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, numbers smaller than n, ...
- 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 : predecessor with precondition

```
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 *)

Definition pred_safe : forall n, n<>0 -> 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.

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

We can build a total nth function:

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

Coq provides another approach to boolean arrays: `Bvector`, not studied here

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
```

- It is a dependent pair
- 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.

An example: bound widening

- 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.
```

General shape of a rich specification

- With `sig`, we can also express *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).
- `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 }.
```

- 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}
  ```

- To use 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 results

- Many Coq functions are currently formulated this way: `eq_nat_dec`, `Z_eq_dec`, `le_lt_dec`, ...
  (see `Search 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
- 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, slower, or memory inefficient.
- Pure & efficient Ocaml/Haskell code can be obtained by extraction.
- Definitions by tactics are unreadable. The `Program` command set is a good alternative.

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 "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: A proof is a guarantee, it does not participate in computation only in their *existence*, we consider them as having no *computational content*.

Extraction

Prop and Type separation used for *extraction*
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)
```

Well-founded recursion

- Binary relations may be well-founded
  - all descending chains terminate
- A well-founded relation can serve to justify a recursive function

```
Fix : forall (A : Type) (R : A -> A -> Prop),
       well_founded R ->
       forall P : A -> Type,
       (forall x : A, (forall y : A, R y x -> P y)
          -> P x) ->
       forall x : A, P x
```

- The last but one argument describes an algorithm
- Recursive calls are restricted to predecessors for `R`

Example well-founded recursive algorithm

Compute x modulo d :
if $x < d$ then x otherwise $(x - d)$ modulo d

```
Lemma decr : forall x d, d <> 0 -> d <= x -> x - d < x.
Proof. intros; omega. Qed.

Definition mod'_F : forall x : nat,
 (forall y, y < x -> forall d, d <> 0 ->  r | r < d) ->
   forall d, d <> 0 -> {r | r < d} :=

 fun x mod' d dn0 =>
   match le_lt_dec d x with
     left h => mod' (x - d) (decr x d dn0 h) d dn0
   | right h' => exist _ x (h' : x < d)
   end.
```

Completing the well-founded definition

Finish by apply `Fix` to all relevant pieces.

```
Definition mod' : nat -> forall y, y <> 0 -> r | r < y
  := Fix lt_wf (fun _ => forall y, y <> 0 -> r | r < y)
                     mod'_F.

Compute mod' 30 7.
exist (fun r : nat => r < 7) 2
         (gt_le_S 2 7 ...)
     : {r : nat | r < 7}
```

Well founded definitions as proofs

```
Definition gcd : nat -> nat -> nat.
apply (Fix lt_wf (fun _ => nat -> nat)).
====================
forall x, (forall y, y < x -> nat -> nat) -> nat -> nat
intros x gcd' y.
====================
 nat
destruct (eq_nat_dec x 0) as [x0 | xn0].
x0 : x = 0
====================
 nat
 exact y.
x0 : x <> 0
====================
 nat
exact (gcd' v hv x).
Defined.
```

Well founded definitions and strong specifications

- Unlike `gcd`, well founded recursive function are better with strong specifications
- For instance one should have defined:

```
mod : forall x d, d <> 0 ->
    {r | exists q, x = q * d + r / r < d}.

gcd : forall x y, {d |
   (exists q1, exists q2, x = q1 * d /\ y = q2 * d) /\
   (exists u1, exists u2, d = u1 * x - u2 * y \/
      d = u1 * y - u2 * x)}
```