Dependently Typed Programming in Coq

Matthieu Sozeau

MPRI 2-7-2 – Proof assistants

February 24th 2014

Outline

1. Dependently-Typed Programming in Coq
   - Subset Types
   - Inductive Families
   - Examples & Exercises

2. A bit about models
   - The groupoid model
   - Homotopy Type Theory

Correctness by construction

Example: bounds checking.

- Partial function

```
Definition nth_impossible : ∀ A, nat → list A → A.
Abort.
```

- A total version in Coq

```
Check nth : ∀ A, nat → list A → ∀ default : A, A.
```

+ lemmas: $\forall\ l,\ n < \mathrm{length}\ l,\ \mathrm{nth}\ l\ n\ default = \mathrm{nth}\ (map\ id\ l)\ n\ default$

- The total function, with types depending on values:

```
Example nth : ∀ A (l : list A), {n : nat | n < length l} → A.
  Exercise.
Defined.
```

Subset types

Subset types are Σ-types:
$\{ \mathrm{x} : A \mid P \} \leftrightarrow \mathit{sigma}\ A\ (\mathtt{fun}\ \mathrm{x} : A \Rightarrow P)$

- Constructor:

```
Check (exist : ∀ A (P : A → Prop), ∀ x : A, P x → { x : A | P x }).
```

- Projections:

```
Check (proj1_sig : ∀ A P, {x : A | P x} → A).
Check (proj2_sig : ∀ A P (p : {x : A | P x}), P (proj1_sig p)).
```

- Binding notation:

```
Check (proj2_sig : ∀ A P (x : A | P x), P (proj1_sig x)).
```

Strong specifications

```
Program Definition euclid_type :=
  ∀ (x : nat) (y : nat | 0 < y),
    { (q, r) : nat × nat | x = q × y + r }.
```

Program

- A tool to program with subset types.
- Coq's type system + the rules:

```
⊢ t : { x : A | P }
-------------------                (Sub-Proj)
⊢ t : A

⊢ t : A    ⊢ P : Prop
--------------------                (Sub-Proof)
⊢ t : { x : A | P }
```

Note `Sub-Proof` does *not* require a proof term.

- An elaboration to Coq terms with holes for the missing proof terms.

Program II

- An elaboration to Coq terms with holes for the missing proof terms.

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

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

- This inserts projections and coercions everywhere needed.
- The translation is correct for a proof-irrelevant Coq only (Sozeau'08, PhD)

Program Example

```
Require Import Arith Omega.

Program Fixpoint euclid (x : nat)
        (y : nat | 0 < y) (* Binder for subsets *)
        { wf lt x } : (* Wellfounded definition *)
  { (q, r) : nat × nat | x = q × y + r } (* Rich type *) :=
  _ (*  Hint: use lt_dec *) .

Next Obligation.
  Exercise.
Defined.
```

Solution

```
Program Fixpoint euclid (x : nat)
        (y : nat | 0 < y) (* Binder for subsets *)
        { wf lt x } : (* Wellfounded definition *)
  { (q, r) : nat × nat | x = q × y + r } (* Rich type *) :=
    if lt_dec x y then (0, x)
    else
      let 'pair q r := euclid (x - y) y in
        (S q, r).

  Next Obligation. omega. Qed.
  Next Obligation.
    destruct euclid. simpl in *. subst x0.
    omega.
  Defined.

Extraction Inline proj1_sig projT2 projT1.
Recursive Extraction euclid.
```

Inductive Families

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

- Indexed
- Recursive
- Terms *and* types carry more information

```
Example x : vect bool 3 :=
 vcons _ true 2 (vcons _ true 1 (vcons _ false 0 (vnil _))).
```

Return clauses

```
Arguments vnil {A}.
Arguments vcons {A} a {n} v.

Fixpoint vect_map {A B : Set} {n} (f : A → B) (v : vect A n)
: vect B n :=
  match v with
    | vnil ⇒ vnil
    | vcons a n v' ⇒ @vcons B (f a) n (vect_map f v')
  end.
```

Return clauses

```
Arguments vnil {A}.
Arguments vcons {A} a {n} v.

Fixpoint vect_map {A B : Set} {n} (f : A → B) (v : vect A n)
: vect B n :=
  match v with
    | vnil ⇒ vnil
    | vcons a n v' ⇒ @vcons B (f a) n (vect_map f v')
  end.

Fixpoint vect_map' {A B : Set} {n} (f : A → B) (v : vect A
n) : vect B n :=
  match v in vect _ n return vect B n with
    | vnil ⇒ vnil
    | vcons a n v' ⇒ vcons (f a) (vect_map' f v')
  end.
```

Refining the input type

```
Definition vect_hd {A : Set} {n} (v : vect A (S n)) : A :=
  match v in vect _ n
        return match n with 0 ⇒ unit | S n ⇒ A end
  with
    | vnil ⇒ tt
    | vcons a n v' ⇒ a
  end.

Definition vect_tl {A : Set} {n} (v : vect A (S n)) : vect A n.
  Exercise.
Defined.
```

Strong specifications again

```
Definition concat_vect {A : Set} {m n}
                (v : vect A m) (w : vect A n) : vect A (m + n).
Proof. Exercise. Defined.
```

- Non-linear case: harder to program, needs *explicit* equality manipulations in Coq.

```
Example diagonal {A : Set} {m} :
  vect (vect A m) m → vect A m.
```

- Certified Programming with Dependent Types (Chlipala, MIT Press) goes into the many tricks needed to program with these types in Coq.

It's all the same

- Inductive families vs subset types.
- Structure vs property.

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

Record Iso (A B : Type) :=
  { f : A → B; g : B → A;
    fog : ∀ x, f (g x) = x;
    gof : ∀ x, g (f x) = x }.

Program Definition vect_ilist {A : Set} (n : nat) :
  Iso (vect A n) (@ilist A n) := { f := _ ; g := _ }.

Next Obligation. Exercise. Qed.
```

The relationship can be made explicit, categorically or using a universe of datatypes, see the literature on Ornaments (McBride, Ghani, Dagand, ...)

More examples

- Matrices, any bounded datastructure

```
Definition square_matrix {A} n := vect (vect A n) n.
```

- Regular expressions indexed by their semantics

```
Require Import String.

Inductive regexp : (string → Prop) → Type :=
| empty : regexp (fun s ⇒ s = ""%string)
| or x y (a : regexp x) (b : regexp y) :
    regexp (fun s ⇒ x s ∨ y s).

Definition matches x (e : regexp x) (s : string) : bool.

Lemma regexp_interp x (e : regexp x) (s : string) :
  matches x e s = true → x s.
Proof. .... Defined.
```

More examples

- Finite sets (fin n has n elements)

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

Definition lookup {A : Set} {n} (v : vect A n) (f : fin n) : A.
  Exercise.
Defined.
```

Well-typed syntax

```
Inductive ty := nat_ty | arrow (t u : ty).
Definition ctx := vect ty.

Inductive expr {n} (Γ : ctx n) : ty → Set :=
| var (f : fin n) : expr Γ (lookup Γ f)

| app {tau tau'}
        (f : expr Γ (arrow tau tau')) (u : expr Γ tau)
  : expr Γ tau'

| lam {tau tau'}
        (t : @expr (S n) (vcons tau Γ) tau')
  : expr Γ (arrow tau tau').
```

And definitional interpreters

```
Fixpoint interp_type (ty : ty) : Set :=
 match ty with
   | nat_ty ⇒ nat
   | arrow tau tau' ⇒ interp_type tau → interp_type tau'
 end.

Definition interp n (Γ : ctx n) (ty : ty) (x : expr Γ ty)
  : interp_type ty.
  Exercise.
Defined.
```

- No ill-typed terms, by construction.

Some history

Many flavors of inductive families and dependent pattern-matching.

- DML (Xi): ML + integer indexed types (presburger arithmetic)
- Agda (Norell), Epigram (McBride): have the K rule that allows working with non-linear cases and a higher level construction (the Equations plugin (Sozeau) does the same for Coq).
- Haskell, OCaml GADTs: indices can be types only, not arbitrary terms.
- CoqMT (Strub): Coq Todulo Theories, extends conversion to arbitrary decidable theories, including presburger arithmetic. No equality manipulations necessary!

And many others: ATS (Xi), Beluga (Pientka), Ωmega (Sheard), Trellys (Weirich), ...

Some bibliography

On dependent pattern-matching and inductive families:

- Inductive types in the system Coq, Paulin, TLCA'93.
- Eliminating dependent pattern-matching, Goguen, McBride and McKinna, LNCS, 2006. McBride's papers include a large number of examples.
- Program: in Coq's reference manual and Programming Finger Trees in Coq, Sozeau, ICFP'07

Outline

1. Dependently-Typed Programming in Coq
   - Subset Types
   - Inductive Families
   - Examples & Exercises

2. A bit about models
   - The groupoid model
   - Homotopy Type Theory

Hofmann & Streicher, 1994

A model of (Martin-Löf) Type Theory can be constructed in *groupoids*. The idea stems from the structure of *propositional equality*:

```
Inductive eq {A : Type} (a : A) : A → Type :=
  eq_refl : eq a a.

Definition eq_sym {A} (a b : A) : eq a b → eq b a.
Proof. destruct 1; reflexivity. Defined.

Definition eq_trans {A} {a b c : A} : eq a b → eq b c → eq a
c.
Proof. intros H H'. destruct H; apply H'. Defined.
```

The groupoid interpretation

A groupoid is just a type with a notion of "equivalence", that is an equivalence relation. Clearly eq is one.

- Each type gets interpreted as itself plus its equality.
- Each term is invariant under equality so transports that information (it's a groupoid functor).
- This forms a categorical model: $\Gamma \vdash t : T \Rightarrow [t] : [\Gamma] \to [T]$.

Equality is non-trivial

Notice how we can define also:

```
Definition eq_trans' {A} {a b c : A} :
  eq a b → eq b c → eq a c.
Proof. intros H H'; destruct H, H'; apply eq_refl. Defined.
```

eq_trans and eq_trans' are *not* definitionally equal, but again *propositionally*!

```
Lemma eq_trans_trans' {A} (a b c : A) (H : eq a b) (H' : eq b c)
  : eq (eq_trans H H') (eq_trans' H H').
Proof.
  Fail eq_refl.
  destruct H, H'. simpl.
  apply eq_refl.
Qed.
```

Equality is non-trivial II

This begs the question, are all equality proofs propositionally equal? Which reduces to:

```
Definition UIP {A} {a : A} := ∀ p : eq a a, p = eq_refl a.
```

- The groupoid model shows that this is an *independent* principle, from the usual MLTT and Coq, i.e. you can't prove it for all A.
- However it is provable for (first-order) datatypes, e.g. nat, bool etc... Hedberg (98) showed any type with *decidable* equality has UIP.
  - OTT/Epigram is a theory where all datatypes have UIP, even without decidability.
  - Its natural interpretation is in *setoids*, the degenerate case of *groupoids* with UIP.

Equality is non-trivial III: the non-trivial proofs

Why is UIP independent?

- Recall a groupoid is just a type with an equivalence relation.
- Find a type and an equivalence relation (intepreting some type of type theory) with multiple *different* proofs of equality.

Equality is non-trivial III: the non-trivial proofs

Why is UIP independent?

- Recall a groupoid is just a type with an equivalence relation.
- Find a type and an equivalence relation (intepreting some type of type theory) with multiple *different* proofs of equality.
- Canonical example: Isomorphisms of types are not unique. E.g. proofs of Iso bool bool. (Exercise: built the two inhabitants of that type).
- So types with iso proofs form a groupoid and type theory can be interpreted this way, contradicting UIP. Qed.

To Homotopy Type Theory

Comes V. Voevodsky (and A. Warren and S. Awodey and ...), looking at the groupoid model.

To Homotopy Type Theory

Comes V. Voevodsky (and A. Warren and S. Awodey and ...), looking at the groupoid model.

- They figure how to build a model in (weak) *infinity*-groupoids. i.e. the rich structure of equalities is infinite (proofs of proofs of equalities are related by proofs of proofs of proofs of equality etc... all they way up (*without* an end).)

To Homotopy Type Theory

Comes V. Voevodsky (and A. Warren and S. Awodey and ...), looking at the groupoid model.

- They figure how to build a model in (weak) *infinity*-groupoids. i.e. the rich structure of equalities is infinite (proofs of proofs of equalities are related by proofs of proofs of proofs of equality etc... all they way up (*without* an end).)
- The model validates a strong extensionality principle, the univalence principle.

Univalence rules

From univalence follows:

- Proof-irrelevance: all proof of the same logical statements are equal
- Propositional extensionality: all logical statements are equal if biequivalent
- Functional extensionality: all functions are equal if pointwise equal
- Invariance under isomorphism: all terms can be transported through isomorphisms of types.

This goes beyond making type theory a good language for classical mathematics, it's a new *foundation*. See the HoTT book.

From the HoTT model back to the syntax

Current state of the art:

- We don't have a computational interpretation of the principle (i.e. it's an inert axiom now, and the models are non-constructive)
- A cubical set model was introduced recently solving (part of) this issue (by Coquand et al).
- We get new datatypes called Higher Inductive Types allowing to form quotients internally, don't know how to give them a syntax and elimination rules.