Programming with Dependent Types in Coq: Dependent pattern-matching and recursion

Matthieu Sozeau, Inria

CASS 2020 Summer School
January 7th 2020
Chile

Paradigm: Programming = Proving

COQ's Type Theory includes both:

- a programming language
- a logic

Programming language

A total, functional programming language:

- Algebraic datatypes (inductive and coinductive types) with pattern-matching: tree-like structures, record types, optionally primitive types (e.g. native integers).
- Higher-order, structurally recursive functions
- Higher-order polymorphism (e.g. not restricted to ML-style polymorphism)

Examples:

```
Inductive nat : Type :=
| 0 : nat
| S : nat -> nat.

Fixpoint iter {A : Type} (f : A -> A) (n : nat) (a : A) : A :=
  match n with
  | 0 => a
  | S n' => iter f n' (f a)
  end.
```

Logic

- A dedicated impredicative sort Prop of propositions, with connectives $\wedge$, $\vee$, $\rightarrow$, $\bot$, $\top$, universal and existential quantification.
- Can talk about "individuals" such as inductive definitions (natural numbers, lists, ...), propositions, predicates and types.

Examples:

```
Lemma map_app : forall {A B} (l l' : list A) (f : A -> B),
  map f (l ++ l') = map f l ++ map f l'.

Inductive In {A} : x -> list A -> Prop :=
| here {x xs} : In x (x :: xs)
| there {y x xs} : In x xs -> In x (y :: xs).

Definition absurd : Prop := forall P : Prop, P.
```

Paradigm: Programming = Proving

To ensure the logic is useful, we want consistency:

$$\text{There is no } p \text{ such that } \vdash p : \texttt{absurd}$$

Equivalently, given:

```
  Inductive void : Type := .
```

No program p of type `void` can be constructed: there is no (closed) value of type `void`.

Canonicity

Constructive type theories like Coq's go further and asks for a canonicity property. From a derivation of:

$$\vdash p : \exists x : nat, P\ x$$

we can always extract an actual numeral $\overline{n}$ and a proof of $P\ \overline{n}$.

This is a cousin of the usual progress and preservation properties of programming languages:

If $\vdash p : \tau$ then p is *convertible* to a value of type τ.

Informally: values $=$ introduction forms (e.g. constructors, lambda-abstractions)

Constructivity II

In COQ's case, the theory also enjoys Strong Normalization:

If $\vdash t \equiv u$ then there exists a unique normal form n such that $t \to n$ and $u \to n$

I.e., convertibility is decidable by a normalization algorithm.

Values are the closed normal forms, so if $\vdash n : \mathbb{N}$, we are additionally guaranteed to find the unique numeral value convertible to n by reduction.

Paradigm: Programming = Proving

To ensure totality, Type Theory *removes* features found in usual programming languages:

- Partiality: where a program can have an undefined value and crash if called outside its domain of definition.
- Non-termination: where a program might not ever give an answer or produce anything useful.
- Or any other "effect".

"Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs"
John Backus, Turing Award Lecture, 1977

"Total Functional Programming"
D.A. Turner, Miranda author, an ancestor of Haskell

Partiality Example

The “nth” function on lists, in an ML-style programming language:

```
val nth : 'a list -> int -> 'a

let rec nth l n =
  match l, n with
  | nil, _ -> assert false
  | cons x _, 0 -> x
  | cons _ xs, n -> nth xs (pred n)

val bug : 'a
let bug = nth [] 0
```

How to model partiality?

We can model or encode the partiality of `nth`:

- Adding a default value argument of type `'a`
- Changing the return type to use the option type (a.k.a. the error monad): `nth` will return `None` if the index is out of bounds

These two solutions change the program, its type and its meaning!

Non-termination

In pure λ-calculus we have the fixed-point combinator:

$$Y := \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x, f\ (x\ x))$$

such that:

$$YF \equiv F(YF)$$

- Allows to encode any recursive function!
- Essential for the Church-Turing thesis: every computable function $f : \mathbb{N} \to \mathbb{N}$ can be represented by a λ-term or a Turing machine.

ML program:

```
let rec fix f x = f (fix f) x
let loop : 'a = fix (fun f x -> f x) 0
```

Clearly we cannot accept this as this would allow to prove `absurd`.

Recursion and Termination

In total type theories, we stick to *structural* recursion on inductive types:

- Via their eliminator in Lean (the algebra / induction principle associated to an inductive type)
- Via a syntactic *guard condition* in Coq.
- Via *sized-types* annotations in Agda (similar to the guard condition but more modular)

All these criterions are quite constraining in practice:

- Termination is not decidable, so all these methods are incomplete.
- Termination can depend on the correction of the program.
  $\Rightarrow$ One must mix program and proof in these cases.

Recursion and Termination

In total type theories, we stick to *structural* recursion on inductive types:

- Via their eliminator in Lean (the algebra / induction principle associated to an inductive type)
- Via a syntactic *guard condition* in Coq.
- Via *sized-types* annotations in Agda (similar to the guard condition but more modular)

All these criterions are quite constraining in practice:

- Termination is not decidable, so all these methods are incomplete.
- Termination can depend on the correction of the program.
  $\Rightarrow$ One must mix program and proof in these cases.

Question: Besides provability of equivalent or stronger type theories (Gödel incompleteness), can you describe one meaningful function $f : \mathbb{N} \to \mathbb{N}$ that cannot be defined in type theory?

A motto

The result of these restrictions:

- 😞 expressivity and faithfullness to the computational behavior of programs the user wants to write
- 😞 ease of use: resorting to encodings has a cost
- 😁 simple equational theory (algebra) for programs
- 😁 programs and logic can be freely mixed (”true” dependency)

Dependent types allow to still express programs of interest without changing their computational behavior

A motto

The result of these restrictions:

- 😞 expressivity and faithfullness to the computational behavior of programs the user wants to write
- 😞 ease of use: resorting to encodings has a cost
- 😁 simple equational theory (algebra) for programs
- 😁 programs and logic can be freely mixed (”true” dependency)

Dependent types allow to still express programs of interest without changing their computational behavior

How, you ask?

1. Give them stronger specifications/types

A motto

The result of these restrictions:

- 😞 expressivity and faithfullness to the computational behavior of programs the user wants to write
- 😞 ease of use: resorting to encodings has a cost
- 😁 simple equational theory (algebra) for programs
- 😁 programs and logic can be freely mixed ("true" dependency)

Dependent types allow to still express programs of interest without changing their computational behavior

How, you ask?

1. Give them stronger specifications/types
2. Provide evidence (proofs) that the specifications can be met

A motto

The result of these restrictions:

- 😞 expressivity and faithfullness to the computational behavior of programs the user wants to write
- 😞 ease of use: resorting to encodings has a cost
- 😁 simple equational theory (algebra) for programs
- 😁 programs and logic can be freely mixed (”true” dependency)

Dependent types allow to still express programs of interest without changing their computational behavior

How, you ask?

1. Give them stronger specifications/types
2. Provide evidence (proofs) that the specifications can be met
3. Develop tools to make this mixed program-proof development easier.

Outline of the lectures

1. Coq's Type theoretic constructs for specification
   - Type Theory 101: dependent functions, inductives and equality
   - The `Prop`/`Type` distinction and erasure
2. Dealing with partiality and termination using proofs and types
   - Subset types
   - Reflecting program behavior in types
   - Well-founded recursion
3. Indexed families and dependent pattern-matching
   - Intrinsic vs extrinsic specifications
   - The meaning of dependent pattern-matching
   - Example: a definitional interpreter for simply-typed λ-calculus with references.

Throughout, I will introduce features of the Program and Equations tools.

Outline

Coq's Syntax

$$
\begin{array}{lll}
t, u, T, U, P := x & & \text{(variable)} \\
\quad \mid \ \texttt{fun}\ x : T \Rightarrow t \mid \lambda x : T.t & & \text{(abstraction)} \\
\quad \mid \ t\ u & & \text{(application)} \\
\quad \mid \ \texttt{forall}\ x : T, U \mid \Pi x : T.U & & \text{(dependent product)}
\end{array}
$$

Coq's Syntax

$t, u, T, U, P := x$	(variable)
$\mid\quad \texttt{fun}\ x : T \Rightarrow t \mid \lambda x : T.t$	(abstraction)
$\mid\quad t\ u$	(application)
$\mid\quad \texttt{forall}\ x : T, U \mid \Pi x : T.U$	(dependent product)
$\mid\quad s \in \{\texttt{Prop}, \texttt{Set}, \texttt{Type}_i\ (i \in \mathbb{N})\}$	(sorts)

Coq's Syntax

$t, u, T, U, P := x$	(variable)
$\mid$ $\texttt{fun}\ x : T \Rightarrow t \mid \lambda x : T.t$	(abstraction)
$\mid$ $t\ u$	(application)
$\mid$ $\texttt{forall}\ x : T, U \mid \Pi x : T.U$	(dependent product)
$\mid$ $s \in \{\texttt{Prop}, \texttt{Set}, \texttt{Type}_i\ (i \in \mathbb{N})\}$	(sorts)
$\mid$ $\texttt{let}\ x\ :\ T\ := t\ \texttt{in}\ t'$	(local definition)

Coq's Syntax

$$
\begin{array}{rll}
t, u, T, U, P := & x & \text{(variable)} \\
\mid & \texttt{fun}\ x : T \Rightarrow t \mid \lambda x : T.t & \text{(abstraction)} \\
\mid & t\ u & \text{(application)} \\
\mid & \texttt{forall}\ x : T, U \mid \Pi x : T.U & \text{(dependent product)} \\
\mid & s \in \{\texttt{Prop}, \texttt{Set}, \texttt{Type}_i\ (i \in \mathbb{N})\} & \text{(sorts)} \\
\mid & \texttt{let}\ x\ :\ T\ := t\ \texttt{in}\ t' & \text{(local definition)} \\
\mid & \mathrm{constant} \mid \textsf{inductive} \mid \textsf{constructor} & \text{(global reference)}
\end{array}
$$

Coq's Syntax

$t, u, T, U, P := x$	(variable)
$\mid$ $\texttt{fun}\ x : T \Rightarrow t \mid \lambda x : T.t$	(abstraction)
$\mid$ $t\ u$	(application)
$\mid$ $\texttt{forall}\ x : T, U \mid \Pi x : T.U$	(dependent product)
$\mid$ $s \in \{\texttt{Prop}, \texttt{Set}, \texttt{Type}_i\ (i \in \mathbb{N})\}$	(sorts)
$\mid$ $\texttt{let}\ x\ :\ T\ := t\ \texttt{in}\ t'$	(local definition)
$\mid$ constant $\mid$ inductive $\mid$ constructor	(global reference)
$\mid$ $\texttt{match}\ t\ \texttt{in}\ \mathsf{ind}\ \overrightarrow{xs}\ \texttt{as}\ x\ \texttt{return}\ P\ \texttt{with}$ $\overrightarrow{\mathsf{C}\ \overrightarrow{ps_i} \Rightarrow u_i}\ \texttt{end}$	(pattern-matching)

Coq's Syntax

$t, u, T, U, P := x$	(variable)
$\mid\ \texttt{fun}\ x : T \Rightarrow t \mid \lambda x : T.t$	(abstraction)
$\mid\ t\ u$	(application)
$\mid\ \texttt{forall}\ x : T, U \mid \Pi x : T.U$	(dependent product)
$\mid\ s \in \{\texttt{Prop}, \texttt{Set}, \texttt{Type}_i\ (i \in \mathbb{N})\}$	(sorts)
$\mid\ \texttt{let}\ x\ :\ T\ := t\ \texttt{in}\ t'$	(local definition)
$\mid$ constant $\mid$ inductive $\mid$ constructor	(global reference)
$\mid\ \texttt{match}\ t\ \texttt{in}\ \mathsf{ind}\ \overrightarrow{xs}\ \texttt{as}\ x\ \texttt{return}\ P\ \texttt{with}$ $\overrightarrow{\mathsf{C}\ \overrightarrow{ps_i} \Rightarrow u_i}\ \texttt{end}$	(pattern-matching)
$\mid\ \texttt{fix}\ f\ \overrightarrow{xs}\ \{\texttt{struct}x_i\} : T := t$ $\overrightarrow{\texttt{with}\ f_i\ \overrightarrow{xs_i} : T_i := t_i}\ \texttt{in}\ f_j$	(fixpoints)
$\mid\ \texttt{cofix}\ f : T := t\ \overrightarrow{\texttt{with}\ f_i\ \overrightarrow{xs_i} : T_i := t_i}\ \texttt{in}\ f_j$	(co-fixpoints)

Coq's Syntax

$$
\begin{array}{lll}
t, u, T, U, P := x & & \text{(variable)} \\
\mid & \texttt{fun}\ x : T \Rightarrow t \mid \lambda x : T.t & \text{(abstraction)} \\
\mid & t\ u & \text{(application)} \\
\mid & \texttt{forall}\ x : T, U \mid \Pi x : T.U & \text{(dependent product)} \\
\mid & s \in \{\texttt{Prop}, \texttt{Set}, \texttt{Type}_i\ (i \in \mathbb{N})\} & \text{(sorts)} \\
\mid & \texttt{let}\ x\ :\ T\ := t\ \texttt{in}\ t' & \text{(local definition)} \\
\mid & \text{constant} \mid \text{inductive} \mid \text{constructor} & \text{(global reference)} \\
\mid & \texttt{match}\ t\ \texttt{in}\ \text{ind}\ \overrightarrow{xs}\ \texttt{as}\ x\ \texttt{return}\ P\ \texttt{with} & \text{(pattern-matching)} \\
 & \overrightarrow{\text{C}\ \overrightarrow{ps_i} \Rightarrow u_i}\ \texttt{end} & \\
\mid & \texttt{fix}\ f\ \overrightarrow{xs}\ \{\texttt{struct}\, x_i\} : T := t & \text{(fixpoints)} \\
 & \overrightarrow{\texttt{with}\ f_i\ \overrightarrow{xs_i} : T_i := t_i}\ \texttt{in}\ f_j & \\
\mid & \texttt{cofix}\ f : T := t\ \overrightarrow{\texttt{with}\ f_i\ \overrightarrow{xs_i} : T_i := t_i}\ \texttt{in}\ f_j & \text{(co-fixpoints)} \\
 & & \\
\Gamma := [\,] \mid \Gamma, x : T \mid \Gamma, x : T := u & & \text{contexts}
\end{array}
$$

Coq's Type Theory

Judgments:

- $\Sigma; \Gamma \vdash$ means local context Γ and global context Σ are well-formed (all items are well-typed)
- $\Sigma; \Gamma \vdash M : A$ means term M has type A in local context Γ and global context Σ
- $\Sigma; \Gamma \vdash A \equiv B$ means terms/types A and B are convertible in context Γ, and global context Σ
- $\Sigma; \Gamma \vdash A \leq B$ means type A is in the cumulativity relation with B in context Γ, and global context Σ

Organized as:

- formation rules (rule for Π)
- introduction rules (rule for λ)
- elimination rules (rule for application)
- computation rules (β-reduction)

Conversion

Convertibility $\equiv$ is the congruence generated by reduction rules: β, ζ, δ, ι, fix, cofix, respecting α-equivalence.

The global context Σ contains definitions of constants (with or without a body) and inductive type declarations. It stays constant in typing rules so we usually omit it.

β	$=$	beta-reduction
δ	$=$	unfolding of constants
ζ	$=$	reduction of let-ins
ι	$=$	reduction of `match`
fix	$=$	fixpoint reduction
cofix	$=$	cofixpoint reduction

Variables

$$\frac{}{[\,]\vdash} \qquad \frac{\Gamma\vdash \quad \Gamma\vdash A : s\ (s \text{ a sort})}{\Gamma, x : A\vdash} \qquad \frac{\Gamma\vdash \quad x : A\in\Gamma}{\Gamma\vdash x : A}$$

Sorts and Cumulativity

$\texttt{Set} = \texttt{Type}_0$ (unless in `-impredicative-set` mode)

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{Prop} : \texttt{Type}_1}\ \textsc{Prop-Intro}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{Type}_n : \texttt{Type}_{n+1}}\ \textsc{Type-Intro}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{Prop} \leq \texttt{Type}_0}\ \textsc{Prop-Cumulativity}$$

$$\frac{\Gamma \vdash \qquad n \leq m}{\Gamma \vdash \texttt{Type}_n \leq \texttt{Type}_m}\ \textsc{Type-Cumulativity}$$

- Cumulativity includes convertibility.

Cumulativity

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s \quad \Gamma \vdash A \leq B}{\Gamma \vdash t : B}\ \textsc{Cumul}$$

Example:

```
Variable P : nat -> Type.
Variable p : P 42.
Definition conv : P (6 * 7) := p.
```

Dependent products and impredicativity

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B \ : \mathtt{max}(s_1, s_2)}\ \Pi\text{-}\textsc{Form}$$

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma, x : A \vdash B \leq B'}{\Gamma \vdash \Pi x : A.B \leq \Pi x : A'.B'}\ \Pi\text{-}\textsc{Cumulativity}$$

$\text{max}(dom, \mathtt{Prop})$	$=$	$\mathtt{Prop}$	impredicativity
$\text{max}(\mathtt{Type}_i, \mathtt{Type}_j)$	$=$	$\mathtt{Type}_{max(i,j)}$	predicativity
$\text{max}(\mathtt{Prop}, \mathtt{Type}_i)$	$=$	$\mathtt{Type}_i$	predicativity

Examples: (`A -> B` $\equiv$ `forall _ : A, B`)

```
Definition tautology : Prop := forall P : Prop, P -> P.
Definition predicative@{i j | i < j} : Type@{j} :=
  forall P : Type{i}, P.
Fail Definition predicativeset : Set := forall A : Set, A -> A.
```

Dependent products

The dependent product handles all kinds of quantifications:

```
Definition over_objects := forall n : nat, 0 <= n.

Definition over_fns := forall _ : (nat -> nat), nat -> nat.

Definition over_types := forall A : Set, list A -> nat.

Definition over_propositions := forall A : Prop, A -> A.

Definition over_typeconstr := forall M : Type -> Type, forall
     A : Type, A -> M A.

...
```

Dependent products

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B \ : \ \mathtt{max}(s_1, s_2)}\ \Pi\text{-}\textsc{Form}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M \ : \ \Pi x : A.B}\ \Pi\text{-}\textsc{Intro}$$

$$\frac{\Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N \ : \ B[N/x]}\ \Pi\text{-}\textsc{Elim}$$

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma, x : A \vdash B \leq B'}{\Gamma \vdash \Pi x : A.B \leq \Pi x : A'.B'}\ \Pi\text{-}\textsc{Cumulativity}$$

Π-COMPUTATION: $(\lambda x : A.t)b \rightarrow^{\beta} t[b/x]$

Inductive Sets

Induction is a very general principle that has many instances in mathematics.

Examples of inductive sets:

- Natural numbers ($\Rightarrow$ mathematical induction)
- Sets/Subsets defined by inference rules
- Generalization to well-founded trees (structural induction)

Natural numbers, inductively

- 0 is a natural number;
- if n is a natural number, then $S(n)$ is a natural number;
- equational theory: discrimination, injectivity;
- induction scheme:
  $P(0)$ and $\forall n.\, P(n) \Rightarrow P(S(n))$ implies $\forall n. P(n)$

Type of Natural Numbers

Martin-Löf style-scheme:

- 1 formation rule:

$$\frac{}{\vdash \mathbb{N} : \mathtt{Type}_0}$$

- 2 introduction rules:

$$\frac{}{\vdash \mathsf{0} : \mathbb{N}} \qquad \frac{\vdash n : \mathbb{N}}{\vdash \mathsf{S}\ n : \mathbb{N}}$$

- 1 elimination rule ($P : \mathbb{N} \to \mathtt{Type}$ as a subset/predicate on $\mathbb{N}$)

$$\frac{\begin{array}{c}\vdash P : \mathbb{N} \to \mathtt{Type} \qquad \vdash n : \mathbb{N} \\ \vdash f_0 : P\ \mathsf{0} \qquad \vdash f_S : \Pi\ n{:}\mathbb{N}.\, P\ n \to P\ (\mathsf{S}\ n)\end{array}}{\vdash \mathrm{nat\_rect}\ P\ f_0\ f_S\ n : P\ n}$$

- 2 computation rules:
  $\mathrm{nat\_rect}\ P\ f_0\ f_S\ \mathsf{0} = f_0$
  $\mathrm{nat\_rect}\ P\ f_0\ f_S\ (\mathsf{S}\ n) = f_S\ n\ (\mathrm{nat\_rect}\ P\ f_0\ f_S\ n)$

Inductive types in Coq

Coq provides the user with a general mechanism:

- Formation: the user declares the name and arity of the inductive type, e.g. nat : `Set`,
  In : `forall` A : `Set`, list $A \to A \to$ `Prop`
- Introduction: constructors of the inductive type
- Elimination: a dependent induction/recursion scheme is derived systematically
- Computation: derived systematically (ι-reduction)

Specificities of COQ's inductive types:

- Coq checks the definition preserves consistency
  $\Rightarrow$ Strictly positive inductive definitions
- Coq allows impredicative inductive definitions in `Prop`
- Elimination/computation is split between `fix` and `match` primitive constructs.

Natural numbers in Coq

Declaration of the natural numbers:

```
Inductive nat : Set :=
| O : nat | S : nat -> nat.
```

(or `Inductive nat := O | S (n:nat).`)

which defines

- a type $\Gamma \vdash \mathbb{N} : \mathtt{Set}$
- a set of introduction rules for this type : constructors

$$\Gamma \vdash \mathsf{O} : \mathbb{N} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{S}\ n : \mathbb{N}}$$

Recursive inductive types: Natural numbers example

which defines also

- an elimination rule (pattern-matching operator with a result depending on the object which is eliminated)

$$\frac{\Gamma \vdash t : \mathbb{N} \quad \Gamma, x{:}\mathbb{N} \vdash P\ x : s \qquad \Gamma \vdash t_1 : P\ \mathsf{O} \quad \Gamma, n{:}\mathbb{N} \vdash t_2 : P\ (\mathsf{S}\ n)}{\Gamma \vdash \texttt{match}\ t\ \texttt{as}\ x\ \texttt{return}\ P\ x\ \texttt{with}\ \mathsf{O} \Rightarrow t_1 \mid \mathsf{S}\ n \Rightarrow t_2\ \texttt{end} : P\ t}$$

- reduction rules preserve typing (ι-reduction)

$(\texttt{match}\ \mathsf{O}\ \texttt{as}\ x\ \texttt{return}\ P\ x\ \texttt{with}\ \mathsf{O} \Rightarrow t_1 \mid \mathsf{S}\ n \Rightarrow t_2\ \texttt{end}) \rightarrow_\iota t_1$

$(\texttt{match}\ \mathsf{S}\ m\ \texttt{as}\ x\ \texttt{return}\ P\ x\ \texttt{with}\ \mathsf{O} \Rightarrow t_1 \mid \mathsf{S}\ n \Rightarrow t_2\ \texttt{end})$
$\rightarrow_\iota t_2[m/n]$

Recursive inductive types

Example of natural numbers

- We obtain case analysis and construction by cases :

```
λP : ℕ → s.
λH_O : P O.
λH_S : ∀m : ℕ. P (S m).
λn : ℕ.
 match n as y return P y with
  | O ⇒ H_O
  | S m ⇒ H_S m
 end
```

- is a proof of

$$\forall P : \mathbb{N} \to s.\, P\ \mathsf{O} \to (\forall m : \mathbb{N}.\, P\ (\mathsf{S}\ m)) \to \forall n : \mathbb{N}.\, P\ n$$

How to derive the standard recursion scheme ?

Fixpoint operator : from case analysis to recursor

case-analysis

```
$\lambda P : \mathbb{N} \to s,$
$\lambda H_O : P\ 0,$
$\lambda H_S : \forall m : \mathbb{N}, P\ (\mathsf{S}\ m),$
$\lambda n : \mathbb{N},$
 match $n$ return $P(n)$ with
    $\mathsf{O} \Rightarrow H_O \mid \mathsf{S}\ m \Rightarrow H_S\ m$
 end
```

has type

$\forall P : \mathbb{N} \to s,$
$P\ \mathsf{O} \to$
$(\forall m : \mathbb{N}, P(\mathsf{S}\ m)) \to$
$\forall n : \mathbb{N}, P\ n$

recursor

```
$\lambda P : \mathbb{N} \to s,$
$\lambda H_O : P\ \mathsf{O},$
$\lambda H_S : \forall m : \mathbb{N}, P\ m \to P\ (\mathsf{S}\ m),$
fix $f$ $(n : \mathbb{N})$ : $P(n)$ :=
  match $n$ return $P\ n$ with
    $\mathsf{O} \Rightarrow H_O \mid \mathsf{S}\ m \Rightarrow H_S\ m\ (f\ m)$
  end
```

has type

$\forall P : \mathbb{N} \to s,$
$P\ \mathsf{O} \to$
$(\forall m : \mathbb{N}, P\ m \to P\ (\mathsf{S}\ m)) \to$
$\forall n : \mathbb{N}, P\ n$

Fixpoint operator : well-foundness

Requirement of the Calculus of Inductive Constructions :

- the argument of the fixpoint has type an inductive definition
- recursive calls are on arguments which are *structurally* smaller

Example of recursor on natural numbers

```
$\lambda P : \mathbb{N} \to s,$
$\lambda H_O : P\ \mathsf{O},$
$\lambda H_S : \forall m : \mathbb{N}, P\ m \to P\ (\mathsf{S}\ m),$
fix $f\ (n : \mathbb{N})\ :\ P(n)\ :=$
  match $n$ as $y$ return $P(y)$ with
     $\mathsf{O} \Rightarrow H_O \mid \mathsf{S}\ m \Rightarrow H_S\ m\ (f\ m)$
  end
```

is correct with respect to CCI : recursive call on m which is structurally smaller than n in the inductive $\mathbb{N}$.

More complex inductive types

inductives.v

Equality tactics reference

Tactics:

- `f_equal` (congruence) $\frac{x=y}{f(x)=f(y)}$
- `discriminate` (constructor discrimination) $\frac{C(t_1,\ldots,t_n)=D(u_1,\ldots,u_k)}{A}$
- `injection` (injectivity of constructors) $\frac{C(t_1,..,t_n)=C(u_1,\ldots,u_n)}{t_1=u_1 \quad \ldots \quad t_n=u_n}$
- `inversion` (necessary conditions) $\frac{\text{even } (S(Sn))}{\text{even } n}$
- `rewrite` (substitution) $\frac{x=y \quad P(y)}{P(x)}$
- `symmetry`, `transitivity`

A proof / program distinction

`Prop`/`Type` and erasure

Proofs and programs

The `Prop`/`Type` distinction of Coq (Paulin-Mohring, 1989; Letouzey, 2004):

- Slightly refining the Curry-Howard correspondence, we distinguish logical and computational parts of our terms.
- This is done by restricting logical reasoning to the sort `Prop` ("proofs") and representation of computational objects (e.g, numbers, structures) to `Set`/`Type`.
- The calculus prevents dependencies of a computational term on a proof.

$$p : x < y \ \vee \ x > y \nvdash \begin{array}{l}\texttt{match}\ p\ \texttt{in}\ \mathsf{or}\ \_\ \_\ \texttt{return}\ \mathsf{bool}\ \texttt{with} \\ \mid \mathsf{in\_left}\ H \Rightarrow \mathsf{true} \\ \mid \mathsf{in\_right}\ H \Rightarrow \mathsf{false} \\ \texttt{end} : \mathsf{bool}\end{array}$$

- However, it is fine for a proof to depend on another proof:

$$p : P \vee Q \vdash \texttt{match}\ p\ \texttt{with} \ldots : Q \vee P$$

Bridges between Prop and Type

- The principle associated to Prop is named *proof-irrelevance*:

$$\forall (P : \texttt{Prop})(p\ q : P), p = q$$

.

Informally: no construction should depend on the specific shape of a proof. We can always replace one proof by another and obtain equal things.

However, the restriction of dependencies from computational terms on propositional ones is not total!

Bridges between Prop and Type

- The principle associated to Prop is named *proof-irrelevance*:

$$\forall (P : \texttt{Prop})(p\ q : P), p = q$$

.

Informally: no construction should depend on the specific shape of a proof. We can always replace one proof by another and obtain equal things.

However, the restriction of dependencies from computational terms on propositional ones is not total!

We can eliminate a proposition P : Prop to produce a computational object $t : T$: Type iff:

1. P has at most one constructor (e.g. True, False)
2. All its arguments are propositions (e.g. prod).

Singleton elimination

We can eliminate a proposition P : `Prop` to produce a computational object $t : T$: `Type` iff:

1. P has at most one constructor (e.g. True, False)
2. All its arguments are propositions (e.g. prod).

We call such propositions singleton types.

Intuition: these propositions naturally validate proof irrelevance: they have at most one proof.

So, eliminating them cannot bring any information: this is like eliminating a dummy unit value.

The case of disjunction

We want to prevent information from proofs to flow to computational bool values:

$$p : x < y \ \vee \ x > y \nvdash \quad \texttt{match}\ p\ \texttt{in}\ \mathsf{or}\ \_\ \_\ \texttt{return}\ \mathsf{bool}\ \texttt{with}$$

```
| in_left H ⇒ true
| in_right H ⇒ false
end : bool
```

Disjunction does not satisfy the singleton inductive criterion as it has two constructors. We hence restrict its elimination sort to Prop. Here bool : Set $\neq$ Prop.

Technically, the match typing rule for inductives in Prop which are not singletons forces the return sort to be Prop (check or_ind and elimination_restrictions.v).

The case of equality

Equality, having a single constructor with no arguments, is also a singleton type:

`Inductive` eq $\{A\}$ $(a : A) : A \to$ `Prop` := eq_refl : eq a a.

$$\frac{\Gamma \vdash e : \mathsf{eq}\ A\ t\ u \qquad \Gamma, y{:}A, e'{:}\mathsf{eq}\,A\,t\,y \vdash C\ y\ e' : s \qquad \Gamma \vdash t : C\ t\ (\mathsf{eq\_refl}_{A,t})}{\Gamma \vdash \left(\begin{array}{l}\texttt{match}\ e\ \texttt{in}\ \mathsf{eq}\ \_\ y\ \texttt{as}\ e'\ \texttt{return}\ C\ y\ e'\ \texttt{with} \\ \quad \mid \mathsf{eq\_refl} \Rightarrow t \\ \texttt{end}\end{array}\right) : C\ u\ e}$$

- This elimination rule allows any sort s here.
- Equality reasoning is hence only logical information with no computational content in this interpretation.
- You will see in N. Tabareau's lecture that this is not the only possible interpretation.

Erasure

The point of this distinction is to allow program erasure (a part of extraction in $\textsc{Coq}$)

Erasure $\mathcal{E}(t)$ produces from a computational term $t : T :$ `Type` a pure λ-calculus term with only the computational content of t:

- We remove all types and proofs.
- This is a compiler for $\textsc{Coq}$!

Erasure

Definition: By computational content of a Coq term we know refer to the associated erased λ-term.

Informally:

- If $t : P :$ `Prop` (t is a proof), then $\mathcal{E}(t) = \square$
- If $t :$ `Type` (t is a type), then $\mathcal{E}(t) = \square$
- Otherwise t is informative (e.g. a natural number) and we recursively extract its subterms.

We want to ensure preservation of observations by erasure:

$$\text{If } \vdash t : \mathbb{N} \text{ et } t \to^* \underline{n} \text{ then } \mathcal{E}(t) \to^* \underline{n}.$$

Erasure examples

Examples:

$$
\begin{array}{lcl}
\mathcal{E}(\lambda(X:\texttt{Type})(x:X).x) & = & \lambda x.x \\
\mathcal{E}(\lambda(m\ n:\mathbb{N})(H:n>0).\texttt{div}\ m\ n\ H) & = & \lambda m\ n.\mathcal{E}(\texttt{div})\ m\ n \\
\mathcal{E}(\texttt{match}\ e\ \texttt{with}\ \mathsf{eq\_refl} \Rightarrow t\ \texttt{end}) & = & \mathcal{E}(t)
\end{array}
$$

The singleton elimination criterion ensure that what was mixed with computational parts can be removed without changing the result of the program (assuming do not want to observe proofs!).

Erasure of existential types

Witness	Predicate	Existential Type	Sort	Introduction
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\Sigma x : A.P\ x$	$: \texttt{Type}$	$(x; y)$
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\{x : A\ \&\ P\ x\}$	$: \texttt{Type}$	$\texttt{existT}\ x\ y$

Erasure of existential types

Witness	Predicate	Existential Type	Sort	Introduction
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\Sigma x : A.P\ x$	$: \texttt{Type}$	$(x; y)$
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\{x : A\ \&\ P\ x\}$	$: \texttt{Type}$	`existT` $x\ y$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\{x : A \mid P\ x\}$	$: \texttt{Type}$	`exist` $x\ p$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\exists x : A, P\ x$	$: \texttt{Prop}$	`ex_intro` $x\ p$

Erasure of existential types

Witness	Predicate	Existential Type	Sort	Introduction
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\Sigma x : A. P\ x$	$: \texttt{Type}$	$(x; y)$
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\{x : A\ \&\ P\ x\}$	$: \texttt{Type}$	`existT` $x\ y$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\{x : A \mid P\ x\}$	$: \texttt{Type}$	`exist` $x\ p$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\exists x : A, P\ x$	$: \texttt{Prop}$	`ex_intro` $x\ p$

Examples

$$\mathcal{E}(\lambda(b : \mathsf{bool})(p : \mathsf{is\_true}\ b).\ (b; p)) \quad = \quad \lambda(b : \Box)(p : \Box).\ (b; p)$$

Erasure of existential types

Witness	Predicate	Existential Type	Sort	Introduction
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\Sigma x : A . P\ x$	$: \texttt{Type}$	$(x; y)$
$A : \texttt{Type}$	$P : A \to \texttt{Type}$	$\{x : A\ \&\ P\ x\}$	$: \texttt{Type}$	$\texttt{existT}\ x\ y$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\{x : A \mid P\ x\}$	$: \texttt{Type}$	$\texttt{exist}\ x\ p$
$A : \texttt{Type}$	$P : A \to \texttt{Prop}$	$\exists x : A, P\ x$	$: \texttt{Prop}$	$\texttt{ex\_intro}\ x\ p$

Examples

$$\begin{array}{lcl}
\mathcal{E}(\lambda(b : \mathsf{bool})(p : \mathsf{is\_true}\ b).\ (b; p)) & = & \lambda(b : \square)(p : \square).\ (b; p) \\
\mathcal{E}(\lambda(n : \mathbb{N})(H : \mathsf{0} \leq \mathsf{S}\ n).\ \mathsf{exist}\ n\ H) & = & \lambda n.\ \texttt{exist}\ n\ \square \\
\mathcal{E}(\lambda(n : \mathbb{N})(H : \mathsf{0} \leq \mathsf{S}\ n).\ \texttt{ex\_intro}\ n\ H) & = & (\lambda n.\ \square) = \square
\end{array}$$

Erasure Summary

Erasure produces untyped terms from typed CIC terms with just the computational content.

- This interpretation preserves the computability properties of the term: e.g. complexity and normal forms for extracted inductive values.
- A detailed account can be found in P. Letouzey's thesis (Letouzey, 2004), where preservation of types for ML/Haskell extraction is also studied.
- The erasure procedure was proven correct (preserving observations) in Coq recently by Sozeau et al. (2020)

Outline

Variants of nth

- Partial function

```
Fail Fixpoint nth {A} (l : list A) (n : nat) : A :=
match l, n with
| nil, n ⇒ _
| cons x xs, 0 ⇒ x
| cons _ xs, S n ⇒ nth xs n
end.
```

- Encoding partiality with the option type

```
Fixpoint nth_option {A} (l : list A) (n : nat) : option A :=
  match l, n with
  | nil, _ ⇒ None
  | cons x xs, 0 ⇒ Some x
  | cons _ xs, S n ⇒ nth_option xs n
  end.
```

```
Lemma nth_in_domain {A} (l : list A) n : n < length l →
  ∃ a, nth_option l n = Some a. Proof. .... Qed.
```

A total version

- Using a default value in place of exceptions

```
Fixpoint nth_default {A} (d : A) (l : list A) (n : nat) : A :=
  match l, n with
  | nil, _ ⇒ d
  | cons x xs, 0 ⇒ x
  | cons _ xs, S n ⇒ nth_default d xs n
  end.

Lemma nth_default_map {A B} (l : list A) n (f : A → B) d :
  f (nth_default d l n) = nth_default (f d) (map f l) n.
Proof. Admitted.

Definition nth_default_bug :=
  nth_default 0 [] 3.
```

A correct by construction version

- The total function, with types depending on values:

```
Example nth : ∀ A (l : list A),
  {n : nat | n < length l} → A.
Proof.
  move⇒ A. elim ⇒ [ |x l IHl].
  - move⇒ [n H].
    exfalso. inversion H.
  - move⇒ [[ | n'] H].
    + simpl in *. exact x.
    + apply IHl. ∃ n'. simpl in *; lia.
Defined.

Extraction nth.
```

- The computational content is mixed with the proof: not clear we return the “right” *A*!
- The erasure is however the same as ML's nth

Subset types

Recall subset types are sigma types where the second component is a proposition:
$\{\ x : A \mid P\ \} \leftrightarrow$ *sigma* A (fun $x : A \Rightarrow P$)

- Constructor:

```
Check (exist : ∀ A (P : A → Prop),
           ∀ x : A, P x → { x : A | P x }).
```

- Projections:

```
Check (proj1_sig : ∀ A P, {x : A | P x} → A).
Check (proj2_sig : ∀ A P (p : {x : A | P x}),
  P (proj1_sig p)).
```

- Binding notation:

```
Check (proj2_sig : ∀ A P (x : A | P x),
  P (proj1_sig x)).
```

Examples

```
Check (@exist nat (fun x : nat ⇒ x = x) 0 eq_refl
        : { x : nat | x = x }).
Check exist (fun x : nat ⇒ x = x) (S 0) eq_refl
        : { x : nat | x = x }.
Check exist (fun x : nat ⇒ x ≤ S x) (S 0) (le_S _ _ (le_n 1))
        : { x : nat | x ≤ S x }.
Check (fun x : { x : nat | x < 42 } ⇒ ‘ x)
    : { x : nat | x < 42 } → nat.
```

Strong specifications

```
Definition euclid_spec :=
  ∀ (x : nat) (y : nat | 0 < y),
    { (q, r) : nat × nat | x = q × ‘y + r }.
```

Program (Sozeau, 2007)

- A tool to program with *subset* types (inspired by PVS). In the programming literature (F*, Liquid Haskell, ...): refinement types / liquid types
- Coq's type system + the rules:

$$\frac{\Gamma \vdash t : \{x : A \mid P\}}{\Gamma \vdash t : A}\ \textsc{Sub-Proj}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma, x : A \vdash P : \texttt{Prop}}{\Gamma \vdash t : \{x : A \mid P\}}\ \textsc{Sub-Proof}$$

  Sub-Proj is an implicit projection/coercion. Note that Sub-Proof does *not* require a proof term.
- An elaboration to Coq terms with holes for the missing proof terms.

Program II

- An elaboration to Coq terms with holes for the missing proof terms.

$$\frac{\Gamma \vdash t : \{x : A \mid P\}}{\Gamma \vdash \mathrm{proj1\_sig}\ t : A}\ \textsc{Sub-Proj}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma, x : A \vdash P : \mathtt{Prop} \quad \Gamma \vdash ?p : P\ t}{\Gamma \vdash \mathsf{exist}\ t\ ?p : \{x : A \mid P\}}\ \textsc{Sub-Proof}$$

- This inserts projections and coercions with *proof obligations* ?*p* everywhere needed.
- The translation is correct for a type theory with proof-irrelevant `Prop` (Sozeau, 2008).

Examples

```
Require Import Arith Program.
```

Notation ! expresses an unreachable point of the program. It abbreviates an application of False_rect.

```
Program Definition imposs (H : 0 = 1) : False := !.
```

The Sub-Proof rule at work:

```
Program Definition exists_nonzero :
  { x : nat | 0 < x } := 1.
```

If the specification is not inhabited then some obligations will remain unprovable.

```
Program Definition exists_lt_zero :
 { x : nat | x < 0 } := 0.
Admit Obligations.
```

Examples

Using a match expression and !, let's implement a total head function:

```
Program Definition head {A}
    (l : list A | l ≠ []) : A :=
    match l with
    | [] ⇒ !
    | x :: xs ⇒ x
    end.
```

Now look at its extraction to OCaml:

```
Eval cbv beta zeta delta [head] in head.
Extraction head.
```

```
let head = function
| Nil → assert false
| Cons (x, _) → x
```

Decorating terms

The definition of head is actually: h

$$
\begin{array}{l}
\lambda\ (A : \texttt{Type})\ (l : \{l : \text{list } A \mid l \neq []\}), \\
\qquad \texttt{match}\ l.1\ \texttt{as}\ l'\ \texttt{return}\ (l' = l.1 \to A)\ \texttt{with} \\
\qquad \mid [] \Rightarrow \lambda\ Heq\_l : [] = l.1,\ ! \\
\qquad \mid hd :: wildcard' \Rightarrow \lambda\ \_ : hd :: wildcard' = l.1,\ hd \\
\qquad \texttt{end}\ \text{eq\_refl}
\end{array}
$$

where head_obligation_1 is of type:

$$\forall\ (A : \texttt{Type})\ (l : \{l : \text{list } A \mid l \neq []\}),\ [] = l.1 \to \text{False}$$

- Program automatically *generalizes* by equalities between the original matched object and the patterns.
- Provides the necessary *logical* information from the program, to solve obligations.

A safe lookup function

```
Program Fixpoint safe_nth {A} (l : list A)
         (n : nat | n < (length l)) : A := exercise.

Eval unfold safe_nth in safe_nth.

Extraction safe_nth.
```

The sumbool type

Coq includes a special disjunction of two propositions A and B but which is computational: that is we can know in a program which of the left or right branch is taken.

```
Inductive sumbool (A B : Prop) : Set :=
| left : A → sumbool A B
| right : B → sumbool A B.
Arguments left {A B}.
Arguments right {A B}.
```

```
Notation " { A } + { B } " := (sumbool A B) : type_scope.
```

The dec combinator allows to *reflect* a test on a boolean in the types.

```
Definition dec : ∀ b : bool, { b = true } + { b = false } :=
  fun b ⇒ match b return { b = true } + { b = false } with
              | true ⇒ left eq_refl
              | false ⇒ right eq_refl
              end.
```

Reflecting boolean tests

By default, Program does not do the generalization by equalities for boolean tests:

```
Program Definition f (n m : nat) : { n = m } + { n ≠ m } :=
  if n =? m then left _ else right _.
  Next Obligation.  No hypothesis to show (n = m)  Abort.
```

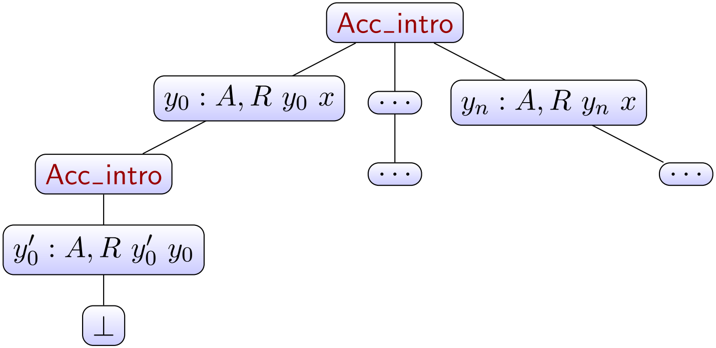
The dec combinator allows to *reflect* the test in the types:

```
Program Definition g (n m : nat) : { n = m } + { n ≠ m } :=
  if dec (n =? m) then left _ else right _.
Next Obligation.  One hypothesis n ?= m = true    by apply:
beq_nat_true. Qed.
Next Obligation.  One hypothesis n ?= m = false    by apply:
beq_nat_false. Qed.
```

Outline

Accessibility

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
  Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x
```



Every inductive value is acyclic: we cannot infinitely descend along the tree.

Well-founded recursion

Recursor definition based on Acc:

```
Section FixWf.
  Context (A : Type) (R : A → A → Prop) (P : A → Type).
  Context (F : ∀ x : A, (∀ y : A, R y x → P y) → P x).

  Definition well_founded := ∀ x, Acc R x.
```

Well-founded recursion

Recursor definition based on Acc:

```
Section FixWf.
  Context (A : Type) (R : A → A → Prop) (P : A → Type).
  Context (F : ∀ x : A, (∀ y : A, R y x → P y) → P x).

  Definition well_founded := ∀ x, Acc R x.

  Fixpoint Fix (x : A) (a : Acc R x) {struct a} : P x :=
    F x (fun (y : A) (h : R y x) ⇒ Fix y (Acc_inv a h)).
```

Well-founded recursion

Recursor definition based on Acc:

```
Section FixWf.
  Context (A : Type) (R : A → A → Prop) (P : A → Type).
  Context (F : ∀ x : A, (∀ y : A, R y x → P y) → P x).

  Definition well_founded := ∀ x, Acc R x.

  Fixpoint Fix (x : A) (a : Acc R x) {struct a} : P x :=
    F x (fun (y : A) (h : R y x) ⇒ Fix y (Acc_inv a h)).

  Definition FixWf (wf : well_founded) : ∀ x : A, P x :=
    fun x ⇒ Fix x (wf x).
End FixWf.
```

Well-founded recursion

Compare with the Y combinator such that $Y F \equiv F(Y F)$.
Fix is structurally recursive on the accessibility proof:

- The unfolding reduction is
  $\mathrm{Fix}\ F\ (\mathsf{Acc\_intro}\ p) \equiv F\ (\mathrm{Fix}\ F\ p)$
- The proof p guards the unfolding of the fixpoint
- It will get stuck if we provide an axiomatic proof
- Most of the time, one provides closed proofs of accessibility, e.g. for `lt` on natural numbers, the proof of `Acc lt x` is done by induction on `x`

Well-founded relations

We now get all the power of the logic and well-founded relations to prove termination:

- Lexicographic orderings
- Measures (size, depth, combined measures of multiple arguments)
- Ordinal measures
- Nested recursions outside the scope of the syntactic guard condition

Library `Coq.Wf` contains combinators to construct well-founded relations.

Program Example

```
Program Fixpoint euclid (x : nat)
        (y : nat | 0 < y)
        { wf lt x } :
  { (q, r) : nat × nat | x = q × y + r } :=
  if dec (Nat.ltb x y) then (0, x)
  else let 'pair q r := euclid (x - y) y in
       (S q, r).
Next Obligation. move/Nat.ltb_ge: H. lia. Qed.
Next Obligation.
  move: Heq_anonymous.
  case: euclid ⇒ [[q' r'] H']; simpl in * ⇒ [= → → ].
  move/Nat.ltb_ge: e. lia.
Defined.
```

Conclusion

- Dependent types allow to specify code as desired by adding pre- and post-conditions using subset types.
- We can use Program to reflect the computational content at the logical level using the equality and accessibility types, to implement dependent functions and justify termination.
- By erasure, this decoration, which consists only of proofs in `Prop`, disappears. It leaves us with the pure computational content we wrote at the beginning.

Exercises: dtp_exercises.v

Next episode: programming and proving with indexed inductive types and dependent pattern-matching.

Bibliography

Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud, July 2004.

Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions. (Program Extraction in the Calculus of Constructions)*. PhD thesis, Paris Diderot University, France, 1989.

Matthieu Sozeau. Program-ing Finger Trees in Coq. In *ICFP'07*, pages 13–24, Freiburg, Germany, 2007. ACM Press.

Matthieu Sozeau. *Un environnement pour la programmation avec types dépendants*. PhD thesis, Université Paris 11, Orsay, France, December 2008.

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. *Coq Coq Correct! Verifying Typechecking and Erasure for Coq, in Coq*. *Proceedings of the ACM on Programming Languages*, 4 (POPL), January 2020.