Introduction to dependent types in Coq

Yves Bertot

October 24, 2008

basic use of the Coq system

- In Coq, you can play with simple values and functions.
- The basic command is called Check, to verify if an expression is well-formed and learn what is its type.
- Check 3. 3 : nat
- Check plus. plus : nat $\rightarrow$ nat $\rightarrow$ nat
- Check plus 3 4. 3 + 4 : nat

Building your own function

- Check fun x:nat => 3 + x.
  fun x:nat => 3 + x : nat → nat
- use type inference when possible,
  Check fun x => 3 + x.
  fun x:nat => 3 + x : nat → nat
- Check fun (x:nat)(y:bool) => if y then x else 3.
  fun (x : nat) (y : bool) => if y then x else 3
  : nat → bool → nat

Applying functions

- Check (fun x => 3 + x) 4
  (fun (x : nat) => 3 + x) 4 : nat
- You can also force functions to compute
  Eval compute in (fun x => 3 + x) 4
  = 7 : nat

Proofs as functions, functions as proofs

- Using the Modus-Ponens rule: *if "A implies B" and "A" both hold, then we can deduce "B"*,

Proofs as functions, functions as proofs

- Using the Modus-Ponens rule: *if "A implies B" and "A" both hold, then we can deduce "B"*,
- The Moduls-Ponens rule transforms any proof of $A \Rightarrow B$ into a function mapping (the type of) proofs of A to (the type of) proofs of B, Try reading without the text in parentheses.

Proofs as functions, functions as proofs

- Using the Modus-Ponens rule: *if "A implies B" and "A" both hold, then we can deduce "B"*,
- The Moduls-Ponens rule transforms any proof of $A \Rightarrow B$ into a function mapping (the type of) proofs of A to (the type of) proofs of B, Try reading without the text in parentheses.
- Using the forall-elimination rule: *if* $\forall x : A, P$ *and* e *has type* A*, then we can deduce* $P[x \backslash e]$,
- The forall elimination rule transforms any proof of $\forall x : A, P$ into a function mapping any element e of A to a proof of $P[x \backslash e]$, P where x is replaced by e.

Proofs as functions, functions as proofs

- Using the Modus-Ponens rule: *if "A implies B" and "A" both hold, then we can deduce "B"*,
- The Moduls-Ponens rule transforms any proof of $A \Rightarrow B$ into a function mapping (the type of) proofs of A to (the type of) proofs of B, Try reading without the text in parentheses.
- Using the forall-elimination rule: *if* $\forall x : A, P$ *and* e *has type* A*, then we can deduce* $P[x\backslash e]$,
- The forall elimination rule transforms any proof of $\forall x : A, P$ into a function mapping any element e of A to a proof of $P[x\backslash e]$, P where x is replaced by e.
- When considering total functions, we also have the reverse:
- any function of type $A \rightarrow B$ can be used to prove $A \Rightarrow B$,

And for universal formulas?

- Simple types, as found in Ocaml or Haskell are not enough,
- The rest of this lecture is about new constructs for universal quantification.
- In the next slides, blue will be use for types.

The Curry-Howard Isomorphism

- Accept the existence of a type Prop whose elements are types,
- All elements of Prop are types of proofs,
- Thus if A and B are types of of proofs, then
- $A \rightarrow B$ is also a type of proof,
- Next, accept that a type of proofs is a "proposition",
- A proposition holds if it contains a proof.

Arrows in the Curry-Howard Isomorphims

- In this frame, assume A, B, and C are propositions,
- They are propositions,
- A function of type $A \rightarrow B$ maps any proof of A to a proof of B,
- It represents a proof of $A \Rightarrow B$.
- for some formulas, we can build function of that type directly,
- When you do this, you do give a proof!
- Example 1: `fun x:A => x` : $A \rightarrow A$

Arrows in the Curry-Howard Isomorphims

- In this frame, assume A, B, and C are propositions,
- They are propositions,
- A function of type $A \rightarrow B$ maps any proof of A to a proof of B,
- It represents a proof of $A \Rightarrow B$.
- for some formulas, we can build function of that type directly,
- When you do this, you do give a proof!
- Example 1: `fun x:A => x` : $A \rightarrow A$
- Example 2:
  `fun (f: A → B → C) (x: B) (y:A) => f y x`
  : $(A \rightarrow B \rightarrow C) \rightarrow (B \rightarrow A \rightarrow C)$

Various kinds of logics

- Can all usual tautologies be proved with pure functions?

Various kinds of logics

- Can all usual tautologies be proved with pure functions?
- No.
- Peirce's Formula: $((A \to B) \to A) \to A$

A	B	$A \to B$	$(A \to B) \to A$	$((A \to B) \to A) \to A$
T	T	T	T	T
T	F	F	T	T
F	T	F	F	T
F	F	T	F	T

Various kinds of logics

- Can all usual tautologies be proved with pure functions?
- No.
- Peirce's Formula: $((A \rightarrow B) \rightarrow A) \rightarrow A$

A	B	A→ B	(A→ B)→ A	((A → B) → A) → A
T	T	T	T	T
T	F	F	T	T
F	T	F	F	T
F	F	T	F	T

- This cannot be proved by a pure function,
- To have full *classical* propositional logic, you have to add the excluded-middle axiom: $\forall P. P \vee \neg P$
- People often don't: you can do a lot without axioms.

Back to universal quantification

- We will introduce families of types,
- We will introduce function that produce results in these families,

Indexed types

- Consider families of types B_i ($i \in A$), where each member of a family is annoted with an index i,
  - Assume the existence of a type of types: Type,
  - Assume the existence of a type of indices A,
  - The family of indexed type can be described by a function B whose type is A $\rightarrow$ Type.

Dependent products

- Consider a family of types $B : A \rightarrow Type$,
- Consider a function that takes as input an element x of A and guarantees that it always return an element of type $B(x)$,
- The type system is extended so that this function is well-typed, the notation for its type is forall x:A, B x,
- The name forall is intuitively acceptable: whenever we have an x in A, we know we have a value in $B(x)$.

Why is it called a dependent product?

- A dependent product is a generalization of a cartesian product,
- A cartesian product has the form $A_1 \times A_2$,
- An cartesian product iterated n times is $A_1 \times \ldots \times A_n$
- It can also be written $\Pi_{i \in \{1 \cdots n\}} A_i$,
- We see the use of a family A_i,
- Given an element of $\Pi_{i \in \{1 \cdots n\}} A_i$, we are sure to have an element of A_i for every i
- This is like the dependent product type of the previous frame.

Dependent products in formulas

- Represent formulas in the predicate calculus,
- Assume even : nat → Prop,
- Assume divides : nat → nat → Prop
- Assume there exists a theorem:
  th1: ∀x y, even x → divides x y → even y
- and a theorem:
  th2: ∀x y, divides x (x * y)

Building functions with a dependent product type

- Any function whose output type depends on the input value.
- Simple example: in a context where x has type nat.
  => fun (h: even x) => h: even x $\longrightarrow$ even x
- Check fun (x:nat) => fun (h: even x) => h.

Building functions with a dependent product type

- Any function whose output type depends on the input value.
- Simple example: in a context where x has type nat.
  => fun (h: even x) => h: even x → even x
- Check fun (x:nat) => fun (h: even x) => h.
  fun (x:nat) (h: even x) => h: ∀x: nat, even x → even x

Building functions with a dependent product type (2)

- More elaborate example: recall
  th1: ∀x y, even x ⟶ divides x y ⟶ even y
  th2: ∀x y, divides x (x * y)
- In a context where x:nat, th2 x x: divides x x
- Check fun x => th2 x x.

Building functions with a dependent product type (2)

- More elaborate example: recall
  th1: ∀x y, even x ⟶ divides x y ⟶ even y
  th2: ∀x y, divides x (x * y)
- In a context where x:nat, th2 x x: divides x x
- Check fun x => th2 x x.
  fun (x : nat) => th2 x x: ∀x : nat, divides x (x * x)

Building functions with a dependent product type (2)

- More elaborate example: recall
  th1: ∀x y, even x ⟶ divides x y ⟶ even y
  th2: ∀x y, divides x (x * y)
- In a context where x:nat, th2 x x: divides x x
- Check fun x => th2 x x.
  fun (x : nat) => th2 x x: ∀x : nat, divides x (x * x)
- Check fun x (h: even x) => th1 x x h (th2 x x).

Building functions with a dependent product type (2)

- More elaborate example: recall
  th1: ∀x y, even x → divides x y → even y
  th2: ∀x y, divides x (x * y)
- In a context where x:nat, th2 x x: divides x x
- Check fun x => th2 x x.
  fun (x : nat) => th2 x x: ∀x : nat, divides x (x * x)
- Check fun x (h: even x) => th1 x x h (th2 x x).
  fun (x : nat) (h : even x) => th1 x x h (th2 x x)
  : ∀x:nat, even x → even (x * x)

Defining new constants

Learn some more of the syntax of Coq:

- Definition *name* : *type* := *value*.
- Definition *name* := *value*.
- Definition *name* (*x* : *type*) := *value*.
  This is equivalent to
  Definition *name* := fun *x* : *type* => *value*.

Defining your own indexed type

- We have played with indexed types as if they existed,
- Can we produce some?

Defining your own indexed type

- We have played with indexed types as if they existed,
- Can we produce some?
- You can define a proposition by quantifying over all propositions:
  Definition even (x:nat) : Prop :=
  $\forall$P : nat $\rightarrow$ Prop, ($\forall$y, P(2*y)) $\rightarrow$ P x.
- This was used a lot in the early days of Coq, (replaced by inductive types),
- How do you define divides in the same style?

Defining your own indexed type

- We have played with indexed types as if they existed,
- Can we produce some?
- You can define a proposition by quantifying over all propositions:
  Definition even (x:nat) : Prop :=
  $\forall$P : nat $\rightarrow$ Prop, ($\forall$y, P(2*y)) $\rightarrow$ P x.
- This was used a lot in the early days of Coq, (replaced by inductive types),
- How do you define divides in the same style?
  Definition divides (x y:nat) : Prop :=
  forall P : nat $\rightarrow$ nat $\rightarrow$ Prop,
  (forall z t:nat, P z (z*t)) $\rightarrow$ P x y.

Logic and reasoning in Coq

- Conjunction, disjunction, equality, existential quantification, negation.
- Proof technology: Goals and tactics.

Conjunction

- A function and $:\text{Prop} \rightarrow \text{Prop} \rightarrow \text{Prop}$,
- A notation A /\ B $\equiv$ and A B
- Two basic theorems to construct and consume conjunctions
  - conj : $\forall$A B:Prop, A $\rightarrow$ B $\rightarrow$ A /\ B
  - and_ind : $\forall$A B P:Prop, (A $\rightarrow$ B $\rightarrow$ P) $\rightarrow$ A /\ B $\rightarrow$ P

Disjunction

- A function or :Prop → Prop → Prop,
- A notation A \/ B ≡ or A B
- Three basic theorems to construct and consume disjunctions
  - or_introl : ∀A B:Prop, A → A \/ B
  - or_intror : ∀A B:Prop, B → A \/ B
  - or_ind : ∀A B P:Prop, (A → P) → (B → P) → A /\ B → P

Equality

- A function eq : $\forall$A:Type, A $\rightarrow$ A $\rightarrow$ Prop,
- A notation x $=$ y $\equiv$ eq _ x y
- Two basic theorems to construct and consume equalities
  - refl_equal : $\forall$(A : Type) (x : A), x $=$ x,
  - eq_ind :
    $\forall$(A : Type) (x : A) (P : A $\rightarrow$ Prop), P x $\rightarrow$ x $=$ y $\rightarrow$ P y

Existential quantification

- A function ex : $\forall$(A:Type), (A $\rightarrow$ Prop) $\rightarrow$ Prop,
- A notation exists x : A, P $\equiv$ ex A (fun x : A => P),
- Two basic theorems to construct and consume existential quantifications
  - ex_intro : $\forall$(A : Type) (P : A $\rightarrow$ Prop) (x : A), P x $\rightarrow$ ex P
  - ex_ind : $\forall$(A : Type) (P : A $\rightarrow$ Prop) (Q : Prop),
    ($\forall$x : A, P x $\rightarrow$ Q) $\rightarrow$ ex P $\rightarrow$ Q

Contradiction and negation

- A value False : Prop
- A basic theorem to use contradiction
  - False_ind : $\forall$P : Prop, False $\rightarrow$ P
- A function not = fun A => False,
- A notation ~A $\equiv$ not A

Example of proof: the hard way

```
Definition th1 (x y : nat) (h : even x) (hd : divides x y) : even y :=
  hd (fun z t => even z → even t)
    (fun (z t : nat) (h' : even z)
       (P : nat → Prop) (hp : forall y' : nat, P (2 * y')) =>
     h' (fun z : nat => P (z * t))
       (fun z' : nat =>
        eq_ind (2 * (z' * t)) (fun n : nat => P n)
          (hp (z' * t)) (2 * z' * t) (mult_assoc 2 z' t))) h.
```

- Too hard to build by hand *(I didn't)*.
- A lot of data is redundant and should be computed for us.

Goal directed proofs and tactics

- Propose a statement,
- Apply commands called *tactics* that build the proof term from the outside, with holes inside,
- Fill the holes progresssively,
- Mix with direct constructions of terms.
- In practice: each commands transforms a goal into a simpler goal,
- Goals contain two parts
  1. An enumeration of all the bound variables that are available for use,
  2. A description of the expected type for the current hole.

Example proof by tactics

- Lemma ex1 : ∀P:Prop, P → P.
  =============
  ∀P:Prop, P → P
  Proof: ?1
- intros P H.
  P : Prop
  H : P
  =============
  ∀P:Prop, P → P
  Proof: fun (P : Prop) (H : P) => ?1

Tactics

	$\rightarrow$	$\forall$	/\	\/
hypothesis H	apply H	apply H	case H elim H destruct H	case H elim H destruct H
goal	intros H'	intros x	split	left right

	$\exists$	=	~
hypothesis H	case H elim H destruct H	rewrite $\rightarrow$ H rewrite $\rightarrow$ H	case H
goal	exists *e*	reflexivity	intros H'

- exact H, assumption when the goal is available from the context,
- unfold *name* to unfold definitions,
- assert (H : *formula*) to propose an intermediate step.

Demonstration