Generalized rewriting in Coq

Simcha van Collem
Advised by Jules Jacobs

Radboud University Nijmegen

24 January 2023

Outline

First-Class Type Classes

Sozeau and Oury, 2008

Notation overloading

- $+$ has different meanings in different contexts
- Refer to monoid $(M, \varepsilon, \cdot)$ as M
- Make pen and paper proofs easier

Type classes in Haskell

```haskell
class Monoid a where
    mempty :: a
    (<>) :: a -> a -> a

instance Monoid [a] where
    mempty = []
    (<>) = (++)

foldMap :: Monoid m => (a -> m) -> [a] -> m
foldMap f [] = mempty
foldMap f (x :: xs) = f x <> foldMap f xs
```

Records

```
Record Person := {
  age  : nat;
  name : string;
}
```

```
Inductive Person :=
| Build_Person : nat -> string -> Person
```

```
age  : Person -> nat
name : Person -> string
```

```
Build_Person 37 "Alice"          : Person
{| age := 42; name := "Bob" |} : Person
```

Dependent records

```
Record Monoid := {
  carrier : Type;
  neutral : carrier;
  op      : carrier -> carrier -> carrier;
  idL     : forall x, op neutral x = x;
  idR     : forall x, op x neutral = x;
  assoc   : forall x y z, op (op x y) z = op x (op y z);
}
```

Type classes

```
Class Monoid := {
  carrier : Type;
  neutral : carrier;
  op      : carrier -> carrier -> carrier;
  idL     : forall x, op neutral x = x;
  idR     : forall x, op x neutral = x;
  assoc   : forall x y z, op (op x y) z = op x (op y z);
}
```

Type classes

```
Class Monoid := {
  carrier : Type;
  neutral : carrier;
  op      : carrier -> carrier -> carrier;
  idL     : forall x, op neutral x = x;
  idR     : forall x, op x neutral = x;
  assoc   : forall x y z, op (op x y) z = op x (op y z);
}

Definition foo (M : Monoid) : carrier M = nat -> ...
```

Type classes

```
Class Monoid (carrier : Type) := {
  neutral : carrier;
  op      : carrier -> carrier -> carrier;
  idL     : forall x, op neutral x = x;
  idR     : forall x, op x neutral = x;
  assoc   : forall x y z, op (op x y) z = op x (op y z);
}

Definition foo (M : Monoid nat) : ...
```

Type class instance

```
Instance monoid_nat : Monoid nat :=
  {| neutral := 0;
     op := plus;
     ... |}.

Instance monoid_pair `{Monoid A, Monoid B} : Monoid (A * B) :=
  {| neutral := (neutral, neutral);
     op := fun '(a, b) '(a', b') => (op a a', op b b');
     ... |}.
```

Fold map

```
Fixpoint fold_map `{Monoid B} {A : Type}
    (f : A -> B) (l : list A) : B :=
  match l with
  | [] => neutral
  | x :: xs => op (f x) (fold_map f xs)
  end.

Definition sum : list (nat * nat) -> nat * nat :=
  fold_map id.
```

Superclasses

```
Class Semigroup (A : Type) := {
  op : A -> A -> A;
  assoc : forall a b c, op (op a b) c = op a (op b c);
}.

Class Monoid (A : Type) `{Semigroup A} := {
  neutral : A;
  idL : forall a, op neutral a = a;
  idR : forall a, op a neutral = a;
}.
```

Substructures

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

Class Symmetric {A : Type} (R : relation A) :=
  sym : forall x y, R x y -> R y x.

Class Transitive {A : Type} (R : relation A) :=
  trans : forall x y z, R x y -> R y z -> R x z.

Class Equivalence {A : Type} (R : relation A) :=
  { Equivalence_Reflexive  :> Reflexive A R;
    Equivalence_Symmetric  :> Symmetric A R;
    Equivalence_Transitive :> Transitive A R }.
```

A New Look at Generalized Rewriting in Type Theory

Sozeau, 2009

Leibniz equality is not enough

```
Definition set : Type := list nat.
Definition set1 : set := [1 ; 2].
Definition set2 : set := [2 ; 1].

set1 <> set2

Definition eqset : relation set := ...
eqset set1 set2
```

Proper type class

```
Class Proper {A : Type} (R : relation A) (m : A) : Prop :=
  proper : R m m.

Instance reflexive_proper `{Reflexive A R}
  (x : A) : Proper R x.
Proof. reflexivity. Qed.
```

Respectful

```
Definition respectful {A B : Type}
    (R : relation A) (R' : relation B)
    : relation (A -> B) :=
  fun (f g : A -> B) =>
    forall (x y : A),
      R x y -> R' (f x) (g y)

Notation " R ==> R' " := (respectful R R')
  (right associativity, at level 55) : signature_scope.

Definition flip {A : Type} (R : relation A) : relation A :=
  fun (x y : A) => R y x.

Notation " R --> R' " := (respectful (flip R) R')
  (right associativity, at level 55) : signature_scope.
```

Contrapostitive using Proper

```
Definition respectful {A B : Type}
    (R : relation A) (R' : relation B)
    : relation (A -> B) :=
  fun (f g : A -> B) =>
    forall (x y : A),
      R x y -> R' (f x) (g y)

forall (P Q : Prop),
  (Q -> P) -> (~ P -> ~ Q)
```

Contrapostitive using Proper

```
Definition respectful {A B : Type}
    (R : relation A) (R' : relation B)
    : relation (A -> B) :=
  fun (f g : A -> B) =>
    forall (x y : A),
      R x y -> R' (f x) (g y)

forall (P Q : Prop),
  (impl Q P) -> (impl (~ P) (~ Q))
```

Contrapostitive using Proper

```
Definition respectful {A B : Type}
    (R : relation A) (R' : relation B)
    : relation (A -> B) :=
  fun (f g : A -> B) =>
    forall (x y : A),
      R x y -> R' (f x) (g y)

forall (P Q : Prop),
  (impl Q P) -> (impl (not P) (not Q))
```

Contrapostitive using Proper

```
Definition respectful {A B : Type}
    (R : relation A) (R' : relation B)
    : relation (A -> B) :=
  fun (f g : A -> B) =>
    forall (x y : A),
      R x y -> R' (f x) (g y)

Proper (impl --> impl) not
```

Proper for union?

```
Definition respectful {A B : Type}
    (R : relation A) (R' : relation B)
    : relation (A -> B) :=
  fun (f g : A -> B) =>
    forall (x y : A),
      R x y -> R' (f x) (g y)

Instance union_proper : Proper (eqset ==> eqset ==> eqset) union.
```

Proper for union?

```
Definition respectful {A B : Type}
    (R : relation A) (R' : relation B)
    : relation (A -> B) :=
  fun (f g : A -> B) =>
    forall (x y : A),
      R x y -> R' (f x) (g y)

Instance union_proper : Proper (eqset ==> eqset ==> eqset) union.
```

```
eqset ==> eqset
  =
fun (f g : set -> set) => forall (s s' : set),
  eqset s s' -> eqset (f s) (g s')
```

Proper for union?

```
Definition respectful {A B : Type}
    (R : relation A) (R' : relation B)
    : relation (A -> B) :=
  fun (f g : A -> B) =>
    forall (x y : A),
      R x y -> R' (f x) (g y)

Instance union_proper : Proper (eqset ==> eqset ==> eqset) union.
```

```
eqset ==> eqset
  =
fun (f g : set -> set) => forall (s s' : set),
  eqset s s' -> eqset (f s) (g s')
```

```
eqset ==> (eqset ==> eqset)
  =
fun (f g : set -> (set -> set)) => forall (t t' : set),
  eqset t t' -> (eqset ==> eqset) (f t) (g t')
```

Proper for union?

```
Definition respectful {A B : Type}
    (R : relation A) (R' : relation B)
    : relation (A -> B) :=
  fun (f g : A -> B) =>
    forall (x y : A),
      R x y -> R' (f x) (g y)

Instance union_proper : Proper (eqset ==> eqset ==> eqset) union.
```

```
eqset ==> eqset
  =
fun (f g : set -> set) => forall (s s' : set),
  eqset s s' -> eqset (f s) (g s')
```

```
eqset ==> (eqset ==> eqset)
  =
fun (f g : set -> (set -> set)) => forall (t t' : set),
  eqset t t' -> forall (s s' : set),
    eqset s s' -> eqset ((f t) s) ((g t') s')
```

Proper for union?

```
Definition respectful {A B : Type}
    (R : relation A) (R' : relation B)
    : relation (A -> B) :=
  fun (f g : A -> B) =>
    forall (x y : A),
      R x y -> R' (f x) (g y)

Instance union_proper : Proper (eqset ==> eqset ==> eqset) union.
```

```
eqset ==> eqset
  =
fun (f g : set -> set) => forall (s s' : set),
  eqset s s' -> eqset (f s) (g s')
```

```
(eqset ==> (eqset ==> eqset)) union union
  =
forall (t t' : set),
  eqset t t' -> forall (s s' : set),
    eqset s s' -> eqset (union t s) (union t' s')
```

How to read Propers

Given

- `f : A1 -> A2 -> ... -> An -> B`
- `Ri : relation Ai` for each `i`
- `R : relation B`
- `xi : Ai` and `yi : Ai` for each `i`

Then `Proper (R1 ==> R2 ==> ... ==> Rn ==> R) f` represents the fact that

- If `Ri xi yi` for each `i`,
- then `R (f x1 ... xn) (f y1 ... yn)`

Rewrite judgement

Constraints of the form $?_x : \Gamma \vdash A$

Rewrite judgement

Constraints of the form $?_x : \Gamma \vdash A$
Given a rewriting lemma ρ, the judgement

$$\Gamma \mid \psi \vdash t \rightsquigarrow^{R}_{p} t' \dashv \psi'$$

states that in environment Γ with constraints ψ, t is rewritten to t' with respect to relation R and p a proof of $R\ t\ t'$, generating new constraints ψ'.

Rewrite judgement

Constraints of the form $?_x : \Gamma \vdash A$
Given a rewriting lemma ρ, the judgement

$$\Gamma \mid \psi \vdash t \rightsquigarrow^{R}_{p} t' \dashv \psi'$$

states that in environment Γ with constraints ψ, t is rewritten to t' with respect to relation R and p a proof of $R\ t\ t'$, generating new constraints ψ'.
Given a goal $\Gamma \vdash t$ and a rewrite lemma ρ, we want to find the following judgement

$$\Gamma \mid \emptyset \vdash t \rightsquigarrow^{\text{flip impl}}_{p} t' \dashv \psi'$$

which reduces the goal to $\Gamma \vdash t'$.

Rewrite example

- Rewrite goal `P x` using rewrite lemma ρ : `eqA x y`
- Goal: find judgement of the form
  $\Gamma \mid \emptyset \vdash P\ x \rightsquigarrow^{\text{flip impl}}_{\dots} \dots \dashv \psi'$

Rewrite example

- Rewrite goal `P x` using rewrite lemma ρ : `eqA x y`
- Goal: find judgement of the form
  $\Gamma \mid \emptyset \vdash P\ x \rightsquigarrow^{\text{flip impl}}_{\dots} \dots \dashv \psi'$
- $\Gamma \mid \emptyset \vdash P \rightsquigarrow^{?_S}_{?_m} P \dashv$
  $\underbrace{\{?_S : \Gamma \vdash \text{relation (A -> Prop)}, ?_m : \Gamma \vdash \text{Proper}\ ?_S\ P\}}_{\psi}$

Rewrite example

- Rewrite goal `P x` using rewrite lemma ρ : `eqA x y`
- Goal: find judgement of the form
  $\Gamma \mid \emptyset \vdash P\ x \rightsquigarrow^{\text{flip impl}}_{\dots} \dots \dashv \psi'$
- $\Gamma \mid \emptyset \vdash P \rightsquigarrow^{?_S}_{?_m} P \dashv \underbrace{\{?_S : \Gamma \vdash \text{relation (A -> Prop)}, ?_m : \Gamma \vdash \text{Proper}\ ?_S\ P\}}_{\psi}$
- $\Gamma \mid \psi \vdash x \rightsquigarrow^{\text{eqA}}_{\rho} y \dashv \psi$

Rewrite example

- Rewrite goal `P x` using rewrite lemma ρ : `eqA x y`
- Goal: find judgement of the form
  $\Gamma \mid \emptyset \vdash P\ x \rightsquigarrow^{\text{flip impl}}_{\dots} \dots \dashv \psi'$
- $\Gamma \mid \emptyset \vdash P \rightsquigarrow^{?_S}_{?_m} P \dashv$
  $\underbrace{\{?_S : \Gamma \vdash \text{relation (A -> Prop)}, ?_m : \Gamma \vdash \text{Proper}\ ?_S\ P\}}_{\psi}$
- $\Gamma \mid \psi \vdash x \rightsquigarrow^{\text{eqA}}_{\rho} y \dashv \psi$
- $\Gamma \mid \emptyset \vdash P\,x \rightsquigarrow^{?_T}_{?_m\,x\,y\,\rho} P\,y \dashv$
  $\{?_T : \Gamma \vdash \text{relation Prop}, ?_m : \Gamma \vdash \text{Proper}\,(\text{eqA} ==> ?_T)\,P\}$

References


Sozeau, M. (2009). A new look at generalized rewriting in type theory. *J. Formaliz. Reason.*, *2*(1), 41–62. https://doi.org/10.6092/issn.1972-5787/1574


Sozeau, M., & Oury, N. (2008). First-class type classes. In O. A. Mohamed, C. A. Muñoz, & S. Tahar (Eds.), *Theorem proving in higher order logics, 21st international conference, tphols 2008, montreal, canada, august 18-21, 2008. proceedings* (pp. 278–293, Vol. 5170). Springer. https://doi.org/10.1007/978-3-540-71067-7_23