A New Look At Generalized Rewriting in Type Theory

Matthieu Sozeau

Harvard University

TYPES'09
May 13th 2009
Aussois, France

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)

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

  $\Pi\ A,\ \mathsf{Proper}\ (\mathrm{pointwise\_relation}\ A\ \mathrm{iff} \mathrel{+\!\!+\!\!>} \mathrm{iff})\ (@\mathrm{all}\ A)$

A new look

- Extensible signatures (shallow embedding)

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

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

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

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

A new look

- Extensible signatures (shallow embedding)

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

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

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

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

- Generic morphism declarations

A new look

- Extensible signatures (shallow embedding)

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

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

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

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

- Generic morphism declarations

- Support for subrelations, rewriting on operators:
  $\mathrm{relation\_equivalence}\ R\ (\mathtt{fun}\ x\ y \Rightarrow \mathsf{True}) \to \forall\ x\ y,\ R\ x\ y$

Outline

Relation classes

```
Definition relation (A : Type) : Type := A → A → Prop.
Notation inverse R := (flip (R : relation _) : relation _).
Definition pointwise_relation {A B} (R : relation B) :
  relation (A → B) := λ f g, ∀ x : A, R (f x) (g x).
```

Relation classes

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

Notation inverse R := (flip (R : relation _) : relation _).

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

Class Reflexive {A} (R : relation A) :=
  reflexivity : ∀ x, R x x.
Class Symmetric {A} (R : relation A) :=
  symmetry : ∀ {x y}, R x y → inverse R x y.
Class Transitive {A} (R : relation A) :=
  transitivity : ∀ {x y z}, R x y → R y z → R x z.
```

Relation classes

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

Notation inverse R := (flip (R : relation _) : relation _).

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

Class Reflexive {A} (R : relation A) :=
  reflexivity : ∀ x, R x x.
Class Symmetric {A} (R : relation A) :=
  symmetry : ∀ {x y}, R x y → inverse R x y.
Class Transitive {A} (R : relation A) :=
  transitivity : ∀ {x y z}, R x y → R y z → R x z.

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

Some instances

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

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

Some instances

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

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

Instance inverse_refl `(Reflexive A R) : Reflexive (inverse R).
Instance inverse_sym `(Symmetric A R) : Symmetric (inverse 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.
```

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 (inverse impl).
```

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

Program Instance respectful_per
  `(PER A R, PER B R') : PER (R ++> R').
```

Outline

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

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

Declarative presentation: unify and stop

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

UNIFY

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

ATOM

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

Declarative presentation: abstraction and application

$$\textsc{App}\quad \frac{\Gamma \mid \psi \vdash f \rightsquigarrow^{R \mathrel{+\!\!\triangleright} S}_{p_f} f' \dashv \psi' \qquad \Gamma \mid \psi' \vdash e \rightsquigarrow^{R}_{p_e} e' \dashv \psi''}{\Gamma \mid \psi \vdash f\ e \rightsquigarrow^{S}_{(p_f\ e\ e'\ p_e)} f'\ e' \dashv \psi''}$$

Declarative presentation: abstraction and application

$$\textsc{App}\quad \frac{\Gamma \mid \psi \vdash f \rightsquigarrow^{R \mapsto S}_{p_f} f' \dashv \psi' \qquad \Gamma \mid \psi' \vdash e \rightsquigarrow^{R}_{p_e} e' \dashv \psi''}{\Gamma \mid \psi \vdash f\ e \rightsquigarrow^{S}_{(p_f\ e\ e'\ p_e)} f'\ e' \dashv \psi''}$$

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

Declarative presentation: subrelations

$$\textsc{Sub}\quad \frac{\Gamma \mid \psi \vdash \tau \rightsquigarrow^{R}_{p} \tau' \dashv \psi' \qquad \psi'' \triangleq \{?_s : \Gamma \vdash \text{subrelation } R\ S\}}{\Gamma \mid \psi \vdash \tau \rightsquigarrow^{S}_{(?_s\ \tau\ \tau'\ p)} \tau' \dashv \psi' \cup \psi''}$$

Declarative presentation: arrows

$$\textsc{Pi}\quad \frac{\begin{array}{l}\textbf{unify}^*(\Gamma, \psi, \rho, \tau_1) \Downarrow \\ \Gamma \mid \psi \vdash \text{all}\ (\lambda x : \tau_1, \tau_2) \rightsquigarrow_p^R \text{all}\ (\lambda x : \tau_1, \tau_2') \dashv \psi'\end{array}}{\Gamma \mid \psi \vdash \Pi x : \tau_1, \tau_2 \rightsquigarrow_p^R \Pi x : \tau_1, \tau_2' \dashv \psi'}$$

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

Implementation

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

Implementation

- Make the rules syntax-directed by integrating the **Sub** rule in **App**. Requires transitivity of subrelation and some other 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 $\mathrm{inverse\ impl}$ if rewriting in the goal.

Implementation

- Make the rules syntax-directed by integrating the **Sub** rule in **App**. Requires transitivity of subrelation and some other 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 $\mathrm{inverse\ impl}$ 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

- Depth-first search using the Proper and subrelation instances and some $\mathcal{L}_{\mathsf{tac}}$ tactics.
- Uses a continuation-based backtracking monad with don't-care non-determinism, allowing safe cuts on proofs of ground goals for example.
- Discrimination nets are used for fast indexing with user control on the rigidity of introduced constants (never unfold inverse!).

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.
```

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.

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

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

Subrelations

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

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

Subrelations

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

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 (inverse m').

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

Instance normarrow `(Normalizes A R0 R1, Normalizes B U0 U1) :
  Normalizes (A → B) (R0 ++> U0) (R1 ++> U1) | 1.
Instance normatom A R : Normalizes A R (inverse R) | 2.
```

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 and produce efficient rewriting strategies: `autorewrite` done right!
- Handling dependent types: possible to write in signatures, but not usable during proof-search yet (higher-order unification issues).
- Automatic tactic to derive Proper instances.

The End