frac{\langle x \mapsto b2 \ast y \mapsto b2 \} \implies \{x \mapsto b2 \ast y \mapsto a2 \} | ??$$
(Write)  

$$\frac{\langle x \mapsto b2 \rangle; 2?}{\langle x \mapsto b2 \rangle \implies \langle x \mapsto b2 \ast y \mapsto a2 \rangle | [*x = b2; ??]$$
(Read)  

$$\frac{\langle x \mapsto b2 \rangle \implies \langle x \mapsto b \ast y \mapsto a2 \rangle | [tt b2 = *y; ??]$$
(Read)  

$$\frac{\langle x \mapsto b2 \rangle \implies \langle x \mapsto b \ast y \mapsto a2 \rangle | [tt a2 = *x; ??]$$
(Read)

$$\{ x, y, a2, b2 \}; \{ emp \} \rightarrow \{ emp \} | skip$$
(Frame)  

$$\{ x, y, a2, b2 \}; \{ y \mapsto a2 \} \rightarrow \{ y \mapsto a2 \} | ??$$
(Write)  

$$y, a2, b2 \}; \{ y \mapsto b2 \} \rightarrow \{ y \mapsto a2 \} | *y = a2; ??$$
(Frame)  

$$; \{ x \mapsto b2 * y \mapsto b2 \} \rightarrow \{ x \mapsto b2 * y \mapsto a2 \} | ??$$
(Write)  

$$\Rightarrow a2 * y \mapsto b2 \} \rightarrow \{ x \mapsto b2 * y \mapsto a2 \} | *x = b2; ??$$
(Read)  

$$2 * y \mapsto b\} \rightarrow \{ x \mapsto b * y \mapsto a2 \} | 1et b2 = *y; ??$$
(Read)  

$$a * y \mapsto b\} \rightarrow \{ x \mapsto b * y \mapsto a\} | 1et a2 = *x; ??$$
(Read)

{ x, y, a2, b2

{ x, y, a2, b2 }; {

{ x, y, a2 }; { x

{ ×, y } ; { >

- void swap(loc x, loc y) {

  - \*x = b2;
  - \*y = a2;

}

- let  $a^2 = *x;$
- let b2 = \*y;

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and Quantifying over Values
- Curry-Howard for Practitioners: Program Synthesis and Proof Repair

- Towards Curry-Howard: the Lambda-Calculus
- Intuitionistic Logic and the idea of Constructive Proofs
- Types and Simply-Typed Lambda Calculus
- Curry-Howard Correspondence: Propositions as Types, **Proofs** = **Programs**
- Polymorphic Lambda-Calculus
- Dependent Types and **Quantifying over Values**
- Curry-Howard for Practitioners: Program Synthesis and Proof Repair

# **Computational Reflection**

### Use mechanisms for type inference for switch from proofs to programs and back.

### Small Scale Reflection for the Working Lean User

### Vladimir Gladshtein 🖂 🏠 🕩

School of Computing, National University of Singapore, Singapore

### George Pîrlea ⊠∦©

School of Computing, National University of Singapore, Singapore

### Ilya Sergey ⊠ **☆** <sup>(</sup>

School of Computing, National University of Singapore, Singapore

### — Abstract -

We present the design and implementation of the Small Scale Reflection proof methodology and tactic language (a.k.a. SSR) for the Lean 4 proof assistant. Like its Coq predecessor SSReflect, our Lean 4 implementation, dubbed LeanSSR, provides powerful rewriting principles and means for effective management of hypotheses in the proof context. Unlike SSReflect for Coq, LeanSSR does not require explicit switching between the *logical* and *symbolic* representation of a goal, allowing for even more concise proof scripts that seamlessly combine deduction steps with proofs by computation.

In this paper, we first provide a gentle introduction to the principles of structuring mechanised proofs using LeanSSR. Next, we show how the native support for metaprogramming in Lean 4 makes it possible to develop LeanSSR entirely within the proof assistant, greatly improving the overall experience of both tactic implementers and proof engineers. Finally, we demonstrate the utility of LeanSSR by conducting two case studies: (a) porting a collection of Coq lemmas about sequences from the widely used Mathematical Components library and (b) reimplementing proofs in the finite set library of Lean's mathlib4. Both case studies show significant reduction in proof sizes.

### Tuesday, 23 April, 14:30

### To take away

Formal proofs are functional programs in disguise; propositions are program types; writing a **proof** = constructing a **program**.

Thinking in terms of Curry-Howard correspondence brings new insights allowing to combine and reuse ideas between Formal Mathematics and Computer Science



slides of this talk (in case you missed it)

Thanks!

## Further reading

- Xavier Leroy. The Curry-Howard correspondence today. <u>xavierleroy.org/CdF/2018-2019/</u>
- Benjamin Pierce. Types and Programming Languages. MIT Press, 2002
- Ilya Sergey. Programs and Proofs: Mechanizing Mathematics with Dependent Types
- Yves Bertot, Pierre Castéran. Interactive Theorem Proving and Program Development.
- David Thrane Christiansen. *Functional Programming in Lean*
- Ilya Sergey and Nadia Polikarpova. Structuring the Synthesis of Heap-Manipulating Programs
- Talia Ringer. Proof Repair. PhD Thesis, 2021

• Kiran Gopinathan, Mayank Keoliya, Ilya Sergey. Mostly Automated Proof Repair for Verified Libraries, 2023

