Dependently-typed Programming in Coq

MPRI - Proof Assistants

M. Sozeau

Outline

1 - Subset types and dependent inductive types.

2 - Programming with vectors and dependent pattern-matching

3 - Examples & exercises

Dependently-typed programming

"Correctness by construction". Example: bounds checking.

Partial function:

```
nth : ∀ (l : list A) (n : nat), A.
```

The total version in Coq:

```
nth : ∀ (l : list A) (n : nat) (default : A), A.
```

+ lemmas: forall l, n < length l,

```
nth l n default = nth (map id l) n default...
```

The total function, with types depending on values:

```
nth : ∀ (l : list A) (n : nat | n < length l), A.
```

Subset Types

```
Inductive sig (A : Set) (P : A -> Prop) :=
  exist : forall a : A, P a -> sig A P.

 Notation "{x : A | P}" := sig A (fun x : A => P).

 Definition proj1_sig {A} {P} (s : {x : A | P x}) : A.
 Definition proj2_sig {A} {P} (s : {x : A | P x}) :
    P (proj1_sig A).
```

=> Associate a property to an object, an invariant.

Strong Specifications

```
euclidian_division :
 forall (x : nat) (y : { y : nat | 0 < y }),
         { (q, r) | x = q * y + r }
```

Program

A tool to program with subset types:

Coq's type system + the rules:

```
⊢p t : { x : A | P }
-----------------
⊢p t : A

⊢p t : A   x : A ⊢p P : Prop
-----------------              <--- No proof of P t
here!
⊢p t : {x : A | P}
```

Program

Translation: ⊢p t : T → ⊢ t' : T'

Inserts projections and coercions for uses of the subtyping rules:

```
⊢ t : { x : A | P }
-----------------
⊢ proj1_sig t : A

⊢ t : A   x : A ⊢ P : Prop    |- ?p : P t
-------------------------------------------
⊢ exist t ?p : {x : A | P}
```

Program

Translation: $\vdash_p t : T \rightarrow \vdash t' : T'$

t', T' might contain holes : proof obligations.

After solving the holes we get a full coq term.

=> DEMO

Inductive types and derivations

```
Inductive types (Γ : ctx) : term → type → Prop :=
| ty_var i : types Γ (Var i) (nth Γ i)
| ty_nat (n : nat) : types Γ (Constant n) nat_ty
...
```

=> Inhabitants of [types Γ t ty] are only valid derivations.

Dependent Inductive types

Indexed datatypes:

```
 Inductive vect (A : Set) : nat → Set :=
 | vnil : vect A 0
 | vcons (a : A) {n} (v : vect A n) : vect A (S n).

Definition x : vect bool 3 :=
   vcons true (vcons true (vcons false vnil)).
```

=> Inhabitants of [vect bool n] are lists of length n.

Strong Specifications again

```
concat_vect {n m} : vect A n -> vect A m -> vect A (n +
m)

diagonal {n} : vect (vect A n) n -> vect A n.

head {n} : vect A (S n) -> A.  <- Total
```

=> All size/indexing invariants are part of the type.

Structure versus Property

```
Inductive vect (A : Set) : nat → Set :=
| vnil : vect A 0
| vcons (a : A) {n : nat} (v : vect A n) : vect A (S n).
```

Inductives with indices can sometimes be seen as subsets:

```
Definition listn (n : nat) :=
  {l : list A | length l = n}.
```

See: Inductive Ornaments (McBride, Ghani...)

Example dependent inductives

Finite sets:

```
Inductive fin : nat → Set :=
| fin0 n : fin (S n)
| finS n (f : fin n) : fin (S n).
```

fin n has n elements.

Well-typed expressions:

```
Inductive expr (Γ : ctx) : ty → Set :=
| var (f : fin (length Γ)) : expr (nth Γ f)
| app {tau tau'} (f : expr Γ (tau -> tau'))
  (u : expr tau) : expr tau'.
```

Example dependent inductives

Matrices, any bounded datastructure:

```
square_matrix n = vect (vect n) n.
```

Regular expressions indexed by their semantics:

```
Inductive regexp : (string → Prop) → Set :=
 | empty : regexp (fun s => s = "")
 | or (x y) (a : regexp x) (b : regexp y) :
     regexp (fun s => x s \/ y s).

Lemma regexp_interp x (e : regexp x) (s : string),
  matches e s -> x s.
```

Type-preserving interpreters

A small "universe" of types:

```
Inductive type := nat_ty : type | unit_ty : type.

Definition interp_type (ty : type) : Set :=
 match ty with
  | nat_ty => nat
  | unit_ty => unit
  end.

Definition interp (Γ : ctx) (ty : type) (x : expr Γ ty)
:
                        interp_type ty.
```

=> No ill-typed terms, by construction.

Computing with Dependent Inductives: Elimination

```
match E in (T x1 ... xn) as y return U with
  | C z1 ...zm => B
  | ...
  end
```

E discriminee of type T u1 ... un

y is a name for E (which might not be a variable)

U is the return type **depending** on [x1 ... xn] and [e].

Each pattern [C z1 ... zm] has type [I t1 .. tm].

Each branch [B] should be of type

U[C z1 .. zm/e,t1/x1..tm/xm]

The whole pattern-matching will have type

U[T u1..um/e, u1/x1..um/xm].

Dependent Elimination

In a sense the pattern-matching should be "general enough" to apply to any object of the family, but is specialized to
a particular instance right away. For example:

```
Inductive fin : nat -> Set :=
| fin0 n : fin (S n)
| finS n (f : fin n) : fin (S n).

Definition nofin0 (f : fin 0) : False :=
  match f in fin n return
     (match n with 0 => False | S n => True)
  with
     | fin0 _ => I
     | finS _ _ => I
  end
```

History

Many flavors of dependent inductive types and pattern-matching.

- DML is ML + size indexes on datatypes
- Haskell, OCaml have GADTs: indices can be types only.
- The Agda, Epigram programming language have approximately the same power as Coq.

References

On dependent pattern-matching:

- Inductive types in the system Coq, Paulin, TLCA'93.
- Eliminating dependent pattern-matching, Goguen, McBride and McKinna, LNCS, 2006.

Program: in Coq's reference manual.

- Programming Finger Trees in Coq, Sozeau, ICFP'07

Coq's dependently typed programming features:

http://adam.chlipala.net/cpdt/html/toc.html

Exercises

http://www.pps.univ-paris-diderot.fr/
~sozeau/teaching/MPRI-dep-exercises.v

A little announcement:

PhD funding for a thesis on Coq, Geometrical proofs and education in Strasbourg:

http://dpt-info.u-strasbg.fr/~narboux/stages.html