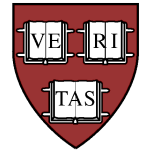
A New Look at Generalized Rewriting in Type Theory

Matthieu Sozeau

Harvard University

1st Coq Workshop
August 21th 2009
Munich, Germany



Generalized Rewriting

- **Equational reasoning** `x = y |- x + 1 ==> y + 1`
- **Logical reasoning** `x <-> y |- (x /\ y) ==> (x /\ x)`
- **Rewriting** `x > y |- x > z ==> y > z`
- **Abstract data types**

```
s, t : list, x =set y |- union x y =set x
==> union x x =set x
```

Rewriting in Type Theory

Moving from substitution to congruence.

- Built-in substitution: Leibniz equality.
  $\Pi\ A\ (P : A \to \texttt{Type})\ (x\ y : A),\ P\ x \to x = y \to P\ y.$
  - ✓ Applies to any context
  - ✗ Large proof term: repeats the context that depends on x
  - ✗ Restricted to equality, one rewrite at a time

Rewriting in Type Theory

Moving from substitution to congruence.

- Built-in substitution: Leibniz equality.
  $\Pi\ A\ (P : A \to \texttt{Type})\ (x\ y : A),\ P\ x \to x = y \to P\ y.$
  - ✓ Applies to any context
  - ✗ Large proof term: repeats the context that depends on x
  - ✗ Restricted to equality, one rewrite at a time
- Congruence.
  $\Pi\ A\ B\ (f : A \to B)\ (x\ y : A),\ x = y \to f\ x = f\ y$
  - ✗ Applies at the toplevel only
  - ✓ Small proof term: mentions the changed terms only
  - ✓ Generalizes to n-ary, parallel rewriting
  - ✗ Still restricted to equality

Rewriting in Type Theory

Moving from substitution to congruence.

- Built-in substitution: Leibniz equality.
  $\Pi\ A\ (P : A \to \texttt{Type})\ (x\ y : A),\ P\ x \to x = y \to P\ y.$
  - ✓ Applies to any context
  - ✗ Large proof term: repeats the context that depends on x
  - ✗ Restricted to equality, one rewrite at a time
- Congruence.
  $\Pi\ A\ B\ (f : A \to B)\ (x\ y : A),\ x = y \to f\ x = f\ y$
  - ✗ Applies at the toplevel only
  - ✓ Small proof term: mentions the changed terms only
  - ✓ Generalizes to n-ary, parallel rewriting
  - ✗ Still restricted to equality

  One can build a set of combinators to rewrite in depth: HOL conversions [Paulson 83], ELAN strategies.

Generalized Rewriting in Type Theory

D. Basin [NuPRL, 94], C. Sacerdoti Coen [Coq, 04]

- Generalized to any relation
  Proper (iff ++> iff) not $\triangleq \Pi\ P\ Q,\ P \leftrightarrow Q \rightarrow \neg\ P \leftrightarrow \neg\ Q$
- Multiple signatures for a given constant
  Proper (impl $\longrightarrow$ impl) not

Generalized Rewriting in Type Theory

D. Basin [NuPRL, 94], C. Sacerdoti Coen [Coq, 04]

- Generalized to any relation
  Proper (iff ++> iff) not $\triangleq \Pi\ P\ Q,\ P \leftrightarrow Q \rightarrow \neg\ P \leftrightarrow \neg\ Q$
- Multiple signatures for a given constant
  Proper (impl $\longrightarrow$ impl) not

Requires proof search:

- Heuristic in NuPRL based on subrelations (impl $\subset$ iff)
- Complete procedure in Coq.

Both are monolithic algorithms with a primitive notion of signature: a list of atomic relations (with variance).

A New Look

- Extensible signatures (shallow embedding)

  $\mathrm{all} : \forall\ A : \mathtt{Type},\ (A \to \mathtt{Prop}) \to \mathtt{Prop}$

  $\Pi\ A,\ \mathsf{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \Rightarrow \mathrm{iff})\ (@\mathrm{all}\ A)$

A New Look

- Extensible signatures (shallow embedding)

  $\mathrm{all} : \forall\ A : \mathtt{Type},\ (A \to \mathtt{Prop}) \to \mathtt{Prop}$

  $\Pi\ A,\ \mathsf{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \Rightarrow \mathrm{iff})\ (@\mathrm{all}\ A)$

- An algebraic presentation, supporting higher-order functions (rewriting under binders) and polymorphism:

  $\Pi\ A\ B\ C\ R_0\ R_1\ R_2,$
  $\quad \mathsf{Proper}\ ((R_1 \Rightarrow R_2) \Rightarrow (R_0 \Rightarrow R_1) \Rightarrow (R_0 \Rightarrow R_2))$
  $\quad\quad (@\mathrm{compose}\ A\ B\ C)$

A New Look

- Extensible signatures (shallow embedding)

  $\mathrm{all} : \forall\ A : \mathtt{Type},\ (A \to \mathtt{Prop}) \to \mathtt{Prop}$

  $\Pi\ A,\ \mathsf{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \Rightarrow \mathrm{iff})\ (@\mathrm{all}\ A)$

- An algebraic presentation, supporting higher-order functions (rewriting under binders) and polymorphism:

  $\Pi\ A\ B\ C\ R_0\ R_1\ R_2,$
  $\quad \mathsf{Proper}\ ((R_1 \Rightarrow R_2) \Rightarrow (R_0 \Rightarrow R_1) \Rightarrow (R_0 \Rightarrow R_2))$
  $\quad\quad (@\mathrm{compose}\ A\ B\ C)$

- Generic morphism declarations.

A New Look

- Extensible signatures (shallow embedding)

  $\mathrm{all} : \forall\ A : \mathtt{Type},\ (A \to \mathtt{Prop}) \to \mathtt{Prop}$

  $\Pi\ A,\ \mathsf{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \Rightarrow \mathrm{iff})\ (@\mathrm{all}\ A)$

- An algebraic presentation, supporting higher-order functions (rewriting under binders) and polymorphism:

  $\Pi\ A\ B\ C\ R_0\ R_1\ R_2,$
  $\quad \mathsf{Proper}\ ((R_1 \Rightarrow R_2) \Rightarrow (R_0 \Rightarrow R_1) \Rightarrow (R_0 \Rightarrow R_2))$
  $\quad\quad (@\mathrm{compose}\ A\ B\ C)$

- Generic morphism declarations.
- Support for subrelations, quotienting the signatures.

A New Look

- Extensible signatures (shallow embedding)

  $\mathrm{all} : \forall\ A : \mathtt{Type},\ (A \to \mathtt{Prop}) \to \mathtt{Prop}$

  $\Pi\ A,\ \mathrm{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \Rightarrow \mathrm{iff})\ (@\mathrm{all}\ A)$

- An algebraic presentation, supporting higher-order functions (rewriting under binders) and polymorphism:

  $\Pi\ A\ B\ C\ R_0\ R_1\ R_2,$
  $\mathrm{Proper}\ ((R_1 \Rightarrow R_2) \Rightarrow (R_0 \Rightarrow R_1) \Rightarrow (R_0 \Rightarrow R_2))$
  $(@\mathrm{compose}\ A\ B\ C)$

- Generic morphism declarations.
- Support for subrelations, quotienting the signatures.
- Rewriting on operators/functions, parallel rewrites...

Outline

Relation definitions

```
Definition relation (A : Type) : Type := A → A → Prop.

Definition inverse {A} (R : relation A) : relation A := flip R.

Notation " R ^-1 " := (inverse R) (at level 0).

Definition pointwise_relation {A B} (R : relation B) :
  relation (A → B) := λ f g, ∀ x : A, R (f x) (g x).
```

Relation Classes

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

Relation Classes

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

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

Standard Instances

```
Instance impl_refl : Reflexive impl.
Instance impl_trans : Transitive impl.

Instance iff_equiv : Equivalence iff.
Instance eq_equiv : Equivalence (@eq A).
```

Standard Instances

```
Instance impl_refl : Reflexive impl.
Instance impl_trans : Transitive impl.

Instance iff_equiv : Equivalence iff.
Instance eq_equiv : Equivalence (@eq A).

Instance inverse_refl `(Reflexive A R) : Reflexive R^-1.
```

Subrelations

```
Class subrelation {A : Type} (R R' : relation A) : Prop :=
  is_subrelation : Π x y, R x y → R' x y.

Instance subrelation_refl : @subrelation A R R.
```

Subrelations

```
Class subrelation {A : Type} (R R' : relation A) : Prop :=
  is_subrelation : Π x y, R x y → R' x y.

Instance subrelation_refl : @subrelation A R R.

Instance iff_impl_sub : subrelation iff impl.
Instance iff_inverse_impl_sub : subrelation iff impl^-1.
```

Morphisms

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

Instance reflexive_proper `(Reflexive A R) (x : A) : Proper R x.
```

Morphisms

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

Instance reflexive_proper `(Reflexive A R) (x : A) : Proper R x.

Definition respectful {A B : Type}
  (R : relation A) (R' : relation B) : relation (A → B) :=
  fun f g ⇒ ∀ x y, R x y → R' (f x) (g y).
```

Morphisms

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

Instance reflexive_proper `(Reflexive A R) (x : A) : Proper R x.

Definition respectful {A B : Type}
  (R : relation A) (R' : relation B) : relation (A → B) :=
  fun f g ⇒ ∀ x y, R x y → R' (f x) (g y).

Notation " R ++> R' " := (respectful R R') (right associativity).

Notation " R ⟶ R' " := (R^-1 ++> R') (right associativity).
```

Morphisms

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

Instance reflexive_proper `(Reflexive A R) (x : A) : Proper R x.

Definition respectful {A B : Type}
  (R : relation A) (R' : relation B) : relation (A → B) :=
  fun f g ⇒ ∀ x y, R x y → R' (f x) (g y).

Notation " R ++> R' " := (respectful R R') (right associativity).

Notation " R ⟶ R' " := (R^-1 ++> R') (right associativity).

Instance not_P : Proper (iff ++> iff) not.
```

Algorithm

Two phases:

1. Constraint generation in ML.
   Recursive descent on the term to find the redex, building a proof skeleton.
2. Constraint solving using type classes and $\mathcal{L}_{\mathsf{tac}}$.
   Depth-first search to solve the constraints with the declared hints.

Declarative presentation: unify and stop

$$\boxed{\Gamma \mid \psi \vdash \tau \rightsquigarrow_p^R \tau' \dashv \psi'}$$

Declarative presentation: unify and stop

$$\boxed{\Gamma \mid \psi \vdash \tau \rightsquigarrow_{p}^{R} \tau' \dashv \psi'}$$

UNIFY

$$\frac{\mathbf{unify}_{\rho}(\Gamma, \psi, t) \Uparrow \psi', \rho' : R\ t\ u}{\Gamma \mid \psi \vdash t \rightsquigarrow_{\rho'}^{R} u \dashv \psi'}$$

Declarative presentation: unify and stop

$$\boxed{\Gamma \mid \psi \vdash \tau \rightsquigarrow^{R}_{p} \tau' \dashv \psi'}$$

UNIFY

$$\frac{\mathbf{unify}_{\rho}(\Gamma, \psi, t) \Uparrow \psi', \rho' : R\ t\ u}{\Gamma \mid \psi \vdash t \rightsquigarrow^{R}_{\rho'} u \dashv \psi'}$$

ATOM

$$\frac{\begin{array}{l}\mathbf{unify}^{*}_{\rho}(\Gamma, \psi, t) \Downarrow \quad \tau \triangleq \mathbf{type}(\Gamma, \psi, t) \\ \psi' \triangleq \{?_S : \Gamma \vdash \text{relation } \tau, ?_m : \Gamma \vdash \text{Proper } \tau\ ?_S\ t\}\end{array}}{\Gamma \mid \psi \vdash t \rightsquigarrow^{?_S}_{?_m} t \dashv \psi \cup \psi'}$$

Declarative presentation: abstraction and application

$$\textsc{Lambda}\ \frac{\Gamma, x : \tau \mid \psi \vdash b \rightsquigarrow_p^S b' \dashv \psi' \qquad S' \triangleq \text{pointwise\_relation}\ \tau\ S}{\Gamma \mid \psi \vdash \lambda x : \tau . b \rightsquigarrow_{(\lambda x : \tau . p)}^{S'} \lambda x : \tau . b' \dashv \psi'}$$

Declarative presentation: abstraction and application

LAMBDA

$$\frac{\Gamma, x : \tau \mid \psi \vdash b \rightsquigarrow_p^S b' \dashv \psi' \qquad S' \triangleq \text{pointwise\_relation } \tau\ S}{\Gamma \mid \psi \vdash \lambda x : \tau . b \rightsquigarrow_{(\lambda x:\tau . p)}^{S'} \lambda x : \tau . b' \dashv \psi'}$$

APP

$$\frac{\Gamma \mid \psi \vdash f \rightsquigarrow_{p_f}^F f' \dashv \psi' \qquad \Gamma \mid \psi' \vdash e \rightsquigarrow_{p_e}^E e' \dashv \psi'' \qquad \textbf{type}(\Gamma, \psi, f)^{\uparrow} \equiv \tau \to \sigma \qquad \textbf{unify}(\Gamma, \psi'' \cup \{?_T : \Gamma \vdash \text{relation } \sigma\}, F, E \mathrel{+\!\!+\!\!>} ?_T) \Uparrow \psi'''}{\Gamma \mid \psi \vdash f\ e \rightsquigarrow_{(p_f\ e\ e'\ p_e)}^{?_T} f'\ e' \dashv \psi'''}$$

Declarative presentation: subrelations

$$\textsc{Sub}\quad \frac{\Gamma \mid \psi \vdash \tau \rightsquigarrow_p^S \tau' \dashv \psi' \quad \textbf{type}(\Gamma, \psi, \tau) \equiv \sigma \qquad \psi' \triangleq \{?_{S'} : \Gamma \vdash \text{relation } \sigma, ?_{sub} : \Gamma \vdash \text{subrelation } S\ ?_{S'}\}}{\Gamma \mid \psi \vdash \tau \rightsquigarrow_{(?_{sub}\ \tau\ \tau'\ p)}^{?_{S'}} \tau' \dashv \psi'}$$

Declarative presentation: arrows

$$\textsc{Pi}\quad \frac{\mathbf{unify}^*_\rho(\Gamma, \psi, \tau_1) \Downarrow \quad \Gamma \mid \psi \vdash \text{all } (\lambda x : \tau_1, \tau_2) \rightsquigarrow^S_p \text{all } (\lambda x : \tau_1, \tau_2') \dashv \psi'}{\Gamma \mid \psi \vdash \Pi x : \tau_1, \tau_2 \rightsquigarrow^S_p \Pi x : \tau_1, \tau_2' \dashv \psi'}$$

$$\textsc{Arrow}\quad \frac{\Gamma \mid \psi \vdash \text{impl } \tau_1\ \tau_2 \rightsquigarrow^S_p \text{impl } \tau_1'\ \tau_2' \dashv \psi'}{\Gamma \mid \psi \vdash \tau_1 \to \tau_2 \rightsquigarrow^S_p \tau_1' \to \tau_2' \dashv \psi'}$$

Implementation

- Make the rules syntax-directed by integrating the **Sub** rule in **App**. Requires transitivity of subrelation and some compatibility properties.

Implementation

- Make the rules syntax-directed by integrating the **Sub** rule in **App**. Requires transitivity of subrelation and some compatibility properties.
- Apply **Sub** at the top to force the output relation to be impl if rewriting in an hypothesis $H : P$, to get a proof of $P \to P'$ and refine H, or inverse impl if rewriting in the goal.

Implementation

- Make the rules syntax-directed by integrating the **Sub** rule in **App**. Requires transitivity of subrelation and some compatibility properties.
- Apply **Sub** at the top to force the output relation to be impl if rewriting in an hypothesis $H : P$, to get a proof of $P \to P'$ and refine H, or inverse impl if rewriting in the goal.
- Implemented as a set of combinators and higher-level strategies for building complex "conversions", e.g bottom-up parallel rewriting with a set of rewrite rules.

Constraint solving: type class search

- Depth-first search using Proper, subrelation instances and custom $\mathcal{L}_{\mathsf{tac}}$. Most specific constraints first.
- Backtracking proof-search with (green, safe) cuts.
- Discrimination nets for fast indexing with user control on the rigidity of constants.

Instances

```
Instance flip_P `(Proper (A → B → C) (RA ++> RB ++> RC) f)
  : Proper (RB ++> RA ++> RC) (flip f).
```

Instances

```
Instance flip_P `(Proper (A → B → C) (RA ++> RB ++> RC) f)
  : Proper (RB ++> RA ++> RC) (flip f).

Instance PER_P `(PER A R) : Proper (R ++> R ++> iff) R.
```

Higher-order morphisms

```
Inductive ex {A : Type} (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P.

Instance ex_iff_P A :
  Proper (pointwise_relation A iff ++> iff) (@ex A).
```

Higher-order morphisms

```
Inductive ex {A : Type} (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P.

Instance ex_iff_P A :
  Proper (pointwise_relation A iff ++> iff) (@ex A).
```

```
Goal Π A P Q, (∀ x : A, P x ↔ Q x) →
  (∃ x, ¬ P x) → (∃ x, ¬ Q x).
```

Higher-order morphisms

```
Inductive ex {A : Type} (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P.

Instance ex_iff_P A :
  Proper (pointwise_relation A iff ++> iff) (@ex A).
```

```
Goal Π A P Q, (∀ x : A, P x ↔ Q x) →
  (∃ x, ¬ P x) → (∃ x, ¬ Q x).
Proof. intros A P Q H HnP.
  setoid_rewrite ← H. exact HnP.
Qed.
```

Subrelations

Goes through products...

```
Instance respect_sub `(subrelation A R2 R1, subrelation B S1 S2) :
  subrelation (R1 ++> S1) (R2 ++> S2).
```

Subrelations

Goes through products...

```
Instance respect_sub `(subrelation A R2 R1, subrelation B S1 S2) :
  subrelation (R1 ++> S1) (R2 ++> S2).
```

The subsumption rule.

```
Lemma proper_sub_P `(Proper A R1 m, subrelation A R1 R2) :
  Proper R2 m.
```

Subrelations

Goes through products...

```
Instance respect_sub `(subrelation A R2 R1, subrelation B S1 S2) :
  subrelation (R1 ++> S1) (R2 ++> S2).
```

The subsumption rule.

```
Lemma proper_sub_P `(Proper A R1 m, subrelation A R1 R2) :
  Proper R2 m.

CoInductive apply_subrelation : Prop := do_subrelation.

Hint Extern 5 (Proper _ _) ⇒
  match goal with
    [ H : apply_subrelation ⊢ _ ] ⇒
    clear H ; apply @subrelation_proper
  end : typeclass_instances.
```

Dual morphisms

```
Instance inverse_P `(Proper A R m) : Proper (inverse R) m.
```

Dual morphisms

```
Instance inverse_P `(Proper A R m) : Proper (inverse R) m.

Class Normalizes A (m m' : relation A) : Prop :=
  normalizes : relation_equivalence m m'^-1.

Lemma inverse_arrow A B (R : relation A) (S : relation B) :
  relation_equivalence (R ++> S) (R^-1 ++> S^-1)^-1.
```

Dual morphisms

```
Instance inverse_P `(Proper A R m) : Proper (inverse R) m.

Class Normalizes A (m m' : relation A) : Prop :=
  normalizes : relation_equivalence m m'^-1.

Lemma inverse_arrow A B (R : relation A) (S : relation B) :
  relation_equivalence (R ++> S) (R^-1 ++> S^-1)^-1.

Lemma proper_normalizes_proper `(Normalizes A R0 R1)
  `(Proper A R1 m) : Proper R0 m.
```

Summary

- A modular, extensible tactic for generalized rewriting.
- Efficient proof search with cuts and indexing.
- Supports polymorphism, higher-order functions and rewriting on morphisms and under binders.
- A subrelation class that can handle dualization and user-defined relation hierarchies.

Current and Future work

- A set of strategies that can be combined to build efficient rewriting strategies: `autorewrite` done right!
- Handling dependent types properly (higher-order unification issues) and other constructs like pattern-matching.
- Automatic tactic to derive Proper instances.

The End