MPRI 2-7-2 — Proof assistants — ~sozeau/teaching/MPRI-2-7-2-270114.pdf

Structures in Coq

January 27th 2014

Matthieu Sozeau
πr^2 project team
Inria Paris & PPS
matthieu.sozeau@inria.fr
www.pps…/~sozeau

Announcement

- Talk by Hendrik Tews, founder of FireEye, Inc. tomorrow at 2pm.
- PhD defense of J. Cretin on the 30th, at 2.30pm, room 227C of P7. “Erasable coercions: a unified approach to type systems”

Outline

1. Record Types
2. Mathematical Structures and Coercions
3. Type Classes and Canonical Structures
4. Monadic Programming with Type Classes
5. Interfaces and Implementations

Last time: Inductive Types

- Least Fixpoints of Polynomial Functors.
- Introduction and elimination
- Guardedness and strict positivity condition
- Dependent elimination and restricted eliminations
- Examples: ordinals and W-type.

Questions?

Record Types

1. **Record Types**
2. Mathematical Structures and Coercions
3. Type Classes and Canonical Structures
4. Monadic Programming using Type Classes
5. Interfaces and Implementations

Record Types

```
Record R (A : Type) : s :=
  { f1 : τ1; ..; fn : τn }.
```

≡

```
Inductive R (A : Type) : s :=
  Build_R (f1 : τ1) .. (fn : τn) : R A.
```

No indices, no recursion.

Projection is pattern-matching

```
Inductive R (A : Type) : s :=
  Build_R (f1 : τ1) .. (fn : τn) : R A.

Definition f_i (A : Type) (r : R A) :
  τi[f_i r/f_i] :=
 match r with Build_R t1 .. tn => ti end.
```

Definitional Equality

By construction:

```
fi (Build_rec t1 .. tn) →βδηι ti
```

But no η law:

```
Build_R (p1 t) .. (pn t) ≡ t.
```

Only derivable for Leibniz equality.

Exercises (10min):

- Define the sigma/dependent sum/pair type as a record.
- Prove the η rule (surjective pairing) for it.
- Define ℚ as a record of two integers.
- Define the equivalence of two rationals as a proposition.
- Definition addition of two rationals and show that it respects the equivalence (on pairwise equivalent arguments, it gives equivalent results).

Mathematical Structures and Coercions

1. Record Types
2. **Mathematical Structures and Coercions**
3. Type Classes and Canonical Structures
4. Monadic Programming using Type Classes
5. Interfaces and Implementations

Mathematical Structures

Defining composite structures as first-class objects:

```
Record monoid := { carrier : Type; unit : carrier;
  op : carrier -> carrier → carrier;
  left_unit : ∀ x, op x unit = x; … }.
```

Generic programming for any monoid:

```
Definition monsquare (m : monoid)
   (x : carrier m) : carrier m := op m x x.
```

Type and value dependency is essential to make this typecheck.

Mathematical Structures and Coercions
Exercise (10min)

- Define the interface of a monoid given partially above, and the (0, +) and (1, *) monoids on $\mathbb{N}$.
- Define a generic exponentiation operation a^n where n : $\mathbb{N}$ and a is in a monoid.
- Show that the exponentiation of the unit is always equal to the unit.

Coercions

```
Record monoid := { carrier : Type;
  unit : carrier; … }.
```

Generic programming for any monoid:

```
Definition monsquare (m : monoid)
  (x : m) : m := op m x x.
Coercion carrier : monoid >-> Sortclass.
```

The term m is **coerced** to a sort (its carrier) in the types.

Coercions

- Coercions are not displayed (by default, see Set Printing Coercions)
- The actual core term is **the same** as before
- Coherence of coercions (only one path up to ≡ between two types) and decidability ensured by a syntactic check.
- Uses: types embedded in records, injections (i.e. from $\mathbb{N}$ to $\mathbb{Z}$), modeling a subtyping relation…

```
Definition Z_of_nat (n : nat) : Z := Zpos n.

Coercion Z_of_nat : nat >-> Z.
```

Mathematical Structures and Coercions
Exercice (10min)

- Define the coercion from $\mathbb{N}$ to $\mathbb{B}$, sending 0 to false and everything else to the true constructor.
- Define the converse coercion from $\mathbb{B}$ to $\mathbb{N}$, giving value 1 to true.
- Show that $\mathbb{B} \to \mathbb{N} \to \mathbb{B}$ is the identity (pointwise)
- Which one should be a coercion?
- Redo the monoid exercise using a coercion for carrier

Type Classes

1. Record Types
2. Mathematical Structures and Coercions
3. **Type Classes and Canonical Structures**
4. Monadic Programming using Type Classes
5. Interfaces and Implementations

Type Classes

Introduced in Haskell (Wadler & Blott, POPL'89) and in Isabelle (Nipkow & Snelting, FPCA'91).

```
class Eq a where
   (==) :: a → a → Bool
instance Eq Bool where
   x == y = if x then y else not y

in :: Eq a ⇒ a → [a] → Bool
in x [] = False
in x (y : ys) = x == y || in x ys
```

Type Classes

Parametrized instances

```
instance (Eq a) => Eq [a] where
    [] == []                = True
    (x : xs) == (y : ys) = x == y && xs == ys
    _ == _                  = False
```

Super-classes

```
class Num a where
    (+) :: a → a → a …

class (Num a) ⇒ Fractional a where
    (=) :: a → a → a …
```

Type Classes in Coq

- Parametrized dependent records

  $\texttt{Record}\ \mathsf{Id}\ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$
  $\{\mathrm{f}_1 : \phi_1\ ;\ \cdots\ ; \mathrm{f}_m : \phi_m\}.$

  Instances are just definitions of conclusion $\mathsf{Id}\ \overrightarrow{t_n}$.

- Custom implicit arguments of projections

  $\mathrm{f}_1 : \forall \{\overrightarrow{\alpha_n : \tau_n}\}, \{\mathsf{Id}\ \overrightarrow{\alpha_n}\} \to \phi_1$

Elaboration example

$\lambda x\ y : \mathsf{bool}.\ \mathrm{eqb}\ x\ y$

$\rightsquigarrow$ { Implicit arguments }

$\lambda x\ y : \mathsf{bool}.\ @\mathrm{eqb}\ \_\ \_\ x\ y$

$\rightsquigarrow$ { Typing }

$\lambda x\ y : \mathsf{bool}.\ @\mathrm{eqb}\ (?_A : \mathtt{Type})\ (?_{eq} : \mathsf{Eq}\ ?_A)\ x\ y$

$\rightsquigarrow$ { Unification }

$\lambda x\ y : \mathsf{bool}.\ @\mathrm{eqb}\ \mathsf{bool}\ (?_{eq} : \mathsf{Eq}\ \mathsf{bool})\ x\ y$

$\rightsquigarrow$ { Proof search for $\mathsf{Eq}\ \mathsf{bool}$ returns $\mathrm{Eq\_bool}$ }

$\lambda x\ y : \mathsf{bool}.\ @\mathrm{eqb}\ \mathsf{bool}\ \mathrm{Eq\_bool}\ x\ y$

Type Classes

- Proof-search à la Prolog
- An elaboration (no kernel change) (Sozeau and Oury, 2008).

Inheritance

$\texttt{Class}\ \mathsf{Monoid}\ A := \{\ \mathrm{monop} : A \to A \to A\ ;\ \ldots\ \}$

$\texttt{Class}\ \mathsf{Group}\ A := \{\ \mathrm{grp\_mon} :> \mathrm{Monoid}\ A\ ;\ \ldots\ \}$

Substructures become **subinstances**:

$\texttt{Class}\ \mathsf{Monoid}\ A := \{\ \mathrm{monop} : A \to A \to A\ ;\ \ldots\ \}$

$\texttt{Class}\ \mathsf{Group}\ A := \{\ \mathrm{grp\_mon} : \mathrm{Monoid}\ A\ ;\ \ldots\ \}$

$\texttt{Instance}\ \mathrm{grp\_mon}\ `\{\mathsf{Group}\ A\} : \mathsf{Monoid}\ A.$

$\texttt{Definition}\ \mathrm{foo}\ `\{\mathsf{Group}\ A\}\ (x : A) : A := \mathrm{monop}\ x\ x.$

Similar to the existing `Structure`s based on coercive subtyping.

Example: numeric overloading

```
Class Num α := { zero : α ; one : α ; plus : α → α → α }.

Instance nat_num : Num nat :=
  { zero := 0%nat ; one := 1%nat ; plus := Peano.plus }.

Instance Z_num : Num Z :=
  { zero := 0%Z ; one := 1%Z ; plus := Zplus }.

Notation "0" := zero.
Notation "1" := one.
Infix "+" := plus.

Check (λ x : nat, x + (1 + 0 + x)).
Check (λ x : Z, x + (1 + 0 + x)).

(* Defaulting *)

Check (λ x, x + 1).
```

Example: value-dependent classes

```
Class Reflexive {A} (R : relation A) :=
  refl : ∀ x, R x x.

Instance eq_refl A : Reflexive (@eq A) := @refl_equal A.

Instance iff_refl : Reflexive iff.
Proof. red. tauto. Qed.

Goal ∀ P, P ↔ P.
Proof. apply refl. Qed.

Goal ∀ A (x : A), x = x.
Proof. intros A ; apply refl. Qed.

Ltac refl := apply refl.

Lemma foo `{Reflexive nat R} : R 0 0.
Proof. intros. refl. Qed.
```

Example: reification

```
Inductive formula :=
| cst : bool → formula
| not : formula → formula
| and : formula → formula → formula
| or : formula → formula → formula
| impl : formula → formula → formula.

Fixpoint interp f :=
  match f with
    | cst b ⇒ if b then True else False
    | not b ⇒ ¬ interp b
    | and a b ⇒ interp a ∧ interp b
    | or a b ⇒ interp a ∨ interp b
    | impl a b ⇒ interp a → interp b
  end.
```

Example: reification

```
Class Reify (prop : Prop) :=
  { reification : formula ;
    reify_correct : interp reification ↔ prop }.

Check (@reification : ∀ prop : Prop, Reify prop → formula).

Implicit Arguments reification [[Reify]].

Program Instance true_reif : Reify True :=
  { reification := cst true }.
Program Instance not_reif `(Rb : Reify b) : Reify (¬ b) :=
  { reification := not (reification b) }.

Example example_prop :=
  reification ((True ∧ ¬ False) → ¬ ¬ False).

Check (refl_equal _ : example_prop =
  impl (and (cst true) (not (cst false))) (not (not (cst false)))).
```

Exercise (30min)

Define a reifier for expressions in the monoid class, a reflexive tactic for simplifying monoidal expressions (unit laws), and an overloaded lemma to apply it.

Canonical Structures

Another way to use records to represent structures, capable of doing much the same as type classes, but based on unification instead of general proof search. At the basis of the ssreflect plugin.

See Mahboubi and Tassi [ITP'13] for a gentle introduction and comparison, and Ziliani et al [ICFP'11] for the previous examples using both representations.

Monadic Programming with Type Classes

1. Record Types
2. Mathematical Structures and Coercions
3. Type Classes and Canonical Structures
4. **Monadic Programming with Type Classes**
5. Interfaces and Implementations

Monadic programming: ML vs Haskell

```
val counter : unit → nat

let counter =

  let x = ref 0 in

    fun () =>

      let x’ = !x in

          x := x’ + 1; x’
```

```
type NatState α = nat → α × nat

return :: α → NatState α

bind :: NatState α → (α → NatState
β) → NatState β

get :: NatState nat

put :: nat → NatState ()

counter :: NatState nat

counter = do x’ <- get;

             put (x’ + 1);

             return x’

counter’ :: () → nat

counter’ () = ???
```

Monad

In our setting, a monad will be defined as:

```
Record Monad (M : Type → Type) :=
  {  return : ∀ {A}, A → M A;
     bind : ∀ {A B}, M A → (A → M B) → M B;
     bind_assoc {A B f g} (m : M A) :
       bind (bind m f) g =
       bind m (fun x : A ⇒ bind (f x) g);
     bind_return_left : …; bind_return_right : … }
```

Monad Instances

Instances of the monad interface:

- id (identity monad)
- option (partiality monad)
- list (“nondeterminism” monad, unit is [x], bind is concat ∘ map)
- continuations, “computation” …

Exercises (±20min)

- Define Monad as a type class with overloaded bind and return/unit operations
- Define monadic multiplication "mult" of type M (M A) → M A for any monad. State and prove what is its relation to the unit.
- Define the identity instance.
- Define the option instance (return is injection, bind propagates "errors" represented as `None`), there is a corresponding "error"/"nothing" action to define. To prove the laws, you will need to use the axiom in `Coq.Logic.FunctionalExtensionality`.
- Define the generic mapM operator for any monad (it processes the effects from left to right). Generalize it to a foldM.
- Optional: Using this, write a partial function turning an integer to a natural, and a function that takes a list of integers and returns the sum of their positive numbers, if they're all positive only.
- Optional: Using the state monad, verify a tree labeling algorithm:

  http://www.pps.univ-paris-diderot.fr/~sozeau/teaching/Classes/Monad.v

Interfaces and Implementations: Fibonacci

1. Record Types
2. Mathematical Structures and Coercions
3. Type Classes and Canonical Structures
4. Monadic Programming with Type Classes
5. **Interfaces and Implementations**

Interfaces vs Implementations

Software engineering principle applied to proof engineering:

Work with **interfaces** instead of **implementations**.

An illustrative example:

Verifying a fast implementation of Fibonacci.

A correct exponentiation algorithm

The following definition is very naïve, but obviously correct:

```
Fixpoint power (a : Z) (n : nat) :=
  match n with
    | 0%nat ⇒ 1
    | S p ⇒ a × power a p
  end.

Eval vm_compute in power 2 40.
= 1099511627776 : Z
```

An efficient exponentiation algorithm

This one is more efficient but relies on a more elaborate property:

```
Function binary_power_mult (acc x : Z) (n : nat)
  {measure (fun i⇒i) n} : Z :=
  match n with
    | 0%nat ⇒ acc
    | _ ⇒ if Even.even_odd_dec n
      then binary_power_mult acc (x × x) (div2 n)
      else binary_power_mult (acc × x) (x × x) (div2 n)
  end.

Definition binary_power (x:Z) (n:nat) :=
  binary_power_mult 1 x n.

Eval vm_compute in binary_power 2 40.
= 1099511627776 : Z

Goal binary_power 2 234 = power 2 234.
Proof. reflexivity. Qed.
```

How to proceed?

- Is binary_power correct (w.r.t. power)?
- Is it worth proving this correctness only for powers of integers?
- And prove it again for powers of real numbers, matrices?

NO!

Program with interfaces, here a **monoid**.

Support for overloading

Quantification on parameters:

```
Definition two {A dot one} {M : @Monoid A dot one} :=
  dot one one.
```

Using implicit generalization:

```
Generalizable Variables A dot one.
Definition three `{Monoid A dot one} := dot two one.
```

Global names for parameters:

```
Definition monop `{Monoid A dot one} := dot.
Definition monunit `{Monoid A dot one} := one.
```

Generic notations:

```
Infix "×" := monop.
Notation "1" := monunit.
```

A generic exponentiation algorithm

Generic power and binary_power.

```
Section Power.
  Context `{Monoid A dot one}.

  Fixpoint power (a : A) (n : nat) :=
    match n with
      | 0%nat ⇒ 1
      | S p ⇒ a × (power a p)
    end.

  Lemma power_of_unit : ∀ n : nat, power 1 n = 1.
  Proof. ...   Qed.
```

A generic, efficient exponentiation algorithm

```
Function binary_power_mult (acc x : A) (n : nat)
  {measure (fun i⇒i) n} : A :=
  match n with
    | 0%nat ⇒ acc
    | _ ⇒ if Even.even_odd_dec n
        then binary_power_mult acc (x × x) (div2 n)
        else binary_power_mult (acc × x) (x × x) (div2 n)
  end.
Definition binary_power (x : A) (n : nat) :=
  binary_power_mult 1 x n.

Lemma binary_spec x n : power x n = binary_power x n.
Proof.  ...   Qed.

End Power.
```

Exercise (1h, for next week)

Get ~sozeau/teaching/MPRI-2-7-2-270114-Fibonacci.v

- Define the monoid instance for multiplication on $\mathbb{Z}$
- Define a commutative semiring type class, comprising 2 monoids, one for multiplication and one for addition and the additional laws (commutativity and one distributive law).
- Define an instance for $\mathbb{Z}$ (you can leave some obligations admitted).
- Define square 2x2 α-valued matrices, for α supporting a commutative semiring structure.
- Define the multiplicative and additive monoids on these matrices and the semiring structure.
- Using the characterization of the Fibonacci sequence below, define Fibonacci in terms of exponentiation of matrices.
- Show that this fast version is equivalent to the naive definition of Fibonacci.
- Experiment with vm_compute and compute on these different implementations (using power, binary power)…

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}.$$