Dependently typed programs with propositions

Pierre Letouzey

Dependent types

- In Coq, types may be parameterized by values.
- Such types are called *dependent*.

Example of dependent types

- Arrays of size n, perfect binary trees of depth p, ...

Example of dependent types

- Arrays of size n, perfect binary trees of depth p, ...
- Logical formulas!
  - Universally quantified theorems are functions
  - Application is instanciation
  - Propositions are types, proofs are elements

Example of dependent types

- Arrays of size n, perfect binary trees of depth p, ...
- Logical formulas!
  - Universally quantified theorems are functions
  - Application is instanciation
  - Propositions are types, proofs are elements
- Partial functions handled via (*preconditions*):
  - `pred_safe : forall x:nat, x<>0 -> nat`

Example : a total predecessor

```
Definition pred_safe (n:nat) : n<>0 -> nat :=
 match n with
  | 0 => fun Hn => False_rect _ (Hn (eq_refl 0))
  | S n => fun _ => n
 end.

(* or *)
```

Example : a total predecessor

```
Definition pred_safe (n:nat) : n<>O -> nat :=
 match n with
  | O => fun Hn => False_rect _ (Hn (eq_refl O))
  | S n => fun _ => n
 end.

(* or *)

Definition pred_safe : forall n, n<>O -> nat.
Proof.
 intros n Hn. destruct n.
  destruct Hn; reflexivity.
  apply n.
Qed.
```

Example : bounded numbers and arrays

```
Inductive bnat (n : nat) : Type :=
  cb : forall m, m < n -> bnat n.
```

Example : bounded numbers and arrays

```
Inductive bnat (n : nat) : Type :=
  cb : forall m, m < n -> bnat n.

Inductive array (n : nat) : Type :=
  ca : forall l : list Z, length l = n -> array n.
```

Example : bounded numbers and arrays

```
Inductive bnat (n : nat) : Type :=
  cb : forall m, m < n -> bnat n.

Inductive array (n : nat) : Type :=
  ca : forall l : list Z, length l = n -> array n.
(* or *)
Inductive vect : nat -> Type :=
 | vnil : vect 0
 | vcons : forall n, Z -> vect n -> vect (S n).
```

Example : bounded numbers and arrays

```
Inductive bnat (n : nat) : Type :=
  cb : forall m, m < n -> bnat n.

Inductive array (n : nat) : Type :=
  ca : forall l : list Z, length l = n -> array n.
(* or *)
Inductive vect : nat -> Type :=
 | vnil : vect 0
 | vcons : forall n, Z -> vect n -> vect (S n).
```

We can build a total nth function:

```
Definition vect_nth : forall n, vect n -> bnat n -> Z.
Proof. ... Defined.
```

Example : perfect binary tree

```
Fixpoint ptree (n:nat) : Type :=
 match n with
  | O => Z
  | S n => (ptree n * ptree n)%type
 end.

Check ((1,2),(3,4)) : ptree 2.

Fixpoint sum_ptree n : ptree n -> Z :=
 match n with
  | O => fun t => t
  | S n =>
    fun t => let (g,d):=t in sum_ptree n g + sum_ptree n d
 end.

Compute (sum_ptree 2 ((1,2),(3,4))).
```

A generic notion of type with restriction

- `bnat` and `array` are quite similar:
  numbers, or lists, *such that* some property hold.

A generic notion of type with restriction

- `bnat` and `array` are quite similar:
  numbers, or lists, *such that* some property hold.
- Coq's generic way to build types with restriction:

```
{ x : A | P x }
```

A generic notion of type with restriction

- `bnat` and `array` are quite similar:
  numbers, or lists, *such that* some property hold.
- Coq's generic way to build types with restriction:

  ```
  { x : A | P x }
  ```
- For instance:

  ```
  Definition bnat n := { m | m < n }.
  Definition array n := { l : list Z | length l = n }.
  ```

A generic notion of type with restriction

- Behind the nice `{ | }` notation, the `sig` type:

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
  exist : forall x : A, P x -> sig P
```

A generic notion of type with restriction

- Behind the nice `{ | }` notation, the `sig` type:

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
  exist : forall x : A, P x -> sig P
```

- To access the element, or the proof of the property:
  - `proj1_sig`, `proj2_sig`
  - or directly `let (x,p) := ... in ...`
  - or in proof mode via the tactics `case`, `destruct`, ...

A generic notion of type with restriction

- Behind the nice `{ | }` notation, the `sig` type:

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
  exist : forall x : A, P x -> sig P
```

- To access the element, or the proof of the property:
  - `proj1_sig`, `proj2_sig`
  - or directly `let (x,p) := ... in ...`
  - or in proof mode via the tactics `case`, `destruct`, ...

- To build a `sig` interactively: the `exists` tactic.

A example: bounded successor

- As a function:

```
Definition bsucc n : bnat n -> bnat (S n) :=
 fun m => let (x,p):= m in exist _ (S x) (lt_n_S _ _ p)
```

A example: bounded successor

- As a function:

```
Definition bsucc n : bnat n -> bnat (S n) :=
 fun m => let (x,p):= m in exist _ (S x) (lt_n_S _ _ p)
```

- Via tactics:

```
Definition bsucc n : bnat n -> bnat (S n).
Proof.
 intros m. destruct m as [x p]. exists (S x).
 auto with arith.
Defined.
```

A example: bounded successor

- As a function:

```
Definition bsucc n : bnat n -> bnat (S n) :=
 fun m => let (x,p):= m in exist _ (S x) (lt_n_S _ _ p)
```

- Via tactics:

```
Definition bsucc n : bnat n -> bnat (S n).
Proof.
 intros m. destruct m as [x p]. exists (S x).
 auto with arith.
Defined.
```

- Via the Program framework :

```
Program Definition bsucc n : bnat n -> bnat (S n) :=
 fun m => S m.
Next Obligation.
 destruct m. simpl. auto with arith.
Qed.
```

General shape of a rich specification

- With `sig`, we can hence express also *post-conditions*:

```
forall x, P x -> { y | Q x y }
```

- Adapt to your needs: multiple arguments or outputs (`y` can be a tuple) or pre or post (`Q` can be a conjonction).
- Apart with Program, `sig` is rarely used for pre-conditions.

The special case of boolean output

- We could handle boolean outputs via `sig`:

```
Definition rich_beq_nat :
 forall n m : nat, { b : bool | b = true <-> n=m }.
```

The special case of boolean output

- We could handle boolean outputs via `sig`:

```
Definition rich_beq_nat :
 forall n m : nat, { b : bool | b = true <-> n=m }.
```

- More convenient: `sumbool`, a type with two alternatives and annotations for characterizing them.

```
Definition eq_nat_dec :
 forall n m : nat, { n=m }+{ n<>m }.
```

The special case of boolean output

- Behind the `{ }+{ }` notation, the `sumbool` type:

```
Inductive sumbool (A B : Prop) : Type :=
 | left : A -> {A}+{B}
 | right : B -> {A}+{B}
```

The special case of boolean output

- Behind the `{ }+{ }` notation, the `sumbool` type:

```
Inductive sumbool (A B : Prop) : Type :=
 | left : A -> {A}+{B}
 | right : B -> {A}+{B}
```

- To analyse a `sumbool` construction:
  - directly via `if ... then ... else ...`
  - or `bool_of_sumbool`
  - or in proof mode via the tactics `case`, `destruct`, ...

The special case of boolean output

- Behind the `{ }+{ }` notation, the `sumbool` type:

```
Inductive sumbool (A B : Prop) : Type :=
 | left : A -> {A}+{B}
 | right : B -> {A}+{B}
```

- To analyse a `sumbool` construction:
  - directly via `if ... then ... else ...`
  - or `bool_of_sumbool`
  - or in proof mode via the tactics `case`, `destruct`, ...

- To build a `sumbool` interactively: the `left` and `right` tactics.

Decidability result

- Many Coq functions are currently formulated this way:
  `eq_nat_dec`, `Z_eq_dec`, `le_lt_dec`, ...
  (see `SearchAbout sumbool`).

Decidability result

- Many Coq functions are currently formulated this way:
  eq_nat_dec, Z_eq_dec, le_lt_dec, ...
  (see `SearchAbout sumbool).`
- For instance:

```
Definition le_lt_dec n m : { n <= m }+{ m < n }.
Proof.
 induction n.
 left. auto with arith.
 destruct m.
  right. auto with arith.
  destruct (IHn m); [left | right]; auto with arith.
Defined.
```

Decidability result

- Many Coq functions are currently formulated this way: `eq_nat_dec`, `Z_eq_dec`, `le_lt_dec`, ... (see `SearchAbout sumbool).`
- For instance:

```
Definition le_lt_dec n m : { n <= m }+{ m < n }.
Proof.
 induction n.
 left. auto with arith.
 destruct m.
  right. auto with arith.
  destruct (IHn m); [left | right]; auto with arith.
Defined.
```

- For equality, see tactic `decide equality`.

Why program with logical annotations ?

- To handle partial functions, instead of dummy values at undefined spots or option types

Why program with logical annotations ?

- To handle partial functions, instead of dummy values at undefined spots or option types
- To satisfy precisely an interface (see exercise on sets)

Why program with logical annotations ?

- To handle partial functions, instead of dummy values at undefined spots or option types
- To satisfy precisely an interface (see exercise on sets)
- To have all-in-one objects (handy for `destruct`).

Why program with logical annotations ?

- To handle partial functions, instead of dummy values at undefined spots or option types
- To satisfy precisely an interface (see exercise on sets)
- To have all-in-one objects (handy for `destruct`).
- To have the right justifications when doing general recursion

Why program with logical annotations ?

- To handle partial functions, instead of dummy values at undefined spots or option types
- To satisfy precisely an interface (see exercise on sets)
- To have all-in-one objects (handy for `destruct`).
- To have the right justifications when doing general recursion

Additional remarks:

- Computations in Coq may then be tricky and/or slower and/or memory hungry.

Why program with logical annotations ?

- To handle partial functions, instead of dummy values at undefined spots or option types
- To satisfy precisely an interface (see exercise on sets)
- To have all-in-one objects (handy for `destruct`).
- To have the right justifications when doing general recursion

Additional remarks:

- Computations in Coq may then be tricky and/or slower and/or memory hungry.
- Pure & efficient Ocaml/Haskell code can be obtained by extraction.

Why program with logical annotations ?

- To handle partial functions, instead of dummy values at undefined spots or option types
- To satisfy precisely an interface (see exercise on sets)
- To have all-in-one objects (handy for `destruct`).
- To have the right justifications when doing general recursion

Additional remarks:

- Computations in Coq may then be tricky and/or slower and/or memory hungry.
- Pure & efficient Ocaml/Haskell code can be obtained by extraction.
- Definitions by tactics are dreadful, Program helps but is still quite young.

Why program with logical annotations ?

- To handle partial functions, instead of dummy values at undefined spots or option types
- To satisfy precisely an interface (see exercise on sets)
- To have all-in-one objects (handy for `destruct`).
- To have the right justifications when doing general recursion

Additional remarks:

- Computations in Coq may then be tricky and/or slower and/or memory hungry.
- Pure & efficient Ocaml/Haskell code can be obtained by extraction.
- Definitions by tactics are dreadful, Program helps but is still quite young.
- Instead of destructing rich objects, other technics can also be convenient (iff, reflect).

Why specific constructs like sig and sumbool ?

- `{ x | P x }` is a clone of `exists x, P x`.
  Both regroup a witness and a justification.

Why specific constructs like sig and sumbool ?

- `{ x | P x }` is a clone of `exists x, P x`.
  Both regroup a witness and a justification.
- Similarly, `{ A }+{ B }` is a clone of `A \/ B` .

Why specific constructs like sig and sumbool ?

- `{ x | P x }` is a clone of `exists x, P x`.
  Both regroup a witness and a justification.
- Similarly, `{ A }+{ B }` is a clone of `A \/ B` .

In fact, `sig`/`sumbool` live in a different world than `ex`/`or`.

The two worlds of Coq

In Coq, two separate worlds (technically, we speak of *sorts*):

The two worlds of Coq

In Coq, two separate worlds (technically, we speak of *sorts*):

- The “logical” world

The two worlds of Coq

In Coq, two separate worlds (technically, we speak of *sorts*):

- The “logical” world
  - a proof : a statement : `Prop`

The two worlds of Coq

In Coq, two separate worlds (technically, we speak of *sorts*):

- The "logical" world
  - a proof : a statement : `Prop`
  - `or_introl _ I` : `True\/False` : `Prop`

The two worlds of Coq

In Coq, two separate worlds (technically, we speak of *sorts*):

- The “logical” world
  - a proof : a statement : `Prop`
  - `or_introl _ I` : `True\/False` : `Prop`
- The “informative” world (everything else).

The two worlds of Coq

In Coq, two separate worlds (technically, we speak of *sorts*):

- The “logical” world
  - a proof : a statement : `Prop`
  - `or_introl _ I` : `True\/False` : `Prop`
- The “informative” world (everything else).
  - a program : a type : `Type`

The two worlds of Coq

In Coq, two separate worlds (technically, we speak of *sorts*):

- The "logical" world
  - a proof : a statement : `Prop`
  - `or_introl _ I` : `True\/False` : `Prop`
- The "informative" world (everything else).
  - a program : a type : `Type`
  - `O` : `nat` : `Type`

The two worlds of Coq

In Coq, two separate worlds (technically, we speak of *sorts*):

- The “logical” world
  - a proof : a statement : `Prop`
  - `or_introl _ I` : `True\/False` : `Prop`
- The “informative” world (everything else).
  - a program : a type : `Type`
  - `O` : `nat` : `Type`
  - `pred` : `nat->nat` : `Type`

The two worlds of Coq

Usually we program in Type and make proofs in Prop. But that's just a convention. We can build functions by tactics, or reciprocally "program" a proof:

```
Definition or_sym A B : A\/B -> B\/A :=
 fun h => match h with
  | or_introl a => or_intror _ a
  | or_intror b => or_introl _ b
 end.
```

The similarity between proofs and programs, between statements and types is called the Curry-Howard isomorphism.

The two worlds of Coq

In Coq, a rigid separation between Prop and Type:

Logical parts should not interfere with computations in Type.

```
Definition nat_of_or A B : A\/B -> nat :=
 fun h => match h with
  | or_introl _ => 0
  | or_intror _ => 1
 end.
Error: ... proofs can be eliminated only to build proofs.
```

Idea: proofs are there only as guarantee, we're interested only in their *existence*, we consider them as having no *computational content*.

Extraction

Coq's strict separation between Prop and Type is the fondation of the *extraction* mechanism: roughly, logical parts are removed, pruned programs still compute the same outputs.

```
Coq < Recursive Extraction le_lt_dec.

type nat = O | S of nat
type sumbool = Left | Right

(** val le_lt_dec : nat -> nat -> sumbool **)

let rec le_lt_dec n m =
  match n with
    | O -> Left
    | S n0 -> (match m with
                 | O -> Right
                 | S m0 -> le_lt_dec n0 m0)
```