Dependently Typed Programming in the Coq Proof Assistant

Nicolas Magaud
School of Computer Science and Engineering
The University of New South Wales

Dagsthul Seminar
12-17.09.2004

Outline

- The Coq Proof Assistant
  - A System based on the Calculus of Inductive Constructions
  - Designed to Write Programs and Reason about them
  - Distinction between Logic and Computation (Set/Prop)
  - Extraction Mechanism
- Dependently Typed Programming
  - Writing Fully Specified Programs
  - Describing Partial Functions, Well-founded Recursion
  - Dependently Typed Programs and their Properties
    or Coq as any other dependently typed programming language.

Functions Definitions

- Defining functions (only total functions)
  - Structural recursive definitions:
    Pattern Matching and Guarded Fixpoint
  - One can also define functions by well-founded recursion.
- Example

```
Inductive nat : Set := O : nat | S : nat -> nat.
Fixpoint plus (n m:nat) {struct n} : nat :=
  match n with | O => m
               | S p => S (plus p m)
  end.
```

- Computational behaviour (ι-reduction)

plus O $m \xrightarrow{\iota} m$ plus (S p) $m \xrightarrow{\iota}$ (S (plus p m))

Fully Specified Programs

- A predecessor function for natural numbers (pred.v)
  - initial implementation: as a function of type nat $\rightarrow$ nat and maybe a comment about what we do for 0
  - refined into a function with a precondition:
    $\forall n : \mathsf{nat}, n \neq 0 \rightarrow \mathsf{nat}$
  - eventually as a fully-specified function
    $\forall n : \mathsf{nat}, \{p : \mathsf{nat} \mid n = (\mathsf{S}\ p)\} + \{n = 0\}$
    This type contains all the information we want to know about pred, especially the computed term p as well as its link with n.
- Especially useful to establish properties of functions for which reasoning by induction will be difficult.

An Academic Example

- Computing elements of the Fibonacci sequence

```
Fixpoint fib (n : nat) : nat :=
  match n with
  | O => 0
  | S p => match p with
           | O => 1
           | S q => fib p + fib q
           end
  end.
```

- How to define it with integers (Z) rather than natural numbers ?
- We'll need well-founded induction and partial functions

Fibonacci with integers

- fib : $\forall n : \mathbb{Z}, 0 \leq n \rightarrow \mathbb{Z}$
- Recursion using accessibility and a well-founded order

```
Inductive Acc (A: Set) (R: A -> A -> Prop): A -> Prop :=
Acc_intro:
  forall x: A, (forall y: A, R y x -> Acc R y) -> Acc R x
```

- Build a higher-order function (one-step computation)

  This is always a dependently typed function.
- Case analysis on a *strongly specified version of boolean expressions*

  Z_le_lt_eq_dec : $\forall x\ y : Z, (x \ \leq \ y) \rightarrow \{(x \ < \ y)\} + \{x = y\}$

  Not only we do case analysis on whether $x < y$ and $x = y$, but we also get it as an assumption in the corresponding branch.

Definitions by well-founded recursion


```
Definition F
  (n : Z)
  (g : forall m : Z, Zwf 0 m n -> (0 <= m) -> Z)
  (h : (0 <= n)) : Z :=
  match Z_le_lt_eq_dec 0 n h with
  | left h' =>
      match Z_le_lt_eq_dec 1 n (t1 _ h') with
      | left h'' =>
          (g (n - 1)  (t2 _ h') (st _ _ (t2 _ h')) +
           g (n - 2)  (t3 _ h'') (st _ _ (t3 _ h'')))
      | right _ => 1
      end
  | right _ => 0
  end.
```

Reasoning about these programs

- Generating associated Fix-point equations (Balaa and Bertot)

$$\begin{aligned}\mathsf{fib}\ 0\ h &= 0\\ \mathsf{fib}\ 1\ h &= 1\\ \mathsf{fib}\ (\mathsf{S}\ (\mathsf{S}\ n))\ h &= \mathsf{fib}\ (\mathsf{S}\ n)\ h' + \mathsf{fib}\ n\ h''\end{aligned}$$

  Handling functions with preconditions is a bit more complex.

- Alternative Approach:
  Recursion on a Ad-Hoc predicate (Bove and Capretta)

Dependently Typed Programs

- Defining vectors (a.k.a. dependent lists)

```
Inductive vect (A : Set) : nat -> Set :=
    vnil : vect A 0
  | vcons : forall n : nat, A -> vect A n -> vect A (S n).

Definition app: forall n m:nat,
    (vect A n) -> (vect A m) -> (vect A (plus n m)).
```

- But remember ! We are in a theorem prover...
- ...so we want to prove theorems about these objects.

```
associativity of append on vectors
forall n:nat, forall vn:(vect A n), (rev n (rev n vn) vn)=vn
```

Equality over Dependently Typed Terms

- Leibniz equality (as an inductive definition)

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  refl_equal : x = x
```

- This equality only allows to compare objects already known to be of the same type.

- Dependent Equality and John Major's Equality

```
Inductive JMeq (A:Set) (x:A) : forall B:Set, B -> Prop :=
    JMeq_refl : JMeq x x.
```

- Equality equipped with a special elimination principle:

```
forall (A:Set) (x y:A) (P:A -> Prop), P x -> JMeq x y -> P y.
```

- Let's see how it works in Coq ! (app.v)

Proof Development

- append:
  $\forall n\ m : \mathsf{nat}, \mathsf{vect}\ n \rightarrow \mathsf{vect}\ m \rightarrow \mathsf{vect}\ (n+m)$
- reverse :
  $\forall n : \mathsf{nat}, \mathsf{vect}\ n \rightarrow \mathsf{vect}\ n$
- how to prove
  $\forall n : \mathsf{nat}, \forall vn : (\mathsf{vect}\ n), \mathsf{reverse}\ n\ (\mathsf{reverse}\ n\ vn) = vn$?
- Trying to build reverse with append
  $\mathsf{reverse}\ x :: xs \rightarrow \mathsf{append}\ n\ 1\ xs\ x : (\mathsf{vect}\ (n+1))$
- however we would prefer $(\mathsf{vect}\ (\mathsf{S}\ n))$ for the recursive definition of reverse, hence we take a specific app_right function.
  (reverse.v, dep2.v)

Summary

- A few applications of dependent types to program in Coq
  - Fully Specified Functions
  - Building Functions using Well-founded Recursion
  - Describing Partial Functions
  - Actually Writing Dependently Typed Functions
- To what extend is it practicable to write dependently typed programs in such a framework ?
- We remain in the same framework to do the proofs.
- We can extract the datatypes and functions to Ocaml or Haskell.