A New Look At Generalized Rewriting in Type Theory

MATTHIEU SOZEAU

Harvard University

TYPES’09
May 13th 2009
Aussois, France
Generalized Rewriting

- **Equational reasoning** \( x = y \mid - x + 1 \implies y + 1 \)
- **Logical reasoning** \( x \leftrightarrow y \mid - (x \land y) \implies (x \land x) \)
- **Rewriting** \( x > y \mid - x > z \implies y > z \)
- **Abstract data types**
  \[ s, t : \text{list}, x = \text{set} y \mid - \text{union} x y = \text{set} x \implies \text{union} x x = \text{set} x \]
Moving from substitution to congruence.

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

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

Moving from substitution to congruence.

- **Built-in substitution: Leibniz equality.**
  
  $$(\Pi A (P : A \to \text{Type}) \ (x \ y : A), \ P \ x \to x = y \to P \ y).$$

  - ✓ 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.**
  \[(\Pi \ A \ (P : A \rightarrow \text{Type}) \ (x \ y : A), \ P \ x \rightarrow x = y \rightarrow P \ y)\].

  ✓ Applies to any context
  ✗ Large proof term: repeats the context that depends on \(x\)
  ✗ Restricted to equality, one rewrite at a time

- **Congruence \(\Pi \ A \ B \ (f : A \rightarrow B) \ (x \ y : A), \ x = y \rightarrow f \ x = f \ y\)**

  ✗ 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
  \[ \text{Proper (iff } \leftrightarrow \text{ iff) not} \uplus \Pi P, Q, P \leftrightarrow Q \rightarrow \neg P \leftrightarrow \neg Q \]

- Multiple signatures for a given constant
  \[ \text{Proper (impl } \rightarrow \text{ impl) not} \]

\[ \begin{align*}
  \text{Proper (iff } \leftrightarrow \text{ iff) not} & \uplus \Pi P, Q, P \leftrightarrow Q \rightarrow \neg P \leftrightarrow \neg Q \\
  \text{Proper (impl } \rightarrow \text{ impl) not} &
\end{align*} \]
D. Basin [NuPRL, 94], C. Sacerdoti Coen [CoQ, 04]

- Generalized to any relation
  \[\text{Proper } (\text{iff } \leftrightarrow \text{iff}) \not\Delta \Pi P, Q, P \leftrightarrow Q \rightarrow \neg P \leftrightarrow \neg Q\]

- Multiple signatures for a given constant
  \[\text{Proper } (\text{impl } \rightarrow \text{impl}) \not\]

Requires proof search:

- Heuristic in NuPRL based on subrelations \((\text{impl} \subset \text{iff})\)

- 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)

  \[ \text{all} : \forall A : \text{Type}, (A \rightarrow \text{Prop}) \rightarrow \text{Prop} \]
  \[ \text{pointwise\_relation} : \forall A B, \text{relation} B \rightarrow \text{relation} (A \rightarrow B) \]
  \[ \Pi A, \text{Proper} \ (\text{pointwise\_relation} A \iff \leftrightarrow \iff) \ (@\text{all} A) \]
A new look

- Extensible signatures (shallow embedding)
  
  \[ \forall A : \text{Type}, \ (A \rightarrow \text{Prop}) \rightarrow \text{Prop} \]
  
  \[ \text{pointwise\_relation} : \forall \ A B, \ \text{relation} \ B \rightarrow \ \text{relation} \ (A \rightarrow B) \]
  
  \[ \Pi \ A, \ \text{Proper} \ \left( \text{pointwise\_relation} \ A \ \text{iff} \quad \leftrightarrow \quad \text{iff} \right) \ (@\text{all} \ A) \]

- An algebraic presentation, supporting higher-order functions (rewriting under binders) and polymorphism:
  
  \[ \Pi \ A \ B \ C \ R_0 \ R_1 \ R_2, \]
  
  \[ \text{Proper} \ \left( (R_1 \ \leftrightarrow \ R_2) \ \leftrightarrow \ (R_0 \ \leftrightarrow \ R_1) \ \leftrightarrow \ (R_0 \ \leftrightarrow \ R_2) \right) \]
  
  \[ (@\text{compose} \ A \ B \ C) \]
Extensible signatures (shallow embedding)

\[ \forall A : \text{Type}, (A \rightarrow \text{Prop}) \rightarrow \text{Prop} \]

pointwise_relation : \( \forall A B, \text{relation } B \rightarrow \text{relation } (A \rightarrow B) \)

\[ \Pi A, \text{Proper} \ (\text{pointwise_relation } A \iff \leftrightarrow \iff) \ (@\text{all } A) \]

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

\[ \Pi A B C R_0 R_1 R_2, \]

\[ \text{Proper } ((R_1 \leftrightarrow R_2) \leftrightarrow (R_0 \leftrightarrow R_1) \leftrightarrow (R_0 \leftrightarrow R_2)) \]

\[ (@\text{compose } A B C) \]

Generic morphism declarations
A new look

- Extensible signatures (shallow embedding)
  all : ∀ A : Type, (A → Prop) → Prop
  pointwise_relation : ∀ A B, relation B → relation (A → B)
  Π A, Proper (pointwise_relation A iff ++> iff) (@all A)

- An algebraic presentation, supporting higher-order functions (rewriting under binders) and polymorphism:
  Π A B C R₀ R₁ R₂,
  Proper ((R₁ ++> R₂) ++> (R₀ ++> R₁) ++> (R₀ ++> R₂))
  (@compose A B C)

- Generic morphism declarations

- Support for subrelations, rewriting on operators:
  relation_equivalence R (fun x y ⇒ True) → ∀ x y, R x y
Outline

1. Generalized Rewriting in Type Theory
2. Preliminaries on relations
3. Algorithm
4. Implementation
Definition \( \text{relation} \ (A : \text{Type}) : \text{Type} := A \to A \to \text{Prop} \).

Notation inverse \( R := \text{flip} ((R : \text{relation } \_ ) : \text{relation } \_ ) \).

Definition \( \text{pointwise\_relation} \ \{A B\} (R : \text{relation } B) : \text{relation } (A \to B) := \lambda f \ g, \forall x : A, R \ (f \ x) \ (g \ x) \).
Definition relation \((A : Type) : Type := A \rightarrow A \rightarrow \text{Prop}\).

Notation inverse \(R := (\text{flip} (R : \text{relation } _)) : \text{relation } _\).

Definition pointwise_relation \(\{A B\} (R : \text{relation } B) : \text{relation} (A \rightarrow B) := \lambda f g, \forall x : A, R (f x) (g x)\).

Class Reflexive \(\{A\} (R : \text{relation } A) := \)
  reflexivity : \(\forall x, R x x\).

Class Symmetric \(\{A\} (R : \text{relation } A) := \)
  symmetry : \(\forall \{x y\}, R x y \rightarrow \text{inverse} R x y\).

Class Transitive \(\{A\} (R : \text{relation } A) := \)
  transitivity : \(\forall \{x y z\}, R x y \rightarrow R y z \rightarrow R x z\).
Relation classes

**Definition** relation \((A : \text{Type}) : \text{Type} := A \to A \to \text{Prop} \).  

**Notation** inverse \(R := (\text{flip } (R : \text{relation } X) : \text{relation } X)\).  

**Definition** pointwise_relation \(\{A B\} (R : \text{relation } B) : \text{relation } (A \to B) := \lambda f g, \forall x : A, R (f x) (g x)\).  

**Class** Reflexive \(\{A\} (R : \text{relation } A) := \text{reflexivity} : \forall x, R x x\).  

**Class** Symmetric \(\{A\} (R : \text{relation } A) := \text{symmetry} : \forall \{x y\}, R x y \to \text{inverse } R x y\).  

**Class** Transitive \(\{A\} (R : \text{relation } A) := \text{transitivity} : \forall \{x y z\}, R x y \to R y z \to R x z\).  

**Class** Equivalence \(\{A\} (R : \text{relation } A) : \text{Prop} := \{ \text{Equivalence\_Reflexive } \Rightarrow \text{Reflexive } R ; \text{Equivalence\_Symmetric } \Rightarrow \text{Symmetric } R ; \text{Equivalence\_Transitive } \Rightarrow \text{Transitive } R \}\).
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).
Class subrelation \( \{ A : \text{Type} \} (R R' : \text{relation } A) : \text{Prop} := \)
\[
is\text{-subrelation} : \Pi x y, R x y \rightarrow R' x y.
\]
Instance subrelation_refl : @subrelation A R R.
Subrelations

Class subrelation \( \{ A : \text{Type} \} (R R' : \text{relation } A) : \text{Prop} := \)

\[ \text{is\_subrelation} : \Pi x y, R x y \rightarrow R' x y. \]

Instance subrelation\_refl : @subrelation A R R.

Instance iff\_impl\_sub : subrelation iff impl.
Instance iff\_inverse\_impl\_sub : subrelation iff (inverse impl).
Class Proper \( \{ A \} (R : \text{relation } A) (m : A) : \text{Prop} \) :=
\[
\text{proper} : R m m.
\]

Instance reflexive_proper ‘(Reflexive A R) (x : A) : Proper R x.
Class \( \text{Proper} \{A\} \ (R : \text{relation} \ A) \ (m : A) : \text{Prop} \) :=
\( \text{proper} : R \ m \ m. \)

**Instance** reflexive_proper ‘(Reflexive \( A \ R \)) \ (x : A) : \text{Proper} \ R \ x.

**Definition** respectful \{A B : \text{Type}\}
\( (R : \text{relation} \ A) \ (R' : \text{relation} \ B) : \text{relation} \ (A \to B) \) :=
\( \text{fun } f \ g \Rightarrow \forall x y, R x y \to R' (f x) \ (g y). \)
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).
Class Proper \( \{ A \} \) \(( R : \text{relation } A) \) \(( m : A) : \text{Prop} \) \(:= \text{proper} : R \ m \ m \).

Instance reflexive_proper ‘(Reflexive A R) \(( x : A) : \text{Proper} R \ x \).

Definition respectful \{ A B : \text{Type} \} \(( R : \text{relation } A) \) \(( R' : \text{relation } B) : \text{relation} (A \to B) := \text{fun } f \ g \Rightarrow \forall \ x \ y, R \ x \ y \to R' (f \ x) (g \ y) \).

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

Notation " R \ +\to \ R' " := (inverse R \ ++ \ R') (right associativity).

Program Instance respectful_per ‘(PER A R, PER B R') : PER (R \ ++ \ R').
Outline

1. Generalized Rewriting in Type Theory
2. Preliminaries on relations
3. Algorithm
4. Implementation
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 $\mathcal{L}_{\text{tac}}$.
   Depth-first search to solve the constraints with the declared hints.
Declarative presentation: unify and stop

\[ \Gamma \mid \psi \vdash \tau \sim_{\rho}^{R} \tau' \vdash \psi' \]
Declarative presentation: unify and stop

\[ \Gamma \mid \psi \vdash \tau \rightsquigarrow^R_p \tau' \vdash \psi' \]

\text{Unify}

\[
\text{unify}(\Gamma, \psi, \rho, t) \uparrow \psi', \rho' : R \ t \ u \implies \\
\Gamma \mid \psi \vdash t \rightsquigarrow^R \rho' \ u \vdash \psi'
\]
Declarative presentation: unify and stop

\[
\Gamma \mid \psi \vdash \tau \leadsto^R_{\rho} \tau' \vdash \psi'
\]

\textbf{Unify}

\[
\text{unify}(\Gamma, \psi, \rho, t) \uparrow \psi', \rho' : R \ t \ u
\]

\[
\Gamma \mid \psi \vdash t \leadsto^R_{\rho'} u \vdash \psi'
\]

\textbf{Atom}

\[
\text{unify}^*(\Gamma, \psi, \rho, t) \downarrow \tau \triangleq \text{type}(\Gamma, t)
\]

\[
\psi' \triangleq \{ ?_R : \Gamma \vdash \text{relation } \tau, ?_m : \Gamma \vdash \text{Proper } \tau ?_R t \}
\]

\[
\Gamma \mid \psi \vdash t \leadsto^?_{?_m} t \vdash \psi \cup \psi'
\]
Declarative presentation: abstraction and application

\[
\begin{align*}
\Gamma | \psi & \vdash f \rightsquigarrow_{R \leftrightarrow S}^{p_f} f' \iff \psi' \\
\Gamma | \psi' & \vdash e \rightsquigarrow_{R}^{p_e} e' \iff \psi'' \\
\Gamma | \psi & \vdash f \ e \rightsquigarrow_{S}^{(p_f \ e \ e' \ p_e)} f' \ e' \iff \psi''
\end{align*}
\]
Declarative presentation: abstraction and application

\textbf{APP}

\[ \Gamma \mid \psi \vdash f \sim_{p_f} R \rightarrow S f' \vdash \psi' \]
\[ \Gamma \mid \psi' \vdash e \sim_{p_e} R e' \vdash \psi'' \]

\[ \frac{\Gamma \mid \psi \vdash f \ e \sim_{S (p_f \ e \ e' \ p_e)} f' \ e' \vdash \psi''}{\Gamma \mid \psi \vdash \lambda x : \tau . b \sim_{R' (\lambda x : \tau . p)} \lambda x : \tau . b' \vdash \psi'} \]

\textbf{LAMBDA}

\[ \Gamma, x : \tau \mid \psi \vdash b \sim_{p} R b' \vdash \psi' \]

\[ R' \triangleq \text{pointwise\_relation} \ \tau \ R \]

\[ \frac{\Gamma \mid \psi \vdash \lambda x : \tau . b \sim_{R' (\lambda x : \tau . p)} \lambda x : \tau . b' \vdash \psi'}{\Gamma \mid \psi \vdash \lambda x : \tau . b \sim_{R' (\lambda x : \tau . p)} \lambda x : \tau . b' \vdash \psi'} \]
Declarative presentation: subrelations

\[
\begin{align*}
\Gamma | \psi \vdash \tau \overset{R}{\sim}^p \tau' \vdash \psi' \\
\psi'' \triangleq \{ ?_s : \Gamma \vdash \text{subrelation } R \ S \} \\
\hline
\Gamma | \psi \vdash \tau \overset{S}{\sim} (?_s \tau \tau' p) \tau' \vdash \psi' \cup \psi''
\end{align*}
\]
Declarative presentation: arrows

\[ \text{unify}^* (\Gamma, \psi, \rho, \tau_1) \Downarrow \]
\[ \Gamma \mid \psi \vdash \text{all} (\lambda x : \tau_1, \tau_2) \rightsquigarrow^R_p \text{all} (\lambda x : \tau_1, \tau'_2) \vdash \psi' \]
\[ \Gamma \mid \psi \vdash \Pi x : \tau_1, \tau_2 \rightsquigarrow^R_p \Pi x : \tau_1, \tau'_2 \vdash \psi' \]

\[ \text{Arrow} \]
\[ \Gamma \mid \psi \vdash \text{impl} \tau_1 \tau_2 \rightsquigarrow^R_p \text{impl} \tau'_1 \tau'_2 \vdash \psi' \]
\[ \Gamma \mid \psi \vdash \tau_1 \rightarrow \tau_2 \rightsquigarrow^R_p \tau'_1 \rightarrow \tau'_2 \vdash \psi' \]
Make the rules syntax-directed by integrating the \textbf{Sub} rule in \textbf{App}. Requires transitivity of \textit{subrelation} and some other properties.
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 $P \rightarrow P'$ and refine $H$, or inverse impl if rewriting in the goal.
- 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 $P \rightarrow P'$ 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

- Depth-first search using the `Proper` and `subrelation` instances and some \( \mathcal{L}_{\text{tac}} \) 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** \texttt{flip\_P} `(\text{Proper (A → B → C) (RA ↔ RB ↔ RC) f})` :
\text{Proper (RB ↔ RA ↔ RC) (flip f)}.

**Instance** \texttt{PER\_P} `(\text{PER A R}) : \text{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.
Instance respect_sub '(subrelation A R₂ R₁, subrelation B S₁ S₂) :
    subrelation (R₁ ↔ S₁) (R₂ ↔ S₂).
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.
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.
Instance `inverse_P `(Proper A R m) : Proper (inverse R) m.
Instance inverse\(_P\) ‘(Proper \(A\ R\ m\)) : Proper (inverse \(R\)) \(m\).

Class Normalizes \(A\) \((m\ m' : \text{relation} \ A) : \text{Prop} := \)

   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 (\(A \rightarrow B\)) (\(R0\ \leftrightarrow U0\)) (\(R1\ \leftrightarrow U1\) | 1.

Instance normatom \(A\ R\) : Normalizes \(A\ R\) (inverse \(R\)) | 2.
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