Programming = proving?
The Curry-Howard correspondence today

Third lecture

Weapons of mass construction: inductive types, inductive predicates

Xavier Leroy

Collège de France

2018-11-28

COLLÈGE
DE FRANCE
1530

Today's lecture

The previous lecture concentrated on functions and the $\Rightarrow, \forall$ fragment of the logic. Let's go beyond!

In programming languages: which mechanisms allow us to define and work with data structures beyond base types (numbers, etc) and functions?

In logics: how to treat all connectives? not just $\Rightarrow, \forall$ but also $\wedge$, $\vee$, $\bot$, $\exists$?
how to define objects manipulated by the logic, such as natural numbers?
and predicates over these objects?

Let's try to answer both questions at the same time, in the spirit of Curry-Howard!

I

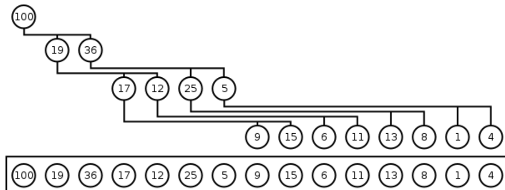
Data structures
in programming languages

Base types and arrays

As early as in FORTRAN, and in almost all later languages:

- Base types: numbers (integers, “reals” (floating-point numbers), “complex”), characters, ...
- Arrays of one or several dimensions, indexed by integers.

Indirections and linked data structures can be encoded using array indices, as in the example of heaps:



Records

COBOL introduces the concept of *records*, with named fields, having different types, and possibly nested.

```
01  Transaction.                              *> record
       05  Number      PIC 9(10).             *> field
       05  Date.                              *> sub-record
             10  Year    PIC 9(4).            *> field
             10  Month   PIC 99.              *> field
             10  Day     PIC 99.              *> field
```

Addresses, references, pointers

Assembly code often manipulates memory addresses for data blocks that do not fit in a register.

This trick enters high-level programming languages under the names of references in ALGOL-W (1966) and of pointers in C, Pascal, etc.

A pointer can be "null" (Hoare: *my billion-dollar mistake*).

Records + pointers $\Rightarrow$ somewhat natural encoding of linked data structures (lists, trees, ...).

```
struct list {
  int head;
  struct list * tail;
};
```

```
struct tree {
  char * name;
  struct tree * left, * right;
}
```

Unions

If records are viewed as the “and” of several types, unions are the “or” of several types. They appear in Algol 68:

```
mode node = union (real, int, string);
node n := "1234";
case n in
  (real r):   print(("real:", r)),
  (int i):    print(("int:", i)),
  (string s): print(("string:", s))
  out         print(("?:", n))
esac
```

This is a discriminated union: the value of type `node` records which of the three cases `real`, `int` and `string` applies. This is unlike unions in C:

```
union u { float f; int i; char * s; };
```

Unions

Pascal combines records and discriminated unions:

```
type
  exprKind = (VAR, CONST, SUM, PROD);
  expr = record
           (* common part *)
           size: integer;
           (* variant part *)
           case kind : exprKind of
             VAR:       (varName: string);
             CONST:     (constVal: integer);
             SUM, PROD: (left, right: ^expr)
         end
```

A universal type: S-expressions

Instead of leaving data structure definitions to the programmer, Lisp provides a ”universal” type, S-expressions, which expresses easily a great many data structures.

$$
\begin{aligned}
\mathit{sexp} &::= \mathit{atom} \mid (\mathit{sexp}\ .\ \mathit{sexp}) \\
\mathit{atom} &::= \mathit{number} \mid \mathit{symbol} \mid \texttt{nil}
\end{aligned}
$$

Canonical encodings for many data structures:

$$
\begin{array}{rrcl}
\text{Lists:} & (x_1\ \ldots\ x_n) & \equiv & (x_1\ .\ (\cdots (x_n\ .\ \texttt{nil}))) \\
\text{Terms:} & \mathit{cstr}0 & \equiv & \mathit{cstr}0 \\
 & \mathit{cstr}(x_1, \ldots, x_n) & \equiv & (\mathit{cstr}\ x_1\ \cdots\ x_n) \\
\text{Functions:} & \lambda x.M & \equiv & (\texttt{lambda}\ (x)\ M)
\end{array}
$$

A universal type: Prolog terms

Prolog provides another universal type, inspired by term algebras as used in logic and in rewriting theory:

$$\mathit{term} ::= \mathit{number} \mid \mathit{atom} \mid \mathit{Variable} \mid \mathit{atom}(\mathit{term}, \ldots, \mathit{term})$$

Atoms represent the constructors (constant or non-constant) of the term algebras.

Cartesian product and binary sum

LCF-ML, the ancestor of the ML family, provides no mechanism to declare record or union types. It predefines two types: Cartesian product (two-field record) and binary sum (discriminated union of two types):

Types:		
	$t ::= \texttt{int} \mid \texttt{bool} \mid \ldots$	base types
	$\mid t_1 \rightarrow t_2$	function types
	$\mid t_1 \# t_2$	products: pairs of a t_1 and a t_2
	$\mid t_1 + t_2$	sums: either a t_1 or a t_2

Note: via Curry-Howard, product types correspond to conjunction $\wedge$ and sum types to disjunction $\vee$.

Recursion in LCF-ML

A type abstraction mechanism allows us to define recursive types as abstract types equipped with constructor functions and destructor functions.

Example: binary trees with values of type * at leaves.

```
absrectype * tree = * + * tree # * tree
  with leaf n = abstree(inl n)
   and node (t1, t2) = abstree(inr(t1, t2))
   and isleaf t = isl(reptree t)
   and leafval t = outl(reptree t) ? failwith ‘leafval‘
   and leftchild t = fst(outr(reptree t) ? failwith ‘leftchild‘
   and rightchild t = snd(outr(reptree t) ? failwith ‘leftchild‘
```

Algebraic types

In 1980 the purely functional language HOPE (Burstall, MacQueen, Sannella) introduced the modern presentation of algebraic types. Later reused by Milner and by Cousineau for integration in CAML and in Standard ML.

```
type expr =
  | Const of int
  | Infinity
  | Sum of expr * expr
  | Prod of expr * expr
```

A sum of product types that is recursive.

Each case of the sum is discriminated by its constructor (`Const`, `Infinity`, `Sum`, `Prod`)

Each constructor carries zero, one or several arguments.

An argument can be of the type being defined (recursion).

Working with an algebraic type

Building values of the type: via constructor application.

```
let e = Sum(Const 1, Prod (Const 2, Infinity))
```

Using values of the type: by pattern matching.

```
let rec floatval e =
  match e with
  | Const n -> Float.of_int n
  | Infinity -> Float.infinity
  | Sum(e1, e2) -> floatval e1 +. floatval e2
  | Prod(e1, e2) -> floatval e2 *. floatval e2
```

II

Inductive types

Inductive types

A variant of algebraic types, compatible with type theory.

- Extension: dependencies in the types of constructors and in the kinds of inductive types.
- Restriction: restricted recursion, to preserve strong normalization and logical consistency.

Inductive types were introduced by Paulin-Mohring and Pfenning in 1989 as an extension of the Calculus of Constructions.

They are primitive notions in Coq and in Agda, encoded in Isabelle/HOL.

Inductive types

Same idea “sum of products + recursion” but a different vision:

universe containing the type

```
Inductive expr : Type :=
  | Const: nat -> expr
  | Infinity: expr
  | Sum: expr -> expr -> expr
  | Prod: expr -> expr -> expr
```

constructors ↑ with their types ↑

Each constructor is a constant of type `expr` or a function returning an `expr`.

The type `expr` is the set of values generated by these constants and functions. (That is, the smallest set containing the constants and stable by the functions.)

Familiar data types

```
Inductive bool : Type :=
 | true  : bool
 | false : bool.

Inductive unit : Type :=
 | tt : unit.

Inductive empty : Type := .    (* empty type, zero constructors *)

Inductive nat : Type :=        (* Peano integers *)
  | O : nat
  | S : nat -> nat.
```

Inductive types and functions

Unlike ordinary term algebras, inductive types can contain functions as arguments of constructors.

Example: Brouwer's ordinals.

```
Inductive ord: Type :=
  | Zero: ord
  | Succ: ord -> ord
  | Limit: (nat -> ord) -> ord.
```

Viewed as a tree, a value of inductive type is not always finite: a node can have infinitely many siblings (like `Limit` above). However, there are no infinite paths.

Parameterized inductive types

An inductive type can be parameterized by types (or values) and have polymorphic constructors.

parameter

```
Inductive list (A: Type) : Type :=        (* lists *)
  | nil : list A
  | cons : A -> list A -> list A.

Inductive prod (A: Type) (B: Type) : Type :=
  | pair : A -> B -> prod A B.            (* Cartesian product *)

Inductive sum (A: Type) (B: Type) : Type :=
  | inl : A -> sum A B                    (* binary sum *)
  | inr : B -> sum A B.
```

Familiar logical connectives

By Curry-Howard magic, these inductive types can also be used as logical connectives: Cartesian product is conjunction; binary sum is disjunction; etc.

Instead of using the same inductives in data types and in logical formulas, Coq prefers to define them twice, once in the `Prop` universe and once in the `Type` universe.

Familiar logical connectives

```
Inductive and (A: Prop) (B: Prop) : Prop :=  (* conjunction *)
  | conj : A -> B -> and A B.

Inductive or (A: Prop) (B: Prop) : Prop :=   (* disjunction *)
  | or_introl : A -> or A B
  | or_intror : B -> or A B.

Inductive True : Prop :=                     (* triviality *)
  | I : True.

Inductive False : Prop := .                  (* absurdity *)
                                   (* zero constructors! *)
```

`A /\ B` is a notation for `and A B`
`A \/ B` is a notation for `or A B`.

Dependently-typed constructors

The constructors of a type can have dependent types: the type of an argument can depend on the value of a preceding argument.

Example: the type of dependent pairs, that is, the type $\Sigma x : A.\ B(x)$ in MLTT, but also the quantifier $\exists x : A.\ P(x)$.

type family indexed by `a:A`

```
Inductive sigma (A: Type) (B: A -> Type) : Type :=
  | exist: forall (a: A), B a -> sigma A B.
```

first argument

second argument (dependent)

Three shades of Sigma

Owing to its `Prop`/`Type` distinction, Coq ends up with three variants of Σ types, all defined as inductives + notations:

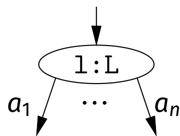
Notation	result	`B(x)`	
`exists x:A, B(x)`	`Prop`	`Prop`	"there exists" quantifier
`{ x: A | B(x) }`	`Type`	`Prop`	subset type: an `x:A` with a proof of `B(x)`
`{ x: A & B(x) }`	`Type`	`Type`	dependent pair: an `x:A` and a `B(x)`

The W type

Another component of MLTT definable as an inductive type is the type W of well-founded trees:

```
Inductive W (L: Type) (A: L -> Type) : Type :=
  | Node: forall (lbl: L), (A lbl -> W L A) -> W L A.
```

`L` is the type of labels carried by each node. A node labeled `l:L` has n subtrees, where n is the number of elements of type `A l`, each subtree being identified by an element of `A l`.



where $a_1, \ldots, a_n$ are the elements of type `A l`

The W type

```
Inductive W (L: Type) (A: L -> Type) : Type :=
  | Node: forall (lbl: L), (A lbl -> W L A) -> W L A.
```

Examples: assume defined the types `empty`, `unit` and `bool` with 0, 1 and 2 elements respectively. Here is type `nat`:

```
W bool (fun b => match b with false => empty
                            | true  => unit   end)
```

The type of lists of `A`:

```
W (option A) (fun l => match l with None => empty
                                  | Some _ => unit  end)
```

The type of binary trees with `A` at leaves and `B` at nodes:

```
W (A + B) (fun l => match l with inl _ => empty
                               | inr _ => bool  end)
```

Elimination of inductive types by pattern matching

As with algebraic types, a pattern-matching construct performs elimination (case analysis) over inductive types.

```
Definition not (b: bool) :=
  match b with
  | true => false
  | false => true
  end.
```

```
Definition pred (n: nat) :=
  match n with
  | O => O
  | S p => p
  end.
```

Elimination of inductive types by pattern matching

Via Curry-Howard, pattern matching corresponds to a proof by case analysis.

Example: a property is true for all `b: bool` if it is true for `b = true` and for `b = false`.

```
Theorem bool_cases: forall (P: bool -> Prop),
                      P true -> P false -> forall b, P b.

Proof fun (P: bool -> Prop) (iftrue: P true) (iffalse: P false)
          (b: bool) =>
        match b with true => iftrue | false => iffalse end.
```

Elimination of inductive types by pattern matching

The general shape of a pattern matching for an inductive with n constructors $C_1, \ldots, C_n$ of arities $a_1, \ldots, a_n$ is:

$$\begin{array}{l}\texttt{match}\ e\ \texttt{with}\\ \mid\ C_1\ x_1^1\ \ldots\ x_1^{a_1}\ \texttt{=>}\ b_1\\ \mid\ \ldots\\ \mid\ C_n\ x_n^1\ \ldots\ x_n^{a_n}\ \texttt{=>}\ b_n\\ \texttt{end}\end{array}$$

The corresponding reduction rule:

$$\texttt{match}\ C_i\ e^1\ \ldots\ e^{a_i}\ \texttt{with}\ \ldots\ \texttt{end}\ \rightarrow\ b_i\{x_i^1 \leftarrow e^1, \ldots, x_i^{a_i} \leftarrow e^{a_i}\}$$

Example of natural numbers:

```
match O   with O => a | S p => b end  → a
match S n with O => a | S p => b end  → b{p←n}
```

Elimination of inductive types by recursors

An alternative to the pattern-matching construct (`match`) is for every `Inductive` definition to produce a recursor function for the type, combining case analysis and recursive computation.

For a type T with n constructors, this recursor has shape

$$T_\texttt{rec}\ \mathit{value}\ \mathit{case}_1\ \ldots\ \mathit{case}_n$$

For example, for Booleans, options, and natural numbers:

```
bool_rec true  a b  →  a
bool_rec false a b  →  b

option_rec None     a b  →  a
option_rec (Some x) a b  →  b x

nat_rec O     a b  →  a
nat_rec (S n) a b  →  b (nat_rec n a b)   (* <- recursion *)
```

Iteration

Recursors provide a simple form of recursion called “iteration”, where recursive calls are always done and only done over direct arguments of recursive constructors.

For example, the “double” function over Peano integers is an iteration, but Fibonacci’s function is not:

```
let rec double n =       (* nat_rec n O (fun x => S (S x)) *)
  match n with O -> O | S p -> S (S (double p))

let rec fib n =
  match n with O -> S O
             | S p -> match p with O -> S O
                                 | S q -> fib p + fib q
```

Exercise: define by iteration the function `fib’ n` $\stackrel{def}{=}$ `(fib n, fib (S n))`.

Pattern matching and recursion

To operate over inductive types, pattern matching (`match`) must be combined with a mechanism to define recursive functions (`let rec` in Caml, `Fixpoint` in Coq).

```
Fixpoint double (n: nat) : nat :=
  match n with O => O | S p => S (S (double p)) end.

Fixpoint fib (n: nat) : nat :=
  match n with O => S O
             | S p => match p with O => S O
                                 | S q => fib p + fib q end
  end.
```

A guard condition ensures termination. Typically, recursive calls must be performed over a strict sub-term of the argument (structural recursion).

Termination and consistency

Almost always, if we can define an expression that reduces infinitely, we can give it an empty type (`False` ou `empty`), which makes it possible to "prove" any proposition `P`.

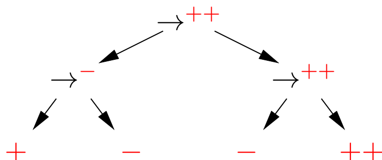
✘ `Fixpoint loop (n: nat) : False := loop (S n).`

✘
```
Theorem inconsistency (P: Prop) : P :=
  match loop O with end.
```

Therefore, it is impossible to have general recursion in a language such as Coq or Agda.

Positivity

For related reasons, an inductive type must appear in strictly positive position in the types of the arguments of its constructors.



$++$	strictly positive	to the left of zero arrows
$+$	positive	to the left of $2n$ arrows
$-$	negative	to the left of $2n+1$ arrows

Positivity

A negative occurrence of an inductive predicate immediately leads to a contradiction:

```
✘ Inductive P : Prop := Pintro : (P -> False) -> P.

Lemma paradox: P <-> ~P.
```

A negative occurrence of an inductive type gives pure lambda-calculus and, therefore, diverging computations:

```
✘ Inductive lam : Type := Lam: (lam -> lam) -> lam.

Definition app (a b: lam) : lam := match a with Lam f => f b end.
Definition delta : lam := Lam (fun x => app x x).
Definition omega : lam := app delta delta.   (* diverges! *)
```

Positivity

A “doubly negative” (positive but not strictly) occurrence leads to a paradox.

✘ `Inductive A : Type := Aintro : ((A -> Prop) -> Prop) -> A.`

```
Definition f (x: A -> Prop) : A := Aintro (fun y => x = y).
```

We show that `f` is an injection from `A -> Prop` (= the subsets of `A`) in `A`, which is impossible by “cardinality”.

A Cantor-style diagonalization leads to a contradiction.

(Coquand, *A new paradox in type theory*, Studies in Logic and the Foundations of Mathematics 134, 1995).

III

Inductive families

Over what does recursion apply?

Parameterized inductive type:

- Function from parameters to a recursive type.
- Parameters are unchanged in the types of constructors.

```
Inductive list (A:Type): Type :=
| nil: list A
| cons: A -> list A -> list A.
                   ↖         ↗
        (* same parameter *)
```

Inductive family:

- Recursive function from parameters to a type.
- Parameters can assume different values in the types of constructors.

```
Inductive t: Type -> Type :=
  | A: t nat
  | B: t bool.
          ↑
 (* different parameters *)
```

Examples of inductive families

Inductive family: the type `fin n` of natural numbers between 0 and `n`.

```
Inductive fin: nat -> Type :=
  | Zero : forall (n: nat), fin n
  | Succ : forall (n: nat), fin n -> fin (S n).
```

Inductive family with one parameter:
the type `vec A n` of lists of `A` having length `n`.

```
Inductive vec (A: Type): nat -> Type :=
  | nil: vec A O
  | cons: forall (n: nat), A -> vec A n -> vec A (S n).
```

Exercise: define a safe function to access the n-th element.

```
nth: forall (A: Type) (n: nat),  vec A (S n) -> fin n -> A.
```

Inductive predicates

Inductive families are very useful to define predicates by inference rules.

- Predicate P of n arguments of types $A_1, \ldots A_n$
  $\Rightarrow \texttt{Inductive}\ P : A_1 \rightarrow \cdots \rightarrow A_n \rightarrow \texttt{Prop}$
- Axiom $\Rightarrow$ constant constructor
- Inference rule with k premises $\Rightarrow$ constructor with k arguments.

Structural recursion over P provides us with a powerful proof principle: by induction on the structure of a derivation and by case analysis on the last rule used.

Example: the predicate “being even” on integers

$$\mathtt{even}(0)$$

$$\frac{\mathtt{even}(n)}{\mathtt{even}(S(S(n)))}$$

```
Inductive even: nat -> Prop :=
  | even_0:
      even 0
  | even_S: forall n,
      even n ->
      even (S (S n)).
```

Example: representing a logic

Inference rules:

$$\Gamma \vdash \top \;(\top\mathrm{I}) \qquad \Gamma_1, P, \Gamma_2 \vdash P \;(\mathrm{Ax})$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\;(\Rightarrow\mathrm{I}) \qquad \frac{\Gamma \vdash P \Rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q}\;(\Rightarrow\mathrm{E}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash P}\;(\bot\mathrm{E})$$

Representing logic formulas:

```
Inductive formula : Type :=
  | True : formula                           (* ⊤, triviality *)
  | False: formula                           (* ⊥, absurdity *)
  | Imp  : formula -> formula -> formula.    (* ⇒, implication *)
```

Example: representing a logic

Transcribing the deduction rules:

```
Inductive sequent : list formula -> formula -> Prop :=
  | TrueI: forall G,
      sequent G True
  | Ax: forall G1 P G2,
      sequent (G1 ++ P :: G2) P
  | ImpI: forall G P Q,
      sequent (P :: G) Q ->
      sequent G (Imp P Q)
  | ImpE: forall G P Q,
      sequent G (Imp P Q) -> sequent G P ->
      sequent G Q
  | FalseE: forall G P,
      sequent G False ->
      sequent G P.
```

$$\Gamma \vdash \top$$

$$\Gamma_1, P, \Gamma_2 \vdash P$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$$

$$\frac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}$$

Well-founded orders

A strict order is well founded if there are no infinitely decreasing sequences.

A more positive characterization: all `x` are "accessible", that is, all decreasing sequences starting in `x` are finite.

```
Inductive Acc (A: Type) (ord: A -> A -> Prop) : A -> Prop :=
  | Acc_intro: forall x:A,
      (forall y:A, ord y x -> Acc A ord y) -> Acc A ord x.

Definition well_founded (A: Type) (ord: A -> A -> Prop) :=
  forall x:A, Acc A ord x.
```

A structural induction over a proof of `Acc x` corresponds to Noetherian induction over `x`.

Equality as an inductive predicate

In MLTT as in many logics, equality is a primitive notion, with specific deduction rules.

Equality can also be defined as an inductive predicate.

```
Inductive equal (A: Type): A -> A -> Prop :=
  | reflexivity: forall (x: A), equal x x.
```

To show a property `P x y` under hypothesis `H: equal A x y`, we do a case analysis over `H`. The only possible case being `H = reflexivity z`, it suffices to show `P z z` for all `z`.

(More details in the lecture of Jan 23, “What is equality?”.)

IV

Generalized Algebraic Data Types (GADTs)

Generalized Algebraic Data Types (GADTs)

It is the closest thing to inductive families for programming languages such as OCaml and Haskell that lack full dependent types.

GADTs are parameterized algebraic types (`'a ty`) where the constructors do not all produce `'a ty` but can produce instances τ `ty`.

Example: an optimized representation of arrays in OCaml

```
type 'a compact_array =
| Array: 'a array -> 'a compact_array    (* default case *)
| Bytes: bytes    -> char compact_array  (* optimized case *)
| Bools: bitvect  -> bool compact_array  (* optimized case *)
```

History of GADTs

- 1992 Coquand: *Pattern-matching with dependent types*. Combining inductive families with pattern matching.
- 1992 Läufer: *Polymorphic Type Inference and Abstract Data Types*. “Existential types”, a special case of GADT.
- 1994 Augustsson, Petersson: *Silly type families* (draft). Let’s remove the regularity condition over constructor types. Problems to infer the types of `match`.
- 2003 Xi, Chen, Chen: *Guarded Recursive Datatype Constructors*. Rediscovery of the same ideas.
- 2006 Peyton-Jones et al + Pottier and Régis-Gianas. First algorithms for partial type inference for GADTs pattern matching.
- 2007 GHC 6.8.1: introduction of GADTs in Haskell.
- 2012 OCaml 4.00: introduction of GADTs in Caml.

Values that determine types

A special case of dependent types that GADTs express well.

Example: the `sprintf` function (formatted printing)

```
sprintf "toto" : string
sprintf "var = %d" : int -> string
sprintf "%s = %d" : string -> int -> string
```

We obtain a typed version `sprintf : 'a format -> 'a`
by encoding formats using the `'a format` GADT below:

```
type _ format =
  | Lit: string * 'a format -> 'a format
  | Param_int: 'a format -> (int -> 'a) format
  | Param_string: 'a format -> (string -> 'a) format
  | End: string format

sprintf (Lit("toto", End)) : string
sprintf (Lit("var = ", Param_int End)) : int -> string
sprintf (Param_string (Param_int End)) : string -> int -> string
```

Existential types and type equalities

GADTs make it possible to existentially quantify over a type:

```
type printable =
  | Printable: 'a * ('a -> string) -> printable
```

To be read as $\exists A : \texttt{Type}.\ A \times (A \to \texttt{string})$.

We can also define equality between two types:

```
type ('a, 'b) equal =
  | Equal: ('c, 'c) equal
```

Conversely, any GADT can be written as a regular parameterized algebraic type + existential types + type equalities.

V

Functional encodings

Inductive types: primitive or encoded?

Inductive types are a primitive concept in Coq and Agda, just like W types are primitive in Martin-Löf type theory.

However, they can almost be defined in the Pure Type Systems from the previous lecture, using only lambda-terms and Pi-types.

Untyped encoding

A generalization of Church's encoding of natural numbers as pure lambda-terms. We start with the equations defining the recursor for an inductive type:

$$
\begin{array}{lll}
\texttt{bool_rec true}\ \ a\ b & \to & a \\
\texttt{bool_rec false}\ a\ b & \to & b \\
\texttt{nat_rec 0}\ \ a\ b & \to & a \\
\texttt{nat_rec (S}\ n\texttt{)}\ \ a\ b & \to & b\ (\texttt{nat_rec}\ n\ a\ b)
\end{array}
$$

We take recursors to be the identity function, hence constructor = recursor:

$$
\begin{array}{lllllll}
\texttt{true}\ \ a\ b & \to & a & \qquad & \texttt{0}\ \ a\ b & \to & a \\
\texttt{false}\ a\ b & \to & b & \qquad & (\texttt{S}\ n)\ a\ b & \to & b\ (n\ a\ b)
\end{array}
$$

We deduce the lambda-terms that correspond to constructors:

$$
\begin{array}{lll}
\texttt{true} = \lambda a.\ \lambda b.\ a & \qquad & \texttt{0} = \lambda a.\ \lambda b.\ a \\
\texttt{false} = \lambda a.\ \lambda b.\ b & \qquad & \texttt{S} = \lambda n.\ \lambda a.\ \lambda b.\ b\ (n\ a\ b)
\end{array}
$$

Simply-typed encoding

With simple types, we are forced to give monomorphic types not only to constructors but also to pattern matchings! Example with Booleans:

```
type bool = t → t → t       (* for a fixed type t *)
true : bool = λa : t. λb : t. a
false : bool = λa : t. λb : t. b
```

We can write an "if then else" whose result has type t, but no other type. Likewise for natural numbers:

```
type nat = t → (t → t) → t (* for a fixed type t *)
```

we can define the sum and the product of two numbers, but not the exponential function.

Theorem (Schwichtenberg)

The functions $\mathbb{N} \to \mathbb{N}$ definable in simply-typed lambda-calculus are extended polynomials (polynomials + "is zero" test).

System F encoding

System F allows us to quantify universally over the result type of the pattern matching:

$$\texttt{type bool} = \forall X.\ X \to X \to X$$

$$\texttt{type nat} \;\; = \forall X.\ X \to (X \to X) \to X$$

This makes the encoding very expressive, in practice and even in theory.

Theorem (Girard)

All functions $\mathbb{N} \to \mathbb{N}$ provably total in second-order Peano arithmetic are definable in System F.

System F_ω encoding

The encoding is obtained from the types of the constructors, where we replace the inductive type by a type variable `R` (= result type), universally quantified. Type parameters become λ in F_ω.

```
pair: A -> B -> prod A B
                prod = λA : ∗. λB : ∗. ∀R : ∗. (A -> B -> R) -> R

inl: A -> sum A B
inr: B -> sum A B
                sum = λA : ∗. λB : ∗. ∀R : ∗. (A -> R) -> (B -> R) -> R

nil: list A
cons: A -> list A -> list A
                list = λA : ∗. ∀R : ∗. R -> (A -> R -> R) -> R
tt: unit
                unit  = ∀R : ∗. R -> R

                empty = ∀R : ∗. R
```

System F_ω encoding

The approach extends to GADTs and to inductive families by replacing the type constructor by a type variable of the same kind, universally quantified.

```
Array: ∀X. array X -> compact_array X
Bytes: bytes -> compact_array char
Bools: bitvect -> compact_array bool
       compact_array = λA : ∗. ∀R : ∗ ⇒ ∗.
                         (∀X. array X -> R X) ->
                         (bytes -> R char) ->
                         (bitvect -> R bool) -> R A

refl: ∀X. equal X X
       equal = λA : ∗. λB : ∗. ∀R : ∗ ⇒ ∗ ⇒ ∗. (∀X. R X X) -> R A B.
```

Limitation: no dependent elimination

In general, the type of a case analysis depends on the value being analyzed. This is often the case when the type is a proposition and the case analysis its proof.

Example: to show $\forall b : \texttt{bool}.\, P(b)$, we do a case analysis
`match b with false => ... | true => ... end`
whose type is `P b`.

The eliminator does not have type $\texttt{bool} \to R \to R \to R$
but it has type $\texttt{bool} \to R\, \texttt{false} \to R\, \texttt{true} \to \forall b : \texttt{bool}, R\, b$.

A functional encoding would lead to a circular type…

```
bool = ∀R: bool -> Type.  R -> R -> ∀b: bool. R b
```

Limitation: no “large elimination”

A pattern matching can produce a result in a universe “above” that of the matched value. For example: we define a type by pattern matching on a `nat`.

```
Fixpoint vec (A: Type) (n: nat) : Type :=
    match n with
    | O => unit
    | S p => prod A (vec A p)
    end.
```

In a functional encoding, this leads to universe inconsistency.

VI

Advanced topics

The limitations of the guard condition

```
Fixpoint f (x: T) :=  ...  f a ...  f b ...
```

The guard condition “a, b strict subterms of x” is

1. not always well defined (many variants in Coq);
2. incompatible with abstraction.

In particular, it does not deal with nested recursion nor with generative recursion (through other functions):

```
Fixpoint f (x: T) :=  ...  f (g x) ...
```

```
Fixpoint f (x: T) :=  ...  f ( ... f x ... ) ...
```

Example of non-structural recursion

Euclidean division over Peano integers:

✔
```
Fixpoint minus (p q: nat) :=
  match p, q with
  | O, _ => O
  | _, O => q
  | S p', S q' => minus p' q'
  end.
```

✘
```
Fixpoint div (p q: nat) :=      (* divide p by q+1 *)
  match p with
  | O => O
  | S p' => S (div (minus p' q) q)
  end.
```

`div` terminates because `minus p' q <= p'`. A Noetherian recursion (over the well-founded order of natural numbers) est nécessaire.

Sized types

For each inductive type T we distinguish:

- T^i : the $t : T$ of size $< i$. (i natural number or ordinal)
- T^{i+1}: the $t : T$ of size $\leq i$.

We can, then, typecheck recursion as follows:

$$\frac{\Gamma, f : T^i \to S \vdash e : T^{i+1} \to S}{\Gamma \vdash \texttt{fixpoint}\, f = e : T \to S}$$

Combined with subtyping $T^i <: T^j$ if $i < j$.

In the Euclidean division example, we can show
$\texttt{minus} : \texttt{nat}^i \to \texttt{nat} \to \texttt{nat}^i$, and conclude that `div` is a correct recursion.

Sized types

Agda uses dependent types to add size annotations to inductive types:

```
sized type SNat : Size -> Set where
  zero: ∀(i: size) -> SNat ($ i)
  succ: ∀(i: size) -> SNat i -> SNat ($ i)
```

`SNat i` is the type of numbers of size $\leq$ `i`.

`Size` is a predefined type $\approx$ ordinals..

$ is “size + 1”.

Other kinds of inductive types

Mutual inductives:

```
Inductive tree : Type :=
| Leaf: A -> tree
| Node: forest -> tree

and forest : Type :=
| Nil: forest
| Cons: tree -> forest -> forest.
```

Nested inductives:

```
Inductive tree : Type :=
| Leaf: A -> tree
| Node: list tree -> tree.
```

OK in Coq and Agda. Induction principles sometimes too weak.

Other kinds of inductive types

Induction-recursion: (P. Dybjer)

- An inductive type `A: Type`
- that uses a function `A -> B`.

Induction-induction:

- An inductive type `A: Type`
- that uses an inductive family `B: A -> Type`.

A motivation: encode a type theory (e.g. MLTT) in another.
(J. Chapman, *Type theory should eat itself*, 2009.)

Quotient types

A construction very often used in mathematics:
the quotient of a set by an equivalence relation.

Example: $\mathbb{Q} = (\mathbb{Z} \times \mathbb{Z}^*) \;/\; R$ where $R\,(p, q)\,(p', q') \stackrel{def}{=} pq' = p'q$.

In type theory, no general notion of “quotient type”.

In special cases we can define the quotient type as the subset type of canonical representatives:

```
Definition Q_canon (pq: Z * Z) : Prop :=
  let (p, q) := pq in q > 0 /\ gcd p q = 1.

Definition Q = { pq: Z * Z | Q_canon pq }
```

Higher Inductive Types (HITs)

A concept from homotopy type theory: define an inductive type by its constructors and the equalities they satisfy.

```
Inductive Z2Z : Type :=
| O: Z2Z
| S: Z2Z -> Z2Z
| mod2: S (S O) = O.
```

Enforce that pattern matchings are compatible with these equalities.

```
match (n: Z2Z) with
| O => a
| S p => b p
| mod2 => (* proof that a = b (S O) *)
end.
```

(To be continued in the lecture of Jan 23, “What is equality?”.)

VII

Further reading

Further reading

- Yves Bertot et Pierre Castéran. *Interactive Theorem Proving and Program Development*. Chapitres 6, 8, 14, 15.
- Adam Chlipala. *Certified Programming with Dependent Types*. `http://adam.chlipala.net/cpdt/`. Chapitres 3, 4, 6, 8, 9.