Interactions between inductive and dependent types

Yves Bertot

August 2009

Mixing inductive types with dependent types gives us more expressive power, stronger invariants for datatypes, new ways to define predicates. In this lesson, we will concentrate on :

- Dependent pattern matching and recursion
- Inductive types with dependently typed constructors
- Inductive families of types used as predicates
- Specific proof techniques

Dependent pattern-matching

Attach to pattern-matching construct an expression that describes the returned type

```
match e as x return t with
   p1 => e1
|  p2 => e2
end
```

The pattern-matching construct is well formed if each e_1 has type $\mathtt{t}[x \backslash p_i]$

Example on dependent pattern-matching

Define a function on natural numbers that maps 0 to another natural number and all other natural numbers to a boolean values
First define the function that describes the returned type

```
Definition rt (x : nat) :=
  match x with 0 => nat | S p => bool end.

Definition f1 (x : nat) :=
  match x as e return rt e with
    0 => 1 | S p => true
  end.

Definition f2 (x : nat) :=
  match x return rt x with 0 => 1 | _ => true end.
```

Dependent pattern-matching with recursion

```
match e as x return t with
   p1 => e1
| p2 => e2
end
```

If the type of e has recursion, then recursive calls may occur in e_1 and e_2, with relevant types...

Recursive dependent pattern-matching on nat

```
Fixpoint f (x : nat) : A x :=
  match x return A x with
    0 => V
  | S p => E p (f p)
  end.
```

The expression V must have type `A 0`
the expression E must have type
`forall p :nat, A p -> A (S p)`

Recursive dependent pattern-matching on nat (2)

```
Fixpoint nat_rect (P:nat -> Type)
  (V : P 0)
  (E : forall n:nat, P n -> P (S n))
  (n : nat) : P n :=
  match n with
    0 => V
  | S p => E p (nat_rect P V E p)
  end.
```

```
nat_rect :
  ∀ P, P 0 -> (∀ n, P n -> P (S n)) -> ∀ n, P n
```

Done automatically when receiving the type definition
Usual mathematical induction principle on natural numbers

Dependently typed constructors

With dependent types, we can set constraints on components of structures

```
Inductive binary_word1 (n:nat) : Type :=
  bwc (l : list bool) (_ : length l = n).
```

The type of the second component depends on the first one.
Alternative approach to fixed length vectors :

```
Inductive binary_word : nat -> Type :=
  bw_nil : binary_word 0
| bw_cons :
  forall n, bool -> binary_word n -> binary_word (S n).
```

Possibly empty inductive type families

```
Inductive even_line : nat -> Type :=
  el0 : even_line 0
| el2 : forall n, even_line n -> even_line (S (S n)).
```

The types `even_line 0`, `even_line 2`, etc, are inhabited
It is possible to show that `even_line 1` is not.

Recursion on inductive types families

Induction principles use predicates that cover all members of the type family
Quantification over the index, then over the members
Each constructor gives rise to an hypotheses (as for nat)

Induction on even_line

```
forall A : forall n, even_line n -> Type,
  A 0 el0 ->
  (forall n (t : even_line n), A n t ->
            A (S (S n)) (el2 n t)) ->
  forall n (t : even_line n), A n t
```

How to write the recursion function

```
Fixpoint elr (A : forall n, even_line n -> Type)
    (V : A 0 el0)
    (E : forall n, even_line n -> even_line (S (S n)))
    (n:nat) (e:even_line n) : A n e :=
  match e as e0 in even_line n0 return A n0 e0 with
    el0 => V
  | el2 p t => E p t (elr A V E p t)
  end.
```

Using inductive types as predicates

As a datatype `even_line` is not interesting
For a given *n*, `even_line` *n* contains zero or one element
The information whether it contains an element is interesting, not the element
It is worth having a new recursion principle that concentrates on *n*

```
forall A : nat -> Prop,
  A 0 ->
  (forall n, even_line n -> A n -> A (S (S n))) ->
  forall n, even_line n -> A n
```

Derivable directly from the previous one
Done automatically when the type is in `Prop` instead of `Type`

A tour of inductive predicates

Most logical connectives actually are inductive predicates
This explains why `case`, `destruct`, `elim` are useful on connectives
The order $\leq$ on natural numbers is also defined inductively
Programming language semantics are also easily described using inductive predicates

- Reminiscent of logic programming
- No problems with function termination

Induction on inductive predicates

Assume the following inductive predicate :

```
Inductive even : nat -> Prop :=
  ev0 : even 0 | ev2 : forall n, even n -> even (S (S n)).
```

Show the obvious property of even numbers :

```
Lemm ex3 : forall n, even n -> exists p, n = 2 * p.
```

A proof by induction will follow the structure of constructors

Induction on an inductive predicate

```
intros n H; induction H.
2 subgoals
================
exists p, nat, 0 = 2 * p
exists 0; reflexivity.
H : even n
IHeven : exists p, n = 2 * p
================
exists p, S (S n) = 2 * p
destruct IHeven as [p q]; subst; exists (S p); ring.
Proof completed.
```

Inversion

Proofs by induction on predicate don't work well if arguments are not variables
A stronger tactic can be used in this case : `inversion`
For a given argument, `inversion` detects which constructors could be used to prove the instance and collects facts from the premises
This tactic makes it possible to move from conclusion to premises, hence the name.

Examples using inversion

```
Lemma ex4 : forall n, even (S (S n)) -> even n.
intros n H; inversion H.
H0 : even n
==============
even n
assumption.
```

The tactic determines that only the constructor ev2 could be used to prove `even (S (S n))`
It adds the corresponding instance of the premise to ev2 to the context

Second example using inversion

```
Lemma ex5 : even 1 -> False.
intros H; inversion H.
proof completed.
```