Dependent types

Yves Bertot

August 2010
- Dependent types: you saw them already
  - Universally quantified theorems
  - Polymorphic functions
- Families of types, functions to families of types
- Usage in safe programming
  - More information in types
  - “Certified” values
  - Controlling termination with types: well-founded recursion
Polymorphic data types and functions

\[ \text{list} : \text{Type} \rightarrow \text{Type} \]

- \( \text{list bool} \) and \( \text{list nat} \) are two different types
- \( \text{nil} \) is a function, it returns values in different types
- Implicit arguments hide the extra type
- Notation \( \text{nil} : \text{forall A : Type, list A} \)
Polymorphic data types and functions

list : Type -> Type

- list bool and list nat are two different types
- nil is a function, it returns values in different types
- Implicit arguments hide the extra type
- Notation nil : forall A : Type, list A

Search list.
nil: forall A : Type, list A
cons: forall A : Type, A -> list A -> list A
tail: forall A : Type, list A -> list A
app: forall A : Type, list A -> list A -> list A
map: forall A B : Type, (A -> B) -> list A -> list B
rev: forall A : Type, list A -> list A
Universally quantified theorems

Propositions are types: Curry-Howard isomorphism

- The elements of propositions-types are the proof
- `even : nat -> Prop` defines a family of types
- `even 0` contains an element, `even 1` does not
- `A -> B` is a type. its elements map proofs of `A` to proofs of `B`
  - elements of `A -> B` prove `A` imply `B` by showing how
- `forall x:A, P x` is also a function type, called a product type
Two levels: types and proofs

Declaring \( P : \text{nat} \rightarrow \text{Prop} \) means \( P \ x \) is a proposition for every \( x : \text{nat} \)

- Not that \( P \ x \) always holds

Declaring \( t : \forall x, P \ x \) means that \( P \) always holds

- \( t \) is not a predicate or a type
Example building proofs

even0 : even 0
even2 : forall x : nat, even x -> even (S (S x))
even2 0 : even 0 -> even 2
even2 0 even0 : even 2
even2 2 : even 2 -> even 4
Example building proofs

even0 : even 0
even2 : forall x : nat, even x -> even (S (S x))
even2 0 : even 0 -> even 2
even2 0 even0 : even 2
even2 2 : even 2 -> even 4
even2 2 (even2 0 even0) : even 4
even2 4 : even 4 -> even 6
even2 4 (even2 2 (even2 0 even0)) : even 6
Building elements in a product type

A product type \( \forall x : A, P \ x \) is a function type

- Construct elements in this type using \( \text{fun} \ .. \Rightarrow .. \)
- The argument has to be in type \( A \) (similar to \( A \rightarrow B \))
- The result has to be in \( P \ x \)
Example building an element in a product type

Check

\[
(\text{fun } (x : \text{nat}) \Rightarrow
  (\text{fun } (h : \text{even } x) \Rightarrow \text{even2 } (S (S x)) (\text{even2 } x h)))).
\]

\[
\text{forall } x : \text{nat}, \text{even } x \Rightarrow \text{even } (S (S (S (S x)))).
\]

You usually don’t need to do this by hand

- Use Goal-directed proof instead
- The tactic intros always builds a fun ... => ...
- The tactic apply constructs an application
Defining new type families

Use inductive types

- Inductive predicates are type families
- Constructors are dependently typed functions
- Used a lot in Coq: and, or, exists, equality, le, etc.
  - Descriptions of programming languages

Use recursive functions

- Proofs are other functions, not necessarily recursive,
- For example: In predicate on lists
- Seldom used in practice
Examples defining type families

Inductive even : nat -> Prop :=
  even0 : even 0
| even2 : forall x : nat, even x -> even (S (S x)).
Examples defining type families

Inductive even : nat -> Prop :=
  even0 : even 0
| even2 : forall x : nat, even x -> even (S (S x)).

Fixpoint ev' (n:nat) : Prop :=
  match n with
  | 0 => True | 1 => False | S (S p)) => ev' p
end.

Definition ev2 : forall x, ev' x -> ev' (S (S x)) :=
  fun x h => h.
Pattern matching constructs give different values for different inputs

- Each value can have a different type
- You have to say explicitly when there is dependency
- This can be mixed with recursion
- Good news: the tactics `case`, `case_eq`, and `destruct` do it for you
Example dependent pattern-matching

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

Print eq_rect.
eq_rect =
fun A (x : A) (P : A -> Type)
  (f : P x) (y : A) (e : x = y) =>
  (match e in (_ = y0) return (P y0) with
   | refl_equal => f
  end : P y)
Dependent types

Dependent types and pattern-matching

Goal directed proof for dependent pattern-matching

Lemma eq_rect :
  \forall (A:Type) (x:A) (P:A \to Type),
  P x \to x = y \to P y.

intros A x P hp heq.
  hp : P x
  heq : x = y
  \-------------------
  P y

case heq.
  ...
  \-------------------
  P x

exact hp.
Qed.
Example recursive dependent function

Lemma th : forall p, (S (S (2 * p))) = (2 * S p).
Proof. intros; ring. Qed.

Fixpoint double_even (n:nat) : even (2 * n) :=
  match n as x return even (2 * x) with
    0 => even0
  | S p => @eq_rect nat (S (S (2 * p)))
    (fun x => even x)
    (even2 (2 * p) (double_even p))
    (2 * (S p)) (th p)
end.
Goal directed proof for recursive dependent function

Lemma double_even : forall n, even (2 * n).
Proof.
induction n.

=======================
even 0
exact even0.
IHn : even (2 * n)
=======================
evenc (2 * (S n))
rewrite <- th; apply even2; exact IHn.
Qed.
Dependent types for safe programming

Use dependent types to assume properties

- A function $f : \forall x, P(x) \rightarrow R$ assumes $x$ satisfies $P$

Use dependent types to guarantee properties

- A type $\{x : A \mid Q(x)\}$ describes the values that satisfy $Q$
- A function $g : \forall x : A, P(x) \rightarrow \{y : B \mid R(x, y)\}$ guarantees some relation between input and output
- A function of type $(\forall y : A, P(y) \rightarrow B) \rightarrow C$ guarantees that it calls its argument only on values that satisfies $P$
- All guarantees verified using types, at compile time!
Dependent types for termination

A notion of well-founded relation

- No infinite decreasing chains
- $x_0 \ldots x_n \ldots$ with $R \ x_{i+1} \ x_i$ stops eventually

**Fix**: $\forall A \ R \ (\text{th} : \text{well-founded } R) \ (P : A \to \text{Type}) \ (\forall x, (\forall y, R \ y \ x \to P \ y) \to P \ x) \to \forall x, P \ x$

- Blue part is the function used for recursive calls
- The programmer has to guarantee that recursive calls are only on smaller arguments
Conclusion on well-founded recursion

The function Fix is still difficult to use directly
  ▶ Better when used in proofs, but still obfuscated
Other tools support general recursion
  ▶ Program Fixpoint
  ▶ Function
General conclusion

- Dependent types are everywhere in Coq
- Describe strong disciplines of programming
- Most verifications done at compile time
- Extraction mechanisms remove verifications