Introduction to dependent types in Coq

Yves Bertot

October 24, 2008
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 → nat → nat
Check plus 3 4. 3 + 4 : nat
Building your own function

- Check fun \( x : \text{nat} \Rightarrow 3 + x \).
  \[
  \text{fun } x : \text{nat} = \Rightarrow 3 + x : \text{nat} \rightarrow \text{nat}
  \]

- Use type inference when possible,
  Check fun \( x \Rightarrow 3 + x \).
  \[
  \text{fun } x : \text{nat} \Rightarrow 3 + x : \text{nat} \rightarrow \text{nat}
  \]

- Check fun \((x : \text{nat})(y : \text{bool}) \Rightarrow \text{if } y \text{ then } x \text{ else } 3\).
  \[
  \text{fun } (x : \text{nat})(y : \text{bool}) = \Rightarrow \text{if } y \text{ then } x \text{ else } 3 \\
  : \text{nat} \rightarrow \text{bool} \rightarrow \text{nat}
  \]
Applying functions

- Check \((\text{fun } x \Rightarrow 3 + x) 4\)
  \((\text{fun } (x : \text{nat}) \Rightarrow 3 + x) 4 : \text{nat}\)

- You can also force functions to compute
  \(\text{Eval compute in } (\text{fun } x \Rightarrow 3 + x) 4\)
  \(= 7 : \text{nat}\)
Proofs as functions, functions as proofs

- Using the Modus-Ponens rule: if “A implies B” and “A” both hold, then we can deduce “B”. The Modus-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 $B$.

- Using the forall-elimination rule: if $\forall x : A, P$ and $e$ has type $A$, then we can deduce $P[e/x]$, where $x$ is replaced by $e$. 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 $P[e/x]$, $P$ 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 $A \Rightarrow B$. Yves Bertot - Introduction to dependent types in Coq
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 Modus-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 Modus-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 $P[x/e]$,*
- 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 $P[x/e]$, $P$ 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 Modus-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.
Accept the existence of a type $\text{Prop}$ whose elements are types,

All elements of $\text{Prop}$ 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 Isomorphisms

- 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: \texttt{fun } x: A \Rightarrow x : A \rightarrow A
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: \[
\text{fun } x : A \Rightarrow x : A \rightarrow A
\]

Example 2:

\[
\text{fun } (f : A \rightarrow B \rightarrow C) (x : B) (y : A) \Rightarrow f y x
\]

\[
: (A \rightarrow B \rightarrow C) \rightarrow (B \rightarrow A \rightarrow C)
\]
Various kinds of logics

- Can all usual tautologies be proved with pure functions?

  Peirce's Formula:
  \[(A \rightarrow B) \rightarrow A \rightarrow A\]

  This cannot be proved by a pure function, to have full classical propositional logic, you have to add the excluded-middle axiom:
  \[\forall P. P \lor \neg P\]

  People often don't: you can do a lot without axioms.
Various kinds of logics

- Can all usual tautologies be proved with pure functions?
  - No.
  - Peirce’s Formula: 

\[
((A \rightarrow B) \rightarrow A) \rightarrow A
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A → B</th>
<th>(A → B) → A</th>
<th>((A → B) → A) → A</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
Various kinds of logics

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A→B</th>
<th>(A→B)→A</th>
<th>((A → B) → A) → A</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

- This cannot be proved by a pure function,
- To have full classical propositional logic, you have to add the excluded-middle axiom: \(\forall P. P \lor \neg P\)
- 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,
Consider families of types $B_i$ ($i \in A$), where each member of a family is annotated with an index $i$,

- Assume the existence of a type of types: $\text{Type}$,
- Assume the existence of a type of indices $A$,
- The family of indexed type can be described by a function $B$ whose type is $A \rightarrow \text{Type}$. 
Consider a family of types $B : A \rightarrow \text{Type}$,

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 $B(x)$. 

Yves Bertot
Introduction to dependent types in Coq
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 $\prod_{i \in \{1\ldots n\}} A_i$,

We see the use of a family $A_i$,

Given an element of $\prod_{i \in \{1\ldots n\}} A_i$, we are sure to have an element of $A_i$ for every $i$

This is like the dependent product type of the previous frame.
Represent formulas in the predicate calculus,
Assume `even : nat → Prop`,
Assume `divides : nat → nat → Prop`
Assume there exists a theorem:
\[ \text{th1: } \forall x \ y, \ even \ x \ → \ divides \ x \ y \ → \ even \ y \]
and a theorem:
\[ \text{th2: } \forall x \ y, \ divides \ x \ (x \ * \ y) \]
Any function whose output type depends on the input value.

Simple example: in a context where \( x \) has type \( \text{nat} \).

\[
\Rightarrow \text{fun (} h : \text{even } x \text{)} \Rightarrow h : \text{even } x \rightarrow \text{even } x
\]

Check \( \text{fun (} x : \text{nat}\text{)} \Rightarrow \text{fun (} h : \text{even } x \text{)} \Rightarrow h \).
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 $\text{nat}$.
  $$\Rightarrow \text{fun } (h: \text{even } x) \Rightarrow h: \text{even } x \rightarrow \text{even } x$$
- Check
  $$\text{fun } (x:\text{nat}) \Rightarrow \text{fun } (h: \text{even } x) \Rightarrow h.$$
  $$\text{fun } (x:\text{nat}) (h: \text{even } x) \Rightarrow h: \forall x: \text{nat}, \text{even } x \rightarrow \text{even } x$$
More elaborate example: recall

\[ \forall x \ y, \text{even } x \rightarrow \text{divides } x \ y \rightarrow \text{even } y \]

\[ \forall x \ y, \text{divides } x \ (x \ * \ y) \]

In a context where \( x: \text{nat} \), \( \text{th2 } x \ x \): divides \( x \ x \)

Check fun \( x \Rightarrow \text{th2 } x \ x \).

\[ \text{fun } (x : \text{nat}) \Rightarrow \text{th2 } x \ x : \forall x : \text{nat}, \text{even } x \rightarrow \text{even } (x \ * \ x) \]
More elaborate example: recall

\textbf{th1}: \ \forall x \ y, \ \text{even} \ x \ \rightarrow \ \text{divides} \ x \ y \ \rightarrow \ \text{even} \ y

\textbf{th2}: \ \forall x \ y, \ \text{divides} \ x \ (x \ \ast \ y)

In a context where \( x : \text{nat} \), \( \text{th2} \ x \ x \) : \( \text{divides} \ x \ x \)

Check \( \text{fun} \ x \Rightarrow \text{th2} \ x \ x \).

\( \text{fun} \ (x : \text{nat}) \Rightarrow \text{th2} \ x \ x \) : \( \forall x : \text{nat}, \ \text{divides} \ x \ (x \ \ast \ x) \)
More elaborate example: recall
\[ \text{th1: } \forall x \ y, \text{ even } x \ \rightarrow \ \text{divides } x \ y \ \rightarrow \ \text{even } y \]
\[ \text{th2: } \forall x \ y, \text{ divides } x \ (x \ast y) \]
In a context where \( x : \text{nat} \), \( \text{th2 } x \ x \) : divides \( x \ x \)
Check fun \( x \Rightarrow \text{th2 } x \ x \).
fun \( (x : \text{nat}) \Rightarrow \text{th2 } x \ x \) : \( \forall x : \text{nat}, \text{divides } x \ (x \ast x) \)
Check fun \( x \ (h : \text{even } x) \Rightarrow \text{th1 } x \ x \ h \ \text{(th2 } x \ x) \).
More elaborate example: recall

\begin{align*}
\text{th1: } & \forall x \ y, \text{ even } x \to \text{ divides } x \ y \to \text{ even } y \\
\text{th2: } & \forall x \ y, \text{ divides } x \ (x \ * \ y)
\end{align*}

In a context where \( x: \text{nat} \), \( \text{th2} \ x \ x \) : divides \( x \ x \)

Check fun \( x \to \text{th2} \ x \ x \).

\begin{align*}
\text{fun (x : nat) } & \to \text{th2} \ x \ x \ : \ \forall x : \text{nat}, \text{ divides } x \ (x \ * \ x)
\end{align*}

Check fun \( x \ (h: \text{even } x) \to \text{th1} \ x \ x \ h \ (\text{th2} \ x \ x) \).

\begin{align*}
\text{fun (x : nat) (h : even x) } & \to \text{th1} \ x \ x \ h \ (\text{th2} \ x \ x) \\
& : \forall x: \text{nat}, \text{ even } x \to \text{ even } (x \ * \ x)
\end{align*}
Defining new constants

Learn some more of the syntax of Coq:

- Definition \textit{name} : \textit{type} := \textit{value}.
- Definition \textit{name} := \textit{value}.
- Definition \textit{name} (\textit{x} : \textit{type}) := \textit{value}.

This is equivalent to

Definition \textit{name} := \text{fun} \ \textit{x} : \textit{type} => \textit{value}.
We have played with indexed types as if they existed,
Can we produce some?
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 :=
    \( \forall P : \text{nat} \rightarrow \text{Prop}, (\forall y, P(2*y)) \rightarrow P x. \)
This was used a lot in the early days of Coq, (replaced by inductive types),
How do you define divides in the same style?
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 :=
    \(\forall P : \text{naturals} \to \text{Prop}, (\forall y, P(2*y)) \to P \ x.\)

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 :=
    \(\forall P : \text{naturals} \to \text{naturals} \to \text{Prop},\)
    \((\forall z, t : \text{naturals}, P z (z * t)) \to P x y.\)
Logic and reasoning in Coq

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

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

- A function \( \text{or} : \text{Prop} \to \text{Prop} \to \text{Prop} \),
- A notation \( A \lor B \equiv \text{or} A B \)
- Three basic theorems to construct and consume disjunctions
  - \( \text{or\_intr\_l} : \forall A B : \text{Prop}, A \to A \lor B \)
  - \( \text{or\_intr\_r} : \forall A B : \text{Prop}, B \to A \lor B \)
  - \( \text{or\_ind} : \forall A B P : \text{Prop}, (A \to P) \to (B \to P) \to A \land B \to P \)
A function \( \text{eq} : \forall A : \text{Type}, A \rightarrow A \rightarrow \text{Prop} \),

A notation \( x = y \equiv \text{eq} \ x \ y \),

Two basic theorems to construct and consume equalities
  \( \text{refl\_equal} : \forall (A : \text{Type}) (x : A), x = x \),
  \( \text{eq\_ind} : \forall (A : \text{Type}) (x : A) (P : A \rightarrow \text{Prop}), P \ x \rightarrow x = y \rightarrow P \ y \)
Existential quantification

- A function \( \text{ex} : \forall (A:\text{Type}), (A \rightarrow \text{Prop}) \rightarrow \text{Prop}, \)
- A notation \( \text{exists } x : A, P \equiv \text{ex } A (\text{fun } x : A \Rightarrow P), \)
- Two basic theorems to construct and consume existential quantifications
  - \( \text{ex_intro} : \forall (A : \text{Type}) (P : A \rightarrow \text{Prop}) (x : A), P x \rightarrow \text{ex } P \)
  - \( \text{ex_ind} : \forall (A : \text{Type}) (P : A \rightarrow \text{Prop}) (Q : \text{Prop}), \)
  \( (\forall x : A, P x \rightarrow Q) \rightarrow \text{ex } P \rightarrow Q \)
Contradiction and negation

- A value \( \text{False} \) : Prop
- A basic theorem to use contradiction
  - \( \text{False}_{\text{ind}} : \forall P : \text{Prop}, \text{False} \rightarrow P \)
- A function \( \text{not} = \text{fun } A \Rightarrow \text{False}, \)
- A notation \( \sim A \equiv \text{not } A \)
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 progressively,
- 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 : \( \forall P : Prop, P \rightarrow P \).

\[ \forall P : Prop, P \rightarrow P \]

Proof: ?1

intros P H.

P : Prop
H : P

\[ \forall P : Prop, P \rightarrow P \]

Proof: fun (P : Prop) (H : P) => ?1
Tactics

<table>
<thead>
<tr>
<th></th>
<th>→</th>
<th>∀</th>
<th>/\</th>
<th>∨</th>
</tr>
</thead>
<tbody>
<tr>
<td>hypothesis H</td>
<td>apply H</td>
<td>apply H</td>
<td>case H</td>
<td>case H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>elim H</td>
<td>elim H</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>destruct H</td>
<td>destruct H</td>
</tr>
<tr>
<td>goal</td>
<td>intros H'</td>
<td>intros x</td>
<td>split</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>right</td>
</tr>
<tr>
<td></td>
<td>∃</td>
<td>=</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>hypothesis H</td>
<td>case H</td>
<td>rewrite → H</td>
<td>case H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>elim H</td>
<td>rewrite → H</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>destruct H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>goal</td>
<td>exists e</td>
<td>reflexivity</td>
<td>intros H'</td>
<td></td>
</tr>
</tbody>
</table>

- 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