Dependently Typied Programming
in the Coq Proof Assistant

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

Dagstuhl 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


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

- Computational behaviour ($\iota$-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 → nat and maybe a comment about what we do for 0
  - refined into a function with a precondition:
    \[ \forall n : \text{nat}, n \neq 0 \rightarrow \text{nat} \]
  - eventually as a fully-specified function
    \[ \forall n : \text{nat}, \{ p : \text{nat} \mid n = (S\ p) \} + \{ n = 0 \} \]
    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

- \texttt{fib} : \forall n : \mathbb{Z}, 0 \leq n \rightarrow \mathbb{Z}

- Recursion using accessibility and a well-founded order
  \texttt{Inductive Acc (A: Set) (R: A \rightarrow A \rightarrow \text{Prop}): A \rightarrow \text{Prop} :=}
  \texttt{Acc_intro:}
  \begin{align*}
  & \forall x : A, (\forall y : A, R y x \rightarrow \text{Acc} R y) \rightarrow \text{Acc} R x
  \end{align*}

- Build a higher-order function (one-step computation)
  This is \texttt{always} a dependently typed function.

- Case analysis on a \textit{strongly specified version of boolean expressions}
  \texttt{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.
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)

\[
\begin{align*}
\text{fib} \ 0 \ h &= 0 \\
\text{fib} \ 1 \ h &= 1 \\
\text{fib} \ (S \ (S \ n)) \ h &= \text{fib} \ (S \ n) \ h' + \text{fib} \ n \ h''
\end{align*}
\]

Handling functions with preconditions is a bit more complex.

- Alternative Approach:
  Recursion on a Ad-Hoc predicate (Bove and Capretta)
Defining vectors (a.k.a. dependent lists)

\[
\text{Inductive vect } (A : \text{Set}) : \text{nat} \to \text{Set} :=
\]
\[
\text{vnil : vect } A \ 0
\]
\[
| \text{vcons : forall } n : \text{nat}, A \to \text{vect } A \ n \to \text{vect } A \ (\text{S } n).
\]

Definition app: forall n m:nat,
\[
(\text{vect } A \ n) \to (\text{vect } A \ m) \to (\text{vect } A \ (\text{plus } n \ m)).
\]

- 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 Dependent Typed Terms

- Leibnitz equality (as an inductive definition)

  \[
  \text{Inductive } \text{eq} \ (A : \text{Type}) \ (x : A) : A \to \text{Prop} := \\
  \text{refl_equal} : x = x 
  \]

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

- Dependent Equality and John Major’s Equality

  \[
  \text{Inductive } \text{JMeq} \ (A:\text{Set}) \ (x:A) : \forall B:\text{Set}, B \to \text{Prop} := \\
  \text{JMeq_refl} : \text{JMeq} \ x \ x. 
  \]

- Equality equipped with a special elimination principle:

  \[
  \forall (A:\text{Set}) \ (x \ y:A) \ (P:A \to \text{Prop}) \ P \ x \to \text{JMeq} \ x \ y \to P \ y. 
  \]

- Let’s see how it works in Coq! (app.v)
Proof Development

- append:
  \[ \forall n \ m : \text{nat}, \text{vect} \ n \rightarrow \text{vect} \ m \rightarrow \text{vect} \ (n + m) \]

- reverse :
  \[ \forall n : \text{nat}, \text{vect} \ n \rightarrow \text{vect} \ n \]

- how to prove
  \[ \forall n : \text{nat}, \forall vn : (\text{vect} \ n), \text{reverse} \ n \ (\text{reverse} \ n \ vn) = vn ? \]

- Trying to build reverse with append
  \[ \text{reverse} \ x :: xs \rightarrow \text{append} \ n \ 1 \ xs \ x : (\text{vect} \ (n + 1)) \]

- however we would prefer (\text{vect} \ (S \ n)) for the recursive definition of reverse, hence we take a specific \text{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 Dependedently 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.