Type Classes for Mathematical Formalizations in Coq

Matthieu Sozeau

October 3rd 2012
IAS, Princeton

Project Team πr^2
INRIA Paris &
PPS, Paris 7 University

Outline

Goal

Enhancing type inference through overloading:

- For generic programming with interfaces rather than concrete implementations.
- For generic proof scripts: refer to proofs by semantic concept rather than name. E.g. `reflexivity of R` instead of `R_refl`.

In general, allows inference of arbitrary additional structure on a given type or value.

Demo

A cheap implementation

- Parametrized dependent records

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

A cheap implementation

- Parametrized dependent records

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

A cheap implementation

- 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 type $\mathsf{Id}\ \overrightarrow{t_n}$.

A cheap implementation

- 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 type Id $\overrightarrow{t_n}$.

- Custom implicit arguments of projections

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

A cheap implementation

- 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 type $\mathsf{Id}\ \overrightarrow{t_n}$.

- Custom implicit arguments of projections

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

Elaboration with classes, an example

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { Implicit arguments }

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { Implicit arguments }

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

$\rightsquigarrow$ { Unification }

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { Implicit arguments }

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

$\rightsquigarrow$ { Unification }

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

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

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

Monomorphic binary power

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 tail-recursive version

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.
```

Questions

- Is binary_power correct (w.r.t. power)?

Questions

- Is binary_power correct (w.r.t. power)?
- Is it worth proving this correctness only for powers of integers?

Questions

- 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?

Questions

- 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!

We aim to prove the equivalence between power and binary_power for any structure consisting of a binary associative operation that admits a neutral element, i.e. any monoid.

Monoids

$\texttt{Class}$ Monoid $\{A{:}\texttt{Type}\}$ $(dot : A \to A \to A)$ $(one : A) : \texttt{Type} :=$
$\{\ \mathrm{dot\_assoc} : \forall\ x\ y\ z : A,\ dot\ x\ (dot\ y\ z) = dot\ (dot\ x\ y)\ z;$
$\quad \mathrm{one\_left} : \forall\ x,\ dot\ one\ x = x;$
$\quad \mathrm{one\_right} : \forall\ x,\ dot\ x\ one = x\ \}.$

Operations as parameters to ease sharing, allows to specify multiple monoids on the same carrier unambiguously, e.g. Monoid 0 *plus* and Monoid 1 *mult*.

Implicit Generalization

Quantification becomes verbose:

```
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.
```

Generic notations

One can define trivial projections to recover global names for parameters:

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

and the corresponding generic notations:

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

Generic power

Let's redefine power and binary_power generically.

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

All following definitions are overloaded over any Monoid structure.

```
  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.
```

Generic binary exponentiation

```
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.
```

Let's build a Monoid instance.

`Instance` ZMult : Monoid Zmult 1%Z.
`Proof`. `split`.
`subgoal` 1 *is*:
$\forall\ x\ y\ z : Z,\ x \times (y \times z) = x \times y \times z$
`subgoal` 2 *is*:
$\forall\ x : Z,\ 1 \times x = x$
`subgoal` 3 *is*:
$\forall\ x : Z,\ x \times 1 = x$

... `Qed`.

Instantiation

We can now use the overloaded power on our new Monoid.

```
About power.
```

$: \forall\ (A : \texttt{Type})\ (dot : A \to A \to A)\ (one : A),\ \mathsf{Monoid}\ dot\ one \to A \to \mathsf{nat} \to A$

Arguments A, dot, one, H are implicit and maximally inserted

Instantiation

We can now use the overloaded power on our new Monoid.

```
About power.
: ∀ (A : Type) (dot : A → A → A) (one : A), Monoid dot one →
A → nat → A
Arguments A, dot, one, H are implicit and maximally inserted
```

```
Set Printing Implicit.
Check power 2 100.
@power Z Z.mul 1 ZMult 2 100 : Z
```

Instantiation

We can now use the overloaded power on our new Monoid.

About power.
$: \forall\ (A : \texttt{Type})\ (dot : A \to A \to A)\ (one : A),\ \mathsf{Monoid}\ dot\ one \to A \to \mathsf{nat} \to A$
Arguments A, dot, one, H are implicit and maximally inserted

Set Printing Implicit.
Check power 2 100.
@power Z *Z.mul* 1 ZMult 2 100 : Z

Compute power 2 100.
$= 1267650600228229401496703205376 : \mathsf{Z}$

Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- Undeterministic proof-search

Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- Undeterministic proof-search
  $\Rightarrow$ Determinacy inference (Kriener and King, ICLP'11)

Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- Undeterministic proof-search
  $\Rightarrow$ Determinacy inference (KRIENER and KING, ICLP'11)
- No forward reasoning or reordering of constraints

Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- Undeterministic proof-search
  $\Rightarrow$ Determinacy inference (KRIENER and KING, ICLP'11)
- No forward reasoning or reordering of constraints
  $\Rightarrow$ Mode analysis (à la PROLOG, TWELF)

Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- Undeterministic proof-search
  $\Rightarrow$ Determinacy inference (KRIENER and KING, ICLP'11)
- No forward reasoning or reordering of constraints
  $\Rightarrow$ Mode analysis (à la PROLOG, TWELF)
- Risk of non-termination

Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- Undeterministic proof-search
  $\Rightarrow$ Determinacy inference (Kriener and King, ICLP'11)
- No forward reasoning or reordering of constraints
  $\Rightarrow$ Mode analysis (à la Prolog, Twelf)
- Risk of non-termination
  $\Rightarrow$ Termination analysis, requires modes

Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- Undeterministic proof-search
  $\Rightarrow$ Determinacy inference (KRIENER and KING, ICLP'11)
- No forward reasoning or reordering of constraints
  $\Rightarrow$ Mode analysis (à la PROLOG, TWELF)
- Risk of non-termination
  $\Rightarrow$ Termination analysis, requires modes
- Little sharing and intelligence in the proof-search

Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification
Hope These are all researched in the logic programming community

- Undeterministic proof-search
  $\Rightarrow$ Determinacy inference (KRIENER and KING, ICLP'11)
- No forward reasoning or reordering of constraints
  $\Rightarrow$ Mode analysis (à la PROLOG, TWELF)
- Risk of non-termination
  $\Rightarrow$ Termination analysis, requires modes
- Little sharing and intelligence in the proof-search
  $\Rightarrow$ Focusing, strategies.

Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- Undeterministic proof-search
  $\Rightarrow$ Determinacy inference (KRIENER and KING, ICLP'11)
- No forward reasoning or reordering of constraints
  $\Rightarrow$ Mode analysis (à la PROLOG, TWELF)
- Risk of non-termination
  $\Rightarrow$ Termination analysis, requires modes
- Little sharing and intelligence in the proof-search
  $\Rightarrow$ Focusing, strategies.
- Scoping of instances... through modules only.

The End

Thank you!