Type Classes for Mathematical Formalizations in CoQ

Matthieu Sozeau

October 3rd 2012
IAS, Princeton
Outline

1. Type Classes theory
2. Demonstration
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 $\text{R_refl}$.

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

- Parametrized dependent records

```
Class Id (α₁:τ₁)· · · (αₙ:τₙ) :=
{f₁:φ₁; · · · ;fm:φm}.
```
Parametrized dependent records

\[
\text{Record } \text{Id} (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\]
Parametrized dependent records

```
Record Id (α_1 : τ_1) ⋯ (α_n : τ_n) :=
{f_1 : φ_1 ; ⋯ ; f_m : φ_m}.
```

Instances are just definitions of type \( \text{Id} \rightarrow t_n \).
A cheap implementation

- Parametrized dependent records

\[
\text{Record } \text{Id} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\]
Instances are just definitions of type \( \text{Id} \arr t_n \).

- Custom implicit arguments of projections

\[
f_1 : \forall \ \alpha_n : \tau_n , \ \text{Id} \arr \alpha_n \rightarrow \phi_1
\]
A cheap implementation

- Parametrized dependent records

```
Record ld (α₁ : τ₁) ⋯ (αₙ : τₙ) :=
{f₁ : φ₁ ; ⋯ ; fₘ : φₘ}.
```

Instances are just definitions of type `ld tₙ`.

- Custom implicit arguments of projections

```
f₁ : ∀{αₙ : τₙ}, ld αₙ → φ₁
```
Elaboration with classes, an example

\[(\lambda x \ y : \text{bool}. \ \text{eqb } x \ y)\]
Elaboration with classes, an example

\[(\lambda x\ y:\ \text{bool}.\ \text{eqb} \ x \ y)\]

\[\leadsto \{ \text{Implicit arguments} \}\]

\[(\lambda x\ y:\ \text{bool}.\ \@\text{eqb} (\ ?_A:\ \text{Type}) (\ ?_{eq}:\ \text{Eq} \ ?_A) \ x \ y)\]

\[\leadsto \{ \text{Unification} \}\]

\[(\lambda x\ y:\ \text{bool}.\ \@\text{eqb} \ ?_A: \text{Type} \ ?_{eq}:\ \text{Eq} \ ?_A) \ x \ y\]
Elaboration with classes, an example

\[(\lambda x \ y : \text{bool}. \ \text{eqb} \ x \ y)\]

\[\leadsto \{ \text{Implicit arguments} \}\]

\[(\lambda x \ y : \text{bool}. \ \text{@eqb} (\ ?_A : \text{Type} ) (\ ?_{eq} : \text{Eq} \ ?_A ) \times y)\]

\[\leadsto \{ \text{Unification} \}\]

\[(\lambda x \ y : \text{bool}. \ \text{@eqb} \text{bool} (\ ?_{eq} : \text{Eq} \text{ bool} ) \times y)\]
Elaboration with classes, an example

\[(\lambda x \ y : \text{bool}. \ \text{eqb} \ x \ y)\]
\[\leadsto \{ \text{Implicit arguments} \} \]
\[(\lambda x \ y : \text{bool}. \ \text{@eqb} (\ ?_A : \text{Type}) (\ ?_{eq} : \text{Eq} \ \ ?_A) \ x \ y)\]
\[\leadsto \{ \text{Unification} \} \]
\[(\lambda x \ y : \text{bool}. \ \text{@eqb} \ \text{bool} (\ ?_{eq} : \text{Eq} \ \text{bool}) \ x \ y)\]
\[\leadsto \{ \text{Proof search for Eq bool returns Eq bool} \} \]
\[(\lambda x \ y : \text{bool}. \ \text{@eqb} \ \text{bool} \ \text{Eq bool} \ x \ y)\]
1 Type Classes in theory

2 Demonstration
   ■ Technically

3 Exponentiation

4 Current issues and perspectives
The following definition is very naïve, but obviously correct:

\[
\text{Fixpoint power (} a : \mathbb{Z}) (n : \text{nat}) := \text{match } n \text{ with} \begin{align*}
| 0\% \text{nat} & \Rightarrow 1 \\
| S \ p & \Rightarrow a \times \text{power } a \ p
\end{align*} \text{ end.}
\]

\text{Eval vm_compute in power 2 40.} \\
\text{= 1099511627776 : } \mathbb{Z}
An efficient tail-recursive version

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

Function $\text{binary\_power\_mult} (\text{acc} \ x : Z) (n : \text{nat})$

{\begin{aligned}
\{ \text{measure } (\text{fun } i \mapsto i) \ n \} : Z := \\
\text{match } n \text{ with} \\
\quad | 0\%\text{nat} \Rightarrow \text{acc} \\
\quad | _ \Rightarrow \text{if } \text{Even.even\_odd\_dec } n \\
\quad \quad \text{then } \text{binary\_power\_mult } \text{acc} (x \times x) (\text{div2 } n) \\
\quad \quad \text{else } \text{binary\_power\_mult} (\text{acc} \times x) (x \times x) (\text{div2 } n) \\
\end{aligned}}$

Definition $\text{binary\_power} (x:Z) (n:\text{nat}) := \\
\text{binary\_power\_mult} 1 \times n.$

Eval $\text{vm\_compute in binary\_power} 2 \ 40.$

$= 1099511627776 : Z$

Goal $\text{binary\_power} 2 \ 234 = \text{power} 2 \ 234.$

Proof. reflexivity. Qed.
Questions

▶ Is \texttt{binary\_power} correct \textit{(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.
Class Monoid \( \{A:\text{Type}\} (dot : A \to A \to A) (one : A) : \text{Type} := \{ \)
\begin{align*}
\text{dotassoc} & : \forall x y z : A, dot x (dot y z) = dot (dot x y) z; \\
\text{oneleft} & : \forall x, dot one x = x; \\
\textoneright} & : \forall x, dot x one = x \}.
\end{align*}
\)

Operations as parameters to ease sharing, allows to specify multiple monoids on the same carrier unambiguously, e.g. Monoid 0 \textit{plus} and Monoid 1 \textit{mult}.
Implicit Generalization

Quantification becomes verbose:

\[
\text{Definition two } \{ A \text{ dot one} \} \{ M : @\text{Monoid A dot one} \} := \text{dot one one}.
\]

Using implicit generalization:

\[
\text{Generalizable Variables } A \text{ dot one}.
\]

\[
\text{Definition three } \{ \text{Monoid A dot one} \} := \text{dot two one}.
\]
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.
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.
Function \textit{binary\_power\_mult} \((acc \times : A) (n : \text{nat})\)

\[
\{\text{measure (fun } i \mapsto i)\ n\} : A :=
\begin{align*}
\text{match } n \text{ with} \\
\mid 0\%\text{nat } \Rightarrow & acc \\
\mid _\_ \Rightarrow & \text{if Even.even\_odd\_dec } n \\
& \text{ then binary\_power\_mult } acc \ (x \times x) \ (\text{div2 } n) \\
& \text{ else binary\_power\_mult } (acc \times x) \ (x \times x) \ (\text{div2 } n) \\
\end{align*}
\]

Definition \textit{binary\_power} \((x : A) (n : \text{nat}) :=\)

\[
\text{binary\_power\_mult } 1 \times n.
\]

Lemma \textit{binary\_spec} \(x \ n : \text{power } x \ n = \text{binary\_power } x \ n.\)

Proof. \ldots \ Qed.

End Power.
Let’s build a Monoid instance.

```coq
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.
```
We can now use the overloaded \texttt{power} on our new \texttt{Monoid}.

About \texttt{power}.
\[
\forall (A : \text{Type}) \ (\text{dot} : A \to A \to A) \ (\text{one} : A), \ \text{Monoid} \ \text{dot} \ \text{one} \to A \to \text{nat} \to A
\]

Arguments \texttt{A}, \texttt{dot}, \texttt{one}, \texttt{H} are implicit and maximally inserted
We can now use the overloaded \texttt{power} on our new \texttt{Monoid}.

\textbf{About \texttt{power}.}

\begin{align*}
\forall (A : \text{Type}) \ (\text{dot} : A \to A \to A) \ (\text{one} : A), \ \text{Monoid} \ \text{dot} \ \text{one} \to A \to \text{nat} \to A
\end{align*}

\textit{Arguments A, dot, one, H are implicit and maximally inserted}

Set Printing Implicit.

Check \texttt{power 2 100}.

@\texttt{power Z Z.mul 1 ZMult 2 100 : Z}
We can now use the overloaded \texttt{power} on our new \texttt{Monoid}.

\textbf{About power}.

: \( \forall (A : \text{Type}) \cdot (\text{dot} : A \to A \to A) \cdot (\text{one} : A), \text{Monoid dot one} \to A \to \text{nat} \to A \)

\textit{Arguments A, dot, one, H are implicit and maximally inserted}

Set Printing Implicit.

Check \texttt{power 2 100}.

\texttt{@power Z Z.mul 1 ZMult 2 100 : Z}

Compute \texttt{power 2 100}.

\( = 1267650600228229401496703205376 : Z \)
1 Type Classes in theory

2 Demonstration
   - Technically

3 Exponentiation

4 Current issues and perspectives
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
  ⇒ Determinacy inference (KRIENER and KING, ICLP’11)
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
Proof search efficiency and control issues...

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

- Undeterministic proof-search
  ⇒ Determinacy inference (Kriener and King, ICLP’11)
- No forward reasoning or reordering of constraints
  ⇒ 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
  ⇒ Determinacy inference (Kriener and King, ICLP’11)

- No forward reasoning or reordering of constraints
  ⇒ 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
  ⇒ Determinacy inference (*Kriener* and *King*, ICLP’11)

- No forward reasoning or reordering of constraints
  ⇒ Mode analysis (à la *Prolog*, *Twelf*)

- Risk of non-termination
  ⇒ Termination analysis, requires modes

- Little sharing and intelligence in the proof-search
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.
Proof search efficiency and control issues...

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

- Undeterministic proof-search
  ⇒ Determinacy inference (Kriener and King, ICLP’11)
- No forward reasoning or reordering of constraints
  ⇒ Mode analysis (à la Prolog, Twelf)
- Risk of non-termination
  ⇒ Termination analysis, requires modes
- Little sharing and intelligence in the proof-search
  ⇒ Focusing, strategies.
- Scoping of instances... through modules only.
Thank you!