Programming = proving?
The Curry-Howard correspondence today

Second lecture

Polymorphism all the way up!
From System F to the Calculus of Constructions

Xavier Leroy

Collège de France

2018-11-21

COLLÈGE
DE FRANCE
1530

Curry-Howard in 1970

An isomorphism between simply-typed λ-calculus and intuitionistic logic that connects

- types and propositions;
- terms and proofs;
- reductions and cut elimination.

This second lecture shows how:

- This correspondence extends to more expressive type systems and to more powerful logics.
- This correspondence inspired formal systems that are simultaneously a logic and a programming language (Martin-Löf type theory, Calculus of Constructions, Pure Type Systems).

I

Polymorphism and second-order logic

Static typing vs. genericity

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

Example

A sorting algorithm applies to any list $\texttt{list}(t)$ of elements of type t, provided it also receives then function $t \rightarrow t \rightarrow \texttt{bool}$ 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:

$$\texttt{sort_list_int} : (\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{bool}) \rightarrow \texttt{list}(\texttt{int}) \rightarrow \texttt{list}(\texttt{int})$$

$$\texttt{sort_list_string} : (\texttt{string} \rightarrow \texttt{string} \rightarrow \texttt{bool}) \rightarrow \texttt{list}(\texttt{string}) \rightarrow \texttt{list}(\texttt{string})$$

Static typing vs. 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:

```
void qsort(void * base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

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)

> *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 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:	$M, N ::= x \mid \lambda x{:}t.\, M \mid M\ N$	
	$\mid \Lambda X.\, M$	abstraction over type X
	$\mid M[t]$	instantiation with type t
Types:	$t ::= X \mid t_1 \to t_2 \mid \forall X.\, t$	

The polymorphic lambda-calculus

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

$$\texttt{sort_list} : \forall X.\ (X \to X \to \texttt{bool}) \to \texttt{list}(X) \to \texttt{list}(X)$$

Its implementation is of the following shape:

$$\texttt{sort_list} = \Lambda X.\ \lambda cmp : X \to X \to \texttt{bool}.\ \lambda l : \texttt{list}(X).\ M$$

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

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

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

Typing and reduction rules

The rules of simply-typed lambda calculus:

$$\Gamma_1, x : t, \Gamma_2 \vdash x : t \qquad \frac{\Gamma, x : t \vdash M : t'}{\Gamma \vdash \lambda x{:}t.\, M : t \to t'} \qquad \frac{\Gamma \vdash M : t \to t' \qquad \Gamma \vdash N : t}{\Gamma \vdash M\, N : t'}$$

Plus two rules for introduction and elimination of polymorphism:

$$\frac{\Gamma \vdash M : t \qquad X \text{ not free in } \Gamma}{\Gamma \vdash \Lambda X.\, M : \forall X.\, t} \qquad \frac{\Gamma \vdash M : \forall X.\, t}{\Gamma \vdash M[t'] : t\{X \leftarrow t'\}}$$

A new form of β-reduction:

$$(\Lambda X.\, M)[t] \;\to_\beta\; M\{X \leftarrow t\}$$

Abstract types

The dual problem of reusing generic functions is the problem of independence w.r.t. the representations of an abstract type.

Example (The abstract type of integer sets)

It is described as a type name `IS` (integer set) and the following constants and operations:

```
{ empty: IS;  add: int -> IS -> IS;  member: int -> IS -> bool }
```

Several implementations for `IS` and its operations are possible, for example bit vectors, or binary search trees.

To leave us with complete freedom to change the implementation, we would like that programs using the `IS` type do not depend on its implementation (bit vectors or search trees?) and use exclusively the constants and operations `empty`, `add`, `member`.

Abstract types

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

Reynolds observes that a way to hide the representation of an abstract type is to make its users polymorphic in its implementation type.

Example (Using sets of integers)

```
let use_intset : ∀IS. {...} -> bool =
  ΛIS. λops: { empty: IS; add: int -> IS -> IS;
               member: int -> IS -> bool }.
  ops.member 1 (ops.add 2 ops.empty)
```

The `IS` type being a type variable, the only way to build and use values of type `IS` is through the operations `ops`.

The polymorphic lambda-calculus in practice

Second-class polymorphism
(≈ polymorphic definitions, but monomorphic values):

- Generics in Ada, Java, C#.
- Hindley-Milner typing in the ML family languages (SML, OCaml, Haskell, etc), with inference of types, of Λ and of instantiations:

Type scheme:	$\sigma ::= \forall \alpha_1, \ldots \alpha_n . \tau$	for `let`-bound variables
Simple types:	$\tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \ldots$	for all other variables and values

First-class polymorphism
(≈ function parameters can have ∀ types):

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

```
type poly_id = { id : 'a. 'a -> 'a }
```

System F

A few years before Reynolds, circa 1970, logician Jean-Yves Girard invented the same polymorphic lambda-calculus under the name "System F".

Girard was motivated not by generic programming, but by the study of second-order arithmetic via a functional interpretation (in the BHK style).

Girard knew Howard's manuscript and acknowledges its influence on his work:

> *System F, contrary to simply-typed λ-calculus, is constructed around Curry-Howard, as the isomorphic image of intuitionistic second-order propositional calculus.*
>
> *(Jean-Yves Girard, The blind spot, ch.6)*

First-order arithmetic, second-order arithmetic

First-order arithmetic:

- Peano natural numbers (zero and successor)
- predicate calculus
- quantification $\forall$, $\exists$ over numbers exclusively.

Second-order arithmetic: the same, plus

- quantification over sets of natural numbers,
  i.e. over predicates $\mathbb{N} \to \texttt{Prop}$,
  i.e. over real numbers.

Second-order arithmetic suffices to express a large part of calculus.

Functional interpretations of arithmetic

(Jean-Yves Girard, *Une extension de l'interprétation fonctionnelle de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types*, 1971)

(Jean-Yves Girard, *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, thèse d'Etat, U. Paris 7, 1972)

Gödel (1958) introduced "System T" (= simply-typed λ-calculus + primitive recursion) to develop a BHK-style functional interpretation for first-order arithmetic.

Girard develops a similar approach to second-order arithmetic.

By showing normalization for his System F, he also proves a conjecture by Takeuti (1953) concerning cut elimination for second-order arithmetic.

Expressiveness of System F

In System T, Gödel was forced to add numbers and recursion as primitive constructs, because simply-typed λ-calculus cannot encode them adequately.

In System F, these notions and many others can be expressed by encodings similar to those of the pure λ-calculus, encodings that System F is able to type polymorphically enough:

- Natural numbers: $\texttt{nat} \equiv \forall X.\, X \to (X \to X) \to X$
- Booleans: $\texttt{bool} \equiv \forall X.\, X \to X \to X$
- Logical connectives:
  $\bot \equiv \forall X.\, X$
  $A \wedge B \equiv \forall X.\, (A \to B \to X) \to X$
  $A \vee B \equiv \forall X.\, (A \to X) \to (B \to X) \to X$
- Quantifiers: $\exists X.\, A \equiv \forall Y.\, (\forall X.\, A \to Y) \to Y$

Normalization for System F

Theorem (Strong normalization for F)

If $\Gamma \vdash M : t$, any β-reduction sequence starting with M is finite.

The proof is considerably more difficult than the proof of normalization for simply-typed λ-calculus:

- System F is impredicative: in $\forall X.t$, variable X can be instantiated by any type, including $\forall X.t$.
- This breaks the recursive definition over types used in the classic normalization proof for simple types (Tait, 1967).

(More details in the lecture on logical relations.)

A first success for Curry-Howard

Work on System F show the Curry-Howard at work, in two different ways:

- A priori: Girard designs System F by searching for the typed language that corresponds with an existing logic. By proving normalization of the language, he shows a difficult conjecture in logic.
- A posteriori: Girard and Reynolds reach the same type system, one while trying to solve logic problems, the other while trying to solve programming problems.

II

Other dimensions of polymorphism: higher order types, dependent types

Type constructors

In examples we used type expressions `list`(t), where `list` is a type constructor with one parameter, that is, a function from types to types: $t \mapsto \texttt{list}(t)$.

Caml, Haskell, and all typed functional languages provide ways to define type constructors having zero, one, or several parameters:

```
type nat = O | S of nat
type 'a list = Nil | Cons of 'a * 'a list
type ('a, 'b) alist = ('a * 'b) list
```

Abstracting over a type constructor

It makes sense to abstract (via a Λ) over a type constructor and not just over a type.

Example (The abstract type of stacks)

Stacks containing elements of type t are modeled by a type constructor STK with one parameter, and by the operations

```
empty: ∀t. STK(t)
push:  ∀t. t -> STK(t) -> STK(t)
pop:   ∀t. STK(t) -> option (t * STK(t))
```

To write clients of this abstract type in the style of Reynolds, we need to abstract over the STK type constructor:

```
let use_stack =
  ΛSTK. λops: { empty: ∀t. STK(t); ... }.
```

System F_ω

F_ω (Girard, 1972) is an extension of System F where type expressions include functions from types to types (i.e. type constructor), and where terms can abstract (using Λ) over all kinds of types.

Terms:	$M, N ::= x \mid \lambda x{:}t.\, M \mid M\; N$	
	$\mid \Lambda X :: k.\, M$	abstraction over type X of kind k
	$\mid M[t]$	instantiation at type t
Types:	$t ::= X \mid t_1 \rightarrow t_2$	
	$\mid \forall X :: k.\, t$	
	$\mid \lambda X :: k.\, t$	type constructor with parameter X
	$\mid t_1\; t_2$	type constructor application
Kinds:	$k ::= *$	regular type
	$\mid k_1 \Rightarrow k_2$	type constructor

(For instance, the `list` type constructor has kind $* \Rightarrow *$)

Kinds and types

Kinds classify types and rule out ill-formed type expressions just like types classify terms.

Judgment $\Gamma \vdash t :: k$ (type t is well formed and has kind k):

$$\Gamma_1, X :: k, \Gamma_2 \vdash X :: k$$

$$\frac{\Gamma, X :: k \vdash t :: k'}{\Gamma \vdash \lambda X :: k.\, t :: k \Rightarrow k'} \qquad \frac{\Gamma \vdash t_1 :: k \Rightarrow k' \qquad \Gamma \vdash t_2 :: k}{\Gamma \vdash t_1\, t_2 :: k'}$$

$$\frac{\Gamma \vdash t_1 :: * \qquad \Gamma \vdash t_2 :: *}{\Gamma \vdash t_1 \rightarrow t_2 :: *} \qquad \frac{\Gamma, X :: k \vdash t :: *}{\Gamma \vdash \forall X :: k.t :: *}$$

This is simply-typed λ-calculus “lifted one level up” and applied to types and to kinds.

The extended Curry-Howard correspondence

System F_ω	intuitionistic logic
kind	type
type	proposition (possibly with parameters)
term	proof
reductions over types	the conversion rule (a deduction rule)
reductions over terms	cut elimination

We recover Church (1932, 1933) lambda-calculus at the level of types: they are lambda-terms that represent propositions.
Type-level β-reduction is Church's β-conversion.

One level "below", we have terms that represent proofs for these propositions.

Terms and types

Same rules as in System F, plus checks over kinds and a rule to handle type conversion:

$$\Gamma_1, x : t, \Gamma_2 \vdash x : t$$

$$\frac{\Gamma \vdash t :: * \qquad \Gamma, x : t \vdash M : t'}{\Gamma \vdash \lambda x{:}t.\, M : t \to t'}$$

$$\frac{\Gamma \vdash M : t \to t' \qquad \Gamma \vdash N : t}{\Gamma \vdash M\, N : t'}$$

$$\frac{\Gamma, X :: k \vdash M : t}{\Gamma \vdash \Lambda X :: k.\, M : \forall X :: k.\, t}$$

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

$$\frac{\Gamma \vdash M : t \qquad t =_\beta t'}{\Gamma \vdash M : t'}\ \text{(conv)}$$

Four forms of parameterization

So far we've seen three parameterization mechanisms:

- a term parameterized by a term: $\lambda x. M$
  (= functions from terms to terms)
- a term parameterized by a type: $\Lambda X. M$
  (= polymorphism, system F)
- a type parameterized by a type: $\lambda X. t$
  (= type constructor, system F_ω)

The fourth form is very interesting as well:

- a type parameterized by a term
  (= dependent types)

Dependent types in logic

Dependent types are analogous to predicates.

Example: the predicates `even` and `odd` over natural numbers n

$$\mathtt{even}(n) \stackrel{def}{=} n \bmod 2 = 0 \qquad \mathtt{odd}(n) \stackrel{def}{=} n \bmod 2 = 1$$

These are type constructors taking as argument a natural number n instead of a type (as in F_ω). The theorem

If n is even, $n + 1$ is odd

corresponds with the type

$$\forall n : \mathbb{N}.\ \mathtt{even}(n) \rightarrow \mathtt{odd}(n + 1)$$

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.

In other words, the `array` type constructor takes two parameters, a type t and a term N, and produces the type $\mathtt{array}(t, N)$.

Lifting the restriction that N is a constant expression, and allowing ourselves to quantify over this N, we can give very precise dependent types to array operations:

$$\mathtt{concat} : \forall t. \forall N_1. \forall N_2.\ \mathtt{array}(t, N_1) \rightarrow \mathtt{array}(t, N_2) \rightarrow \mathtt{array}(t, N_1 + N_2)$$

($\rightarrow$ P. E. Dagand's seminar in week 2.)

Dependent types for programs

Quiz: are the following array types compatible?

$$
\begin{aligned}
\mathtt{array}(t, 6) &\quad \text{and} \quad \mathtt{array}(t, 5+1) \\
\mathtt{array}(t, N+M) &\quad \text{and} \quad \mathtt{array}(t, M+N) \\
\mathtt{array}(t, \mathtt{fact}(N)/\mathtt{fact}(N-1)) &\quad \text{and} \quad \mathtt{array}(t, N) \\
\mathtt{array}(\mathtt{random}(10)) &\quad \text{and} \quad \mathtt{array}(6)
\end{aligned}
$$

We would like the types $\mathtt{array}(t, N)$ and $\mathtt{array}(t, N')$ to be convertible as soon as the expressions N and N' denote the same natural number. However, this is

- undecidable if the expression language is rich enough;
- undefined if the expression language contains effects.

Dependent types and conversion

$$\frac{\Gamma \vdash M : t \qquad t \approx t'}{\Gamma \vdash M : t'} \text{(conv)}$$

The notion of convertibility $\approx$ is not just β-conversion (as in F_ω) but includes other equivalences to deal with terms that can occur in the types t and t'.

F* uses automatic theorem proving to show that t and t' are equal modulo theories (arithmetic, bit vectors, etc).

Coq and Agda use exclusively computation rules as equivalences, hence $\mathtt{array}(t, 5+1) \approx \mathtt{array}(t, 1+5)$ but $\mathtt{array}(t, N+1) \not\approx \mathtt{array}(t, 1+N)$.

To go further, we need to build and use retyping functions such as $f : \forall t. \forall N. \mathtt{array}(t, N+1) \to \mathtt{array}(t, 1+N)$

Automath and dependent types

Automath (N. de Bruijn, 1968): the first proof assistant; the birth of the idea that computers can check proofs developed by humans.

Automath has "proof = lambda-term" but not "proposition = type":

Types: $t ::= \texttt{bool}$ — the type of propositions
$\quad\mid T(b)$ — the dependent type of proofs of $b : \texttt{bool}$
$\quad\mid t_1 \to t_2$

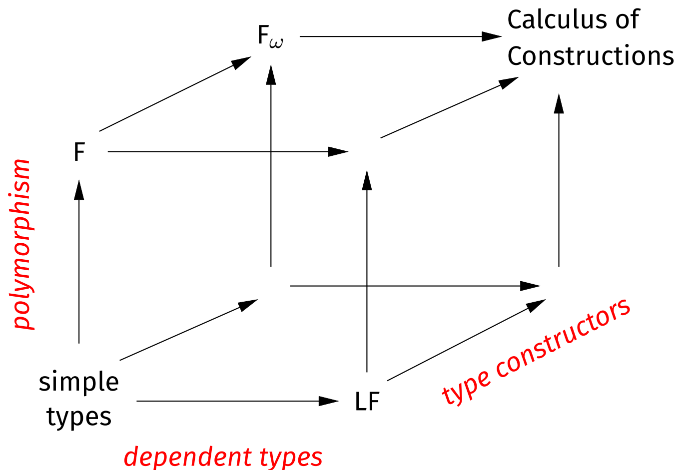
Logical connectives live within the term level and do not need to be reflected at the type level

$$\wedge : \texttt{bool} \to \texttt{bool} \to \texttt{bool} \qquad \forall : (\texttt{bool} \to \texttt{bool}) \to \texttt{bool}$$

This approach is used in the Edinburgh Logical Framework (LF), a "meta"-formalism to define logics and languages.

Barendregt's lambda-cube

Three directions to extend simply-typed lambda-calculus:



III

Pure Type Systems

Towards unification

Between 1971 and 1985 several formal systems appear that are

- directly inspired by Howard's notes;
- usable as intuitionistic logics and as typed lambda-calculi;
- combining polymorphism, type constructors, and dependent types;
- more expressive than F, F_ω and LF.

History

Martin-Löf's intuitionistic type theory (MLTT):

- A constructive logic based on dependent types.
- A few type constructors, including Π (dependent function types) and Σ (dependent product types).
- 1971–1979: several successive versions.

The Calculus of Constructions (CC):

- Coquand & Huet, 1986–1988: F_ω + dependent types.
- Many extensions during the development of Coq.

The Pure Type Systems (PTS):

- Berardi, 1988; Terlouw, 1989; Barendregt, 1992: a unified reformulation.

One lambda, one Pi, many terms

In F_ω:

- Three syntactic categories: terms, types, kinds.
- Three lambdas: $\lambda x : t. M$ / $\Lambda X :: k. M$ / $\lambda x :: k. t$
- Three function types: $t_1 \rightarrow t_2$ / $\forall X :: k. t$ / $k_1 \Rightarrow k_2$.

In PTS:

- Same syntax for terms, types, and kinds.
- A single lambda $\lambda x : A. B$ for all abstractions.
- A single type $\Pi x : A. B$ (dependent function type) for all lambdas.
- Universes to distinguish terms, types, kinds, etc.

Pure Type Systems: syntax

Universe: $U \in \mathcal{U}$

Terms, types:

$$
\begin{array}{rcll}
A, B, C & ::= & x & \text{variables} \\
& | & \lambda x : A.\, B & \text{abstractions} \\
& | & A\ B & \text{applications} \\
& | & \Pi x : A.\, B & \text{function type} \\
& | & U & \text{universe name}
\end{array}
$$

Notation: $A \to B \stackrel{def}{=} \Pi x : A.\, B$ if x not free in B.

Example: encoding F_ω

Two universes: $*$ for types, $\square$ for kinds.

Terms:	$\lambda x : t.\ M$	$\equiv$	$\lambda x : t.\ M$	with $t : *$
	$\Lambda X :: k.\ M$	$\equiv$	$\lambda X : k.\ M$	with $k : \square$
Types:	$t_1 \rightarrow t_2$	$\equiv$	$t_1 \rightarrow t_2$	with $t_1, t_2 : *$
	$\forall X :: k.\ t$	$\equiv$	$\Pi X : k.\ t$	with $k : \square$, $t : *$
Kinds:	$*$	$\equiv$	$*$	with $* : \square$
	$k_1 \Rightarrow k_2$	$\equiv$	$k_1 \rightarrow k_2$	with $k_1, k_2 : \square$

Pure Type Systems: typing

$$\frac{(U, U') \in \mathcal{A}}{\emptyset \vdash U : U'} \text{ (ax)} \qquad \frac{\Gamma \vdash A : U}{\Gamma, x : A \vdash x : A} \text{ (var)} \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : U}{\Gamma, x : C \vdash A : B} \text{ (wk)}$$

$$\frac{\Gamma \vdash A : U_1 \quad \Gamma, x : A \vdash B : U_2 \quad (U_1, U_2, U_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A.B : U_3} \text{ (pi)}$$

$$\frac{\Gamma, x : A \vdash B : C \quad \Gamma \vdash \Pi x : A.C : U}{\Gamma \vdash \lambda x.A.\, B : \Pi x.A.\, C} \text{ (abstr)}$$

$$\frac{\Gamma \vdash f : \Pi x : A.\, B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\{x \leftarrow a\}} \text{ (app)} \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : U \quad B =_\beta B'}{\Gamma \vdash A : B'} \text{ (conv)}$$

Controlling expressiveness via universes

A specific PTS is obtained by defining

- the set $\mathcal{U}$ of universes;
- the relation $\mathcal{A} \subseteq \mathcal{U} \times \mathcal{U}$ that states which universe belong to which universe;
- the relation $\mathcal{R} \subseteq \mathcal{U} \times \mathcal{U} \times \mathcal{U}$ that states what can be parameterized over what.

$$\frac{(U, U') \in \mathcal{A}}{\emptyset \vdash U : U'} \text{(ax)} \qquad \frac{\Gamma \vdash A : U_1 \quad \Gamma, x : A \vdash B : U_2 \quad (U_1, U_2, U_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A.B : U_3} \text{(pi)}$$

Barendregt's cube

The systems from Barendregt's cube have $\mathcal{U} = \{*, \Box\}$, where

- $*$ the universe of types;
- $\Box$ the universe of sorts (the types of types);
- $\mathcal{A} = \{ (*, \Box) \}$, that is, $* : \Box$

The 3 dimensions of the cube are controlled by relation $\mathcal{R}$:

Feature	*elt. of* $\mathcal{R}$	*supported functions*
simple types	$(*, *, *)$	from terms to terms
polymorphism	$(\Box, *, *)$	from types to terms
type constructors	$(\Box, \Box, \Box)$	from types to types
dependent types	$(*, \Box, \Box)$	from terms to types

Playing with universes

Additional universes:

E.g. a variant of CC splits $*$ in two universes, `Prop` and `Set`.
`Prop`: the universe of logical propositions
`Set`: the universe of data types and computable functions.

Infinitely many universes: $*_0 : *_1 : \cdots : *_n : *_{n+1} : \cdots$

Russell-style stratification.

A single universe: $* : *$ (read: "Type colon Type")

Danger warning! Logical paradox!!

The Burali-Forti paradox (1897)

Assume that the set O of all ordinals exists.

The order $<_{ord}$ between ordinals is a well-order over O.

Hence O ordered by $<_{ord}$ is an ordinal.

Besides, O is strictly greater than any ordinal.

Hence $O <_{ord} O$ and a contradiction.

Burali-Forti in Coq with `Type : Type`

Let `U` be a universe such that we can define the following type (the type of well-ordered types):

```
Record ord : U := mkord {
  carrier: U;
  rel: carrier -> carrier -> Prop;
  rel_wf: well_founded rel
}.
```

This is the strict order over this type of ordinals:

```
Record emb (A B: ord) : Prop := {
  f: carrier A -> carrier B;
  f_inj: forall x y, rel A x y -> rel B (f x) (f y);
  sup: B;
  f_sup: forall x, rel B (f x) sup
}.
```

Burali-Forti in Coq with Type : Type

This order is well founded:

```
Lemma wf_emb: well_founded emb.
```

We can therefore define the ordinal of all ordinals:

```
Definition Omega: ord := mkord ord emb wf_emb.
```

If this definition is accepted by Coq, we have a paradox:

```
Lemma emb_Omega_Omega: emb Omega Omega.

Lemma paradox: False.  (* emb_Omega_Omega contradicts wf_emb *)
```

Paradox avoided (almost)

Universe `U`	Coq options	what happens
$\texttt{Type}_i$	default	`ord` : $\texttt{Type}_{i+1}$ bur not `ord` : $\texttt{Type}_i$ `Omega` not definable
`Type`	`-type-in-type`	a proof of `False`
`Prop`	default	projection `carrier: ord -> Prop` not definable
`Set`	`-impredicative-set`	projection `carrier: ord -> Set` not definable

Girard's paradox (1972)

System U: an extension of F_ω with polymorphism over kinds.

- Three universes: $* : \Box : \triangle$
- Formation rules: those of F_ω plus $(\triangle, *, *)$ and $(\triangle, \Box, \Box)$.

Makes it possible to write e.g.

$$\lambda k : \Box.\ \lambda\alpha : k \to k.\ \lambda\beta : k.\ \alpha\ (\alpha\ \beta) \quad : \quad \Pi k : \Box.\ (k \to k) \to (k \to k)$$

Girard's paradox: an encoding of the Burali-Forti paradox in System U, more subtle than the previous encoding with `Type:Type`.

Corollary: this shows that MLTT 1971 (with `Type:Type`) is inconsistent as a logic. Later versions of MLTT are stratified using universes.

(See also `https://github.com/coq-contribs/paradoxes`)

Predicativity and impredicativity

Russell's "vicious circle principle"

> *Whatever involves all of a collection must not be one of the collection.*

Predicativity: $\Pi X : U.\, A$ lives in a universe "above" U.

In other words: no "vicious circle" in the sense of Russell.

Example: $(\Pi X : \mathtt{Type}_n.\, A) : \mathtt{Type}_{n+1}$ in MLTT, Coq, Agda.

Impredicativity: $\Pi X : U.\, A$ lives in universe U or below.

In other words: it is possible to quantify "over one self".

Example: $(\forall X{:} * .\, t) : *$ in System F and F_ω

Example: `(forall (P: Prop), Q) : Prop` in Coq.

Cumulativity and universe polymorphism

How to make a definition usable in several universes?

Cumulativity: a form of subtyping between universes.

$$\frac{\Gamma \vdash A : \texttt{Type}_i}{\Gamma \vdash A : \texttt{Type}_{i+1}}$$

Polymorphism: add universe variables *i* and ways to quantify universally over them, For instance in Coq:

```
Polymorphic Definition idtype@{i} : Type@{i+1} :=
    forall A: Type@{i}, A -> A.
Polymorphic Definition identity@{i} : idtype@{i} :=
    fun (A: Type@{i}) (x: A) => x.
```

Symbolic expressions for universes: $u ::= U \mid i \mid u + 1 \mid \max(u_1, u_2)$

Application: Coq and Agda

	Coq	Agda
Universes	$\mathtt{Prop}$ $\mathtt{Set} = \mathtt{Type}_0$ $\mathtt{Type}_1, \ldots, \mathtt{Type}_n, \ldots$	$\mathtt{Set} = \mathtt{Set}_0$ $\mathtt{Set}_1, \ldots, \mathtt{Set}_n, \ldots$
Impredicative?	$\mathtt{Prop}$ (and, earlier, $\mathtt{Set}$)	no
$\mathcal{R}$	$(\mathtt{Type}_i, \mathtt{Type}_j, \mathtt{Type}_{\max(i,j)})$ $(U, \mathtt{Prop}, \mathtt{Prop})$	$(\mathtt{Set}_i, \mathtt{Set}_j, \mathtt{Set}_{\max(i,j)})$
Cumulativity?	yes	no
Polymorphism?	yes (recently)	yes (recently)

IV

Summary

Curry-Howard, continued

Presented today:

- Explorations of many extensions:
  polymorphism, type constructors, dependent types,
  `Type : Type`, universes, ...
- Convergence towards *Pure Type Systems*, a family of typed lambda-calculi usable as programming languages and as intuitionistic logics...
- ...but centered on functions and their Π-types, that is, on the logic connectors $\Rightarrow$ and $\forall$.

Next week:

- Other data types? Other logical connectives?
- General mechanisms to define data types and logical connectives: inductive types and inductive predicates.

V

Further reading

Further reading

- Jean-Yves Girard. *The blind spot: Lectures on logic.* European Mathematical Society, 2011. Chapter 6.
- Benjamin Pierce. *Types and Programming Languages.* MIT Press, 2002. Chapters 22, 23, 24, 29, 30.
- Morten Heine Sørensen, Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism.* 1998. Chapters 11 to 14.
  `https://disi.unitn.it/~bernardi/RSISE11/Papers/curry-howard.pdf`