A New Look at Generalized Rewriting in Type Theory

Matthieu Sozeau

Harvard University
mattam@eecs.harvard.edu

Abstract. Rewriting is an essential tool for computer-based reasoning, both automated and assisted. It is so because rewriting is a general notion that permits to model a wide range of problems and provides means to effectively solve them. In a proof assistant, rewriting can be used to replace terms in arbitrary contexts, generalizing the usual equational reasoning to reasoning modulo arbitrary relations. This can be done provided the necessary proofs that functions appearing in goals are congruent with respect to specific relations. We present a new implementation of generalized rewriting in the Coq proof assistant, making essential use of the expressive power of dependent types and a recently implemented type class mechanism. The tactic improves on and generalizes previous versions by supporting natively higher-order functions, polymorphism and subrelations. The type class system inspired from Haskell provides a perfect interface between the user and such tactics, making them easily extensible.

1 Introduction

By generalized rewriting we mean the ability to replace a subterm t of an expression by another term t' when they are related by a relation R. When the relation is Leibniz equality, this reduces to standard equational reasoning which is allowed in essentially any context[1]. However, when the relation is different, the contexts in which a replacement can occur are restricted and one needs to prove a compatibility lemma about the specific context involved to show that the replacement is valid. Luckily, one can prove that rewriting in a context is allowed compositionally by combining compatibility lemmas for each constant.

By using an automatic tactic to find this proof, we can completely forget this boring part of reasoning and use generalized rewriting to support seemingly **extensional** reasoning with formulas, setoids and any other user-defined relations like enriched logical operators (e.g. separation logic connectives as in [1]).

We will first introduce and review related work in this area (§2) before presenting our new system (§3) and analysing it (§4), finally concluding in §5.

2 Related Work

Generalized rewriting is a notion that appears in many forms in the literature. For example, it is at the core of "window inferencing" systems like the one described in [2], that permits to prove goals by refinement steps each of which being an application of a lemma of the form $t\ \mathcal{R}\ t'$ or by recursively opening "windows", new subgoals that refine a given subterm of the current goal. We review the approaches to integrate this idea in type theory.

[1] Except if the rewrite involves capture of binders in an intensional type theory like Coq

LCF The idea of combining rewriting tactics appears in the Boyer-Moore and LCF systems, in particular in Lawrence Paulson's work [3] in Cambdrige LCF. He designs a set of so-called "conversions", higher-order rewriting tactics that can be used to implement custom rewriting strategies. He first focuses on the primitive rewriting tools of the system, β-reduction and Leibniz equality; then shows how to extend the technique to logical formulae using logical equivalence as the rewriting relation. In this system, it's still the user who combines the tactics himself to create a strategy.

NuPRL The step further was to automatically infer the combination of proofs needed to show that a rewriting is allowed, given compatibility lemmas on the constants involved. This was done by Basin [4] in NuPRL who also generalizes on the relations involved. He supposes given a set of lemmas showing the compatibility of operators with respect to some relations and combines them automatically to build the appropriate proof term when the user tries a rewrite step. In this setting, it is possible to give multiple *signatures* to a single constant, for example addition can be given the signatures:

$$+ : \{x = x' \to y = y' \to x + y = x' + y'\} \quad + : \{x < x' \to y \leq y' \to x + y < x' + y'\}$$
$$+ : \{x \leq x' \to y < y' \to x + y < x' + y'\} \quad + : \{x \leq x' \to y = y' \to x + y \leq x' + y'\}$$
$$+ : \{x = x' \to y \leq y' \to x + y \leq x' + y'\}$$

The first declares that addition is congruent for equality (actually, all objects are) and the later show that it is monotone for the various combinations of $=$, $<$ and $\leq$ on its arguments. The algorithm must sometimes choose one of these proofs during proof search, as the output relation is generally not known in advance, and the obvious combinatorial explosion in this setting led the author to find a heuristic for this choice and implement a partial search. This heuristic is based on user-provided information on the subrelation property of these relations: $=$ and $<$ are incomparable here, and both are stronger (smaller) that $\leq$. Choosing the strongest signature gives the best experimental results, hence choosing one of the first three signatures over the last three (implication is covariant for strongness on the right).

The implementation is based again on a set of tactics that are composed on-the-fly to produce a deterministic rewriting step that makes the goal progress.

Coq Finally, the `setoid_rewrite` tactic developed by Claudio Sacerdoti Coen [5] in Coq (after an initial implementation by Samuel Boutin already improved upon by Clément Renard) is slightly different. It differs from Basin's approach in a number of ways:

- The tactic is *complete*: instead of using a heuristic when multiple signatures can be selected, the algorithm tries all possibilities. The rationale for this choice is that goals are not deep enough in general to warrant a more efficient implementation that avoids the exponential factor. The tactic does not support subrelations hence it could not use Basin's heuristic.
- The tactic is *semi-reflexive*, which means it is separated in two parts, one meta part (written in ML) that builds a trace for the rewrite using a database of user lemmas and another part (in Coq) which proves a general theorem showing that any trace gives rise to a correct rewrite. The trace consists of the applied user lemmas along with information on variance.
- The tactic supports *variance* natively for asymmetric relations. Signatures are written point-free (without explicit mention of the objects) from the algebra (deeply embedded, as an inductive definition) of terms for atomic relations and the combinators $++\!\!>$, $\longrightarrow$, $\Longrightarrow$ for respectively covariant, contravariant and equivariant relations on arrow types. Symmetry is treated natively

(arguably for simplicity and user-friendliness) when using the contravariant and equivariant arrows, as each signature defines an opposite signature which has the same set of associated morphisms and one need to write only one of them. In comparison, our implementation does not treat the equivariant arrow but supports the automatic inference of opposite signatures (§3.4).
– The tactic also supports *non-reflexive* relations, generating subgoals for reflexivity on unchanged arguments when needed.

Sacerdoti Coen [5] indicates some possible optimizations on the proof search algorithm which is entangled with the recursive search for rewrites. However, in practice, it is not sufficient to speed up the trace creation process when the goal is very deep. This system was also somewhat limited due to the deep embedding in supporting polymorphic or dependent relations and functions.

Rewriting with Leibniz equality Finally, our work can be compared with the existing support for rewriting with the native equality of the system, for which every construction is congruent (except when capturing binders). That is, the standard rewrite works in contexts involving let-binders, pattern-matching or fixpoints and it allows substitution when type dependencies are involved, none of which is handled here. The current setup of the `rewrite` tactic is to use the standard rewrite when rewriting with a Leibniz equality, and use generalized rewriting for other relations.

3 A new tactic for Generalized Rewriting

We will present a new, generalized implementation of generalized rewriting in Coq that blends better with the system, directly supporting polymorphism, higher-order functions and rewriting under binders. Our algorithm is a mix of Basin's and Sacerdoti Coen's work.

We will split the problem in two parts to get a clearer view on the whole system: a constraint generation procedure (in ML) and a customizable proof search that is also at the meta-level (in $\mathcal{L}_{\text{tac}}$), based on type classes [6]. This simplification follows a current trend in the design of proof search algorithms (e.g. for type inference [7]) to make them more modular: it allows the study and more practically the independent modification of each part.

The resulting system allows to experiment more efficient proof-search strategies and supports all the previously-mentioned features, some of which are implemented solely using the extensibility capabilities of type classes. The tactic uses a set of general-purpose definitions on relations that we will present now.

3.1 Relations

We will begin by defining a number of standard concepts around relations and introduce a few useful type classes. We first introduce combinators on relations. We recall that in Coq a homogeneous binary relation R on a given type A is represented as a function of type $A \to A \to$ `Prop` into the propositions. The inverse (or converse) relation, noted R^{-1}, is easily obtained by flipping the order of arguments using the flip combinator:

```
Definition inverse {A} (R : relation A) : relation A := flip R.
Notation " R ^-1 " := (inverse R) (at level 2) : relation_scope.
```

The complementary relation is classicaly defined as a negation:

```
Definition complement {A} (R : relation A) : relation A := λ x y, R x y → False.
```

We can define the pointwise extension of a relation on B to $A \to B$:

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

We use the combinator all to represent universal quantification as a constant application.

```
Definition all {A} (P : A → Prop) : Prop := ∀ x : A, P x.
```

Properties We also introduce type classes that formalize the usual notions of reflexivity, symmetry, transitivity and their duals.

```
Class Reflexive {A} (R : relation A) := reflexivity : ∀ x, R x x.
Class Irreflexive {A} (R : relation A) := irreflexivity :> Reflexive (complement R).
Class Symmetric {A} (R : relation A) := symmetry : ∀ {x y}, R x y → R⁻¹ x y.
Class Transitive {A} (R : relation A) := transitivity : ∀ {x y z}, R x y → R y z → R x z.
```

These class declarations introduce overloaded methods that can be used to refer to arbitrary reflexivity, symmetry or transitivity proofs afterwards. Note that these classes are all indexed by a type and a value, making essential use of dependent types. See [6] for an introduction to type classes.

Standard Instances We can already populate the instance database with easy proofs by duality. All these properties are preserved by inversion, for example:

```
Instance flip_Reflexive '(Reflexive A R) : Reflexive R⁻¹ := reflexivity (R := R).
```

Finaly we define some instances for the standard logical operators. We use the Program extension [8] to define these instances. In this mode, each undefined field is turned into an obligation that is automatically proved using a default tactic including `firstorder` reasoning which is enough in this case. Implication impl = `fun A B : Prop ⇒ A → B` is reflexive and transitive:

```
Program Instance impl_Reflexive : Reflexive impl.
Program Instance impl_Transitive : Transitive impl.
```

Both logical equivalence (defined as double implication and denoted by iff or $\leftrightarrow$) and Leibniz equality form equivalences, so they both have Reflexive, Symmetric and Transitive instances.

Subrelations The last interesting concept we introduce is that of subrelations: the inclusion order on relations. We make it a class so that we can declare logic clauses to dynamically prove it on given relations.

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

An essential property of the subrelation relation is its reflexivity:

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

We declare the two following subrelation instances by default:

```
Instance iff_impl_subrelation : subrelation iff impl.
Instance iff_inverse_impl_subrelation : subrelation iff (inverse impl).
```

3.2 Signatures and morphisms

Contrary to Sacerdoti Coen's tactic, we chose a shallow embedding of signatures in the dependent type theory. This has the disadvantage that we cannot write algorithms on the signatures in Coq itself but we can always do so using the $\mathcal{L}_{\text{tac}}$ system. This choice also makes sense because the unification procedure that is needed later when trying to find constants having a given signature cannot be deeply embedded easily, nor is it really desirable for efficiency. Another obvious advantage is that one can design and support new constructions in signatures easily.

Morphisms The central notion of the tactic is that of being a morphism for a given relation R. We say that an object m is a morphism for a relation R when $R\ m\ m$, that is m is in the kernel of R, or m is a Proper element of R, using PER terminology[2]. Note that this definition is very general and not in any way specialized for functions, i.e. objects of arrow types which we will speak of as morphisms instead of proper elements, following the terminology used in previous work.

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

We make this notion a class, hence users can easily add new Proper instances to the type class database. We make Proper's type an implicit argument as it can always be infered from the signature R or the object itself.

Clearly, any element in a type accompanied by a reflexive relation is a proper element for it. We basically add a new logic clause for the Proper $R\ x$ theorem saying it is enough to find a proof of Reflexive $A\ R$ to solve it.

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

Signatures We declare a new parsing scope for relations seen as signatures so that the notations we use later can be given other meanings in different contexts.

```
Delimit Scope signature_scope with signature.
Open Local Scope signature_scope.
```

Another essential notion is the signature for objects with arrow types. We define a single compatibility arrow as a parametric extensionality relation on arrow types for two given relations on the input and output type.

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

Naturally, a function f respects respectful $R\ R'$ if for any two objects related by R the outputs of f applied to those are related by R'. The respectful definition gives a relational version of respect, which can be applied to two different functions, which will eventually be instantiated by the same object in a proper statement. As this is a shallow embedding, we won't be able to match on applications of respectful in Coq but we will be able to do so *via* tactics in $\mathcal{L}_{\text{tac}}$.

We obtain the other combinators simply using a set of notations. The respect arrows associate to the right following the arrow type. The notation ++> for covariance is the naked respectful definition, $R \longrightarrow R'$ is an abbreviation for R^{-1} ++> R'. The equivariant arrow $\Longrightarrow$ is currently an alias for the covariant arrow.

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

[2] The standard library of Coq 8.2 uses Morphism instead of Proper

`Notation` " R ⟶ R' " := (respectful R^{-1} R') (*right associativity*, `at` *level* 55) : *signature_scope*.

We can start declaring Proper instances using these notations for usual operators like logical negation not : `Prop` → `Prop`.

```
Program Instance contraposed_morphism : Proper (impl ⟶ impl) not.
Program Instance not_iff_morphism : Proper (iff ++> iff) not.
```

Unfolding the definitions of respectful, inverse and Proper, we find that the goals boil down to usual compatibility lemmas, e.g. for the first: ∀ x y, impl y x → impl (¬ x) (¬ y)

It is also possible to declare parametric instances, which act like Horn clauses in logic programming. Here we assert that every transitive relation is itself a morphism:

```
Program Instance trans_morphism `(Transitive A R) : Proper (R ⟶ R ++> impl) R.
```

The signature indicates that for every transitive relation R we have R x' x → R y y' → R x y → R x' y'. Using this morphism instance we will be able to rewrite with any transitive relation. Before going into a description of the set of Proper instances used to rewrite with the standard operators, we will present the algorithm that generates these Proper constraints.

3.3 Constraint Generation

The ML algorithm is in charge of finding the subterm to be replaced and generating a proof skeleton and a set of constraints. Once these are solved (if possible), we will simply substitute the proofs inside the skeleton to get a complete proof that the rewrite is valid.

We will present the algorithm as a set of inference rules from which we will derive a syntax-directed variant in a standard way. The algorithm is parameterized by a rewriting lemma ρ of type $\forall \overrightarrow{\phi}, R\ \overrightarrow{\alpha}\ t\ u$, i.e. a type whose ultimate codomain is an applied binary relation (note that any variable in $R\ \overrightarrow{\alpha}\ t\ u$ can be bound in $\overrightarrow{\phi}$ here). The typing context Γ, represents the set of local hypotheses and the global context, it grows when we go under abstractions. The set of constraints builds up incrementally in each rule, so there are both input and output sets denoted with ψ which contain constraints of the form $?_x : \tau$ declaring hypothetical objects of a given potentially incomplete type τ.

The rewriting judgment $\Gamma \mid \psi \vdash \tau \rightsquigarrow^{R}_{p} \tau' \dashv \psi'$ defined in figure 1 means that in environment Γ, ψ, τ is rewritten to τ' with respect to relation R with p a proof of type $R\ \tau\ \tau'$ in context Γ, ψ'.

Initially, given a goal $\Gamma \vdash \tau$ and a rewrite lemma ρ we will want to find a judgment of the form

$$\Gamma \mid \emptyset \vdash \tau \rightsquigarrow^{\text{impl}^{-1}}_{\_} \_ \dashv \_$$

Given a proof of such a rewrite from τ to some τ', that is a proof of τ impl^{-1} τ' we can apply it to the goal to progress to $\Gamma \vdash \tau'$. Dually, we use impl as the top relation when trying to rewrite in a hypothesis and specialize it with the resulting proof. *N.B.: We supposed that relations, hypotheses and goals were always in* `Prop`*, but the construction works just as well in* `Type`*, with computational relations.*

Rules The inference rules suppose as given a function **type**(Γ,ψ,t) which returns the type of a given term in a context. All our terms are well-typed so these calls can never fail. The unification function for a given lemma ρ $\mathbf{unify}_{\rho}(\Gamma, \psi, \tau)$ takes as input typing and constraint environments and a type and tries to unify the left-hand-side of the lemma's applied relation with the type. It

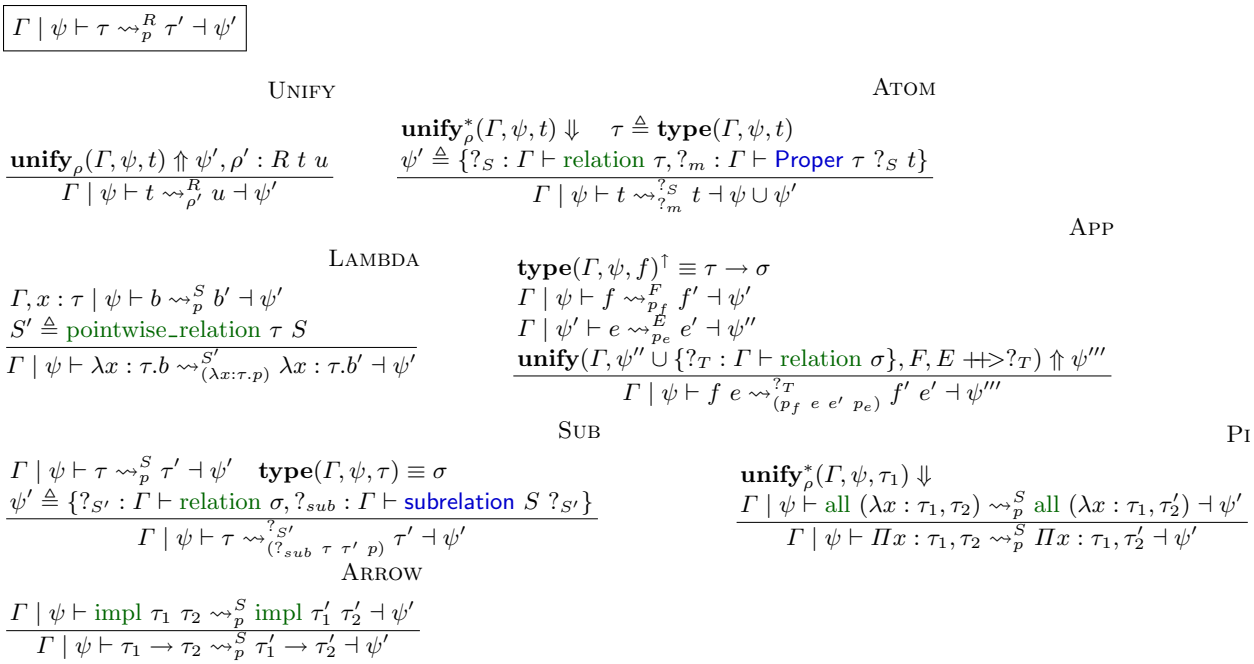

Fig. 1. Rewriting Constraint Generation - declarative version

may fail ($\Downarrow$) or succeed ($\Uparrow$), generating new constraints for the uninstantiated lemma arguments and an instantiated lemma ρ' whose type must be of the form $R\ t\ u$ for some R, u. The variant $\mathbf{unify}^*_\rho(\Gamma, \psi, \tau)$ tries unification on all subterms and succeeds if at least one unification does. The function $\mathbf{unify}(\Gamma, \psi, t, u)$ does a standard unification of t and u.

Let us now describe each rule:

- **Unify** The unification rule fires when the toplevel term unifies with the lemma. It directly uses the generated proof for the rewrite from t to some u with respect to some R.
- **Atom** This rule applies only when no other rule can apply and no rewrite can happen in the term. It asserts that the term must remain unchanged for some arbitrary relation $?_S$ during the rewrite, which is witnessed by a Proper proof. Typically, these constraints are either generated for unmodified arguments of a function and the Proper proof is solved by a reflexivity proof for the appropriate relation or they are generated for a function itself and the constraints get instantiated by user-provided proofs.
- **Lambda** To rewrite under an abstraction we simply rewrite the body inside the enriched context. The resulting proof can be extended pointwise to a closed proof in the original context by simply enclosing it in a λ.
- **App** We rewrite under an application by rewriting successively in the function and the argument. Here we assert that the rewrite relation for the function must unify with $E \mathrel{+\!\!+\!\!>} ?_T$ for some new relation $?_T$ to ensure that the constraint on the argument corresponds to the expected relation for the function argument. The resulting proof is a combination of the respectful proof for the function and the proof found for the argument which builds related results in $?_T$.
- **Sub** We add a subsumption rule to the system which allows to assign multiple relations to a single rewrite. The subrelation type class models a lattice of relations ordered by inclusion. It

allows for example to see that subrelation (pointwise_relation τ S) (eq_τ ++> S) and use a rewrite under an abstraction as a premise of the App rule.
– **Pi** This rule is an administrative step to rewrite inside the codomain of a dependent product, knowing that we can't rewrite in its domain. It translates the product into an application of the combinator all whose Proper instances will be presented later.
– **Arrow** The rewrite can happen in the domain only in non-dependent products. In this case we use the impl combinator instead.

Algorithm We must now derive an algorithm from this declarative specification of the system. To do so, we must eliminate the subsumption rule which is not syntax directed. The side conditions of the other rules are sufficient to ensure determinism otherwise. As the subrelation class is user-driven, we will only make assumptions on the associated set of instances. First, the relation must be transitive to be able to compress a stack of Sub applications into a single one. It must also be closed under pointwise_relation to go through the Lambda rule. The last problem is with App as it forces the relation on the function to be of a particular shape: we must simply change the rule to integrate Sub in the first premise:

$$\frac{\begin{array}{l}\Gamma \mid \psi \vdash f \rightsquigarrow^{F}_{p_f} f' \dashv \psi' \quad \mathbf{type}(\Gamma, \psi, f)^{\uparrow} \equiv \tau \to \sigma \\ \Gamma \mid \psi' \vdash e \rightsquigarrow^{E}_{p_e} e' \dashv \psi'' \\ \psi''' \triangleq \{?_T : \Gamma \vdash \mathrm{relation}\ \sigma, ?_{sub} : \Gamma \vdash \mathrm{subrelation}\ F\ (E \mathrel{++>} ?_T)\}\end{array}}{\Gamma \mid \psi \vdash f\ e \rightsquigarrow^{?_T}_{?_{sub}\ f\ f'\ p_f\ e\ e'\ p_e} f'\ e' \dashv \psi'' \cup \psi'''} \quad \textsc{AppSub}$$

Note that we explicitly take the head-normal form of the function's type to be able to generate the constraints and we assume that this arrow is not dependent.

With these changes, we can directly extract an algorithm $\mathbf{rew}(\Gamma, \rho, \tau)$ directed by the type τ, which always succeeds and returns a tuple (ψ, R, τ', p) with the output constraints, a relation R, a new term τ' and a proof $p : R\ \tau\ \tau'$. In case no rewrite happens, we will just have an application of Atom. Obviously, we can decorate the actual algorithm to count the number of successful unifications and fail if nothing was rewritten. We can use this to stop at the first rewrite too.

We now have the skeleton of a proof with holes and just need to solve the constraints to complete the proof. We assume here that we can mark constraints in the constraint set to indicate if they come from the unification of the lemma or as part of the algorithm itself. We will solve only the latter and leave the former for the user to prove.

3.4 Resolution

The proof-search problems generated by the **rew** algorithm are sets of constraints of the form Proper A (R_1 ++> $\cdots$ ++> R_n) m or subrelation A R_1 R_2, where A, m are closed terms but the R_i's are open. The existential variables appearing in them may of course be shared across multiple constraints, notably because of the App rule.

The R_i's may actually be arbitrary relations, and we want to be able to support some particular signature constructions automatically, notably the inverse combinator. We also need to actually implement a satisfying subrelation relation and support other features like higher-order morphisms and partial applications. To handle all these, we will write logic clauses that allow to prove Proper and subrelation as class instances. We will also extend the proof-search algorithm using a few $\mathcal{L}_{\mathrm{tac}}$ tactics to handle more complex resolution steps. All of this is defined in a standard Coq script that we present now.

As seen before, we can declare generic morphisms for standard polymorphic combinators that preserve compatibility, e.g. for flip:

```
Program Instance flip_proper ‘(mor : Proper (A → B → C) (RA ++> RB ++> RC) f) :
  Proper (RB ++> RA ++> RC) (flip f).
```

For higher-order morphisms, we must show how applications to pointwise equivalent functional arguments are related. For example, to show that existential quantification lifts logical equivalence we have to prove:

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

The statement of this proof unfolds to: $(\forall\ x,\ P\ x \leftrightarrow Q\ x) \rightarrow (\exists\ x,\ P\ x) \leftrightarrow (\exists\ x,\ Q\ x)$. Using this instance we can now rewrite under existentials with the equivalence relation, e.g:

```
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. assumption. Qed.
```

We won't detail here the various proper declarations for standard operators and classes like PER, Equivalence, etc... they can be found in the standard library modules. We will just detail the specific ones that allow to support some interesting features.

Partial applications During constraint generation we build signatures for applications starting at the first rewritten argument as we generate invariance constraints for the largest invariant subterms. This allows us to support parametric morphisms very easily as we are generally not interested in rewriting in the first few type parameters. However, this interacts with non-parametric morphisms as well. Suppose we have $P \rightarrow Q$ and rewrite with $H : Q \leftrightarrow Q'$. The generated constraint will be of the form Proper (iff ++> inverse impl) (impl P). However, we generally declare our morphisms for complete applications, e.g.: Proper (iff ++> iff ++> inverse impl) impl. Hence we need a way to derive the former from the latter. It suffices to declare the following instance whose application will generate a metavariable for the unknown relation on the argument.

```
Instance partial ‘(Proper (A → B) (R ++> R') m) ‘(Proper A R x) : Proper R' (m x) | 4.
```

We give a low priority to this instance so that it won't be used except if no other Proper is declared on $m\ x$.

Subrelations We have seen that logical equivalence is smaller than implication or its inverse. This means that any morphism that ends with iff can be viewed as a morphism producing impl-related arguments or its dual. We can actually make this even clearer by proving the subrelation instance for respect arrows. Classicaly, the arrow is contravariant for the subrelation relation on the left and covariant on the right:

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

We mentionned previously that for the constraint generation algorithm to be sound and complete with respect to the declarative presentation, subrelation had to be transitive. We will not add anything like a generic recursive transitive subrelation instance as that would render proof-search useless: it would always loop if we tried to search for an invalid subrelation constraint. Instead transitivity should be proved for the specific set of instances that are declared at some point. We

can assure that the set of instances declared in the library is transitive: no two rules could form the premises of a non-trivial use of transitivity.

We also mentionned that pointwise_relation had to be congruent for subrelation. Indeed it is a covariant morphism for it:

```
Instance pointwise_sub : Proper (subrelation ++> subrelation) (@pointwise_relation A B).
Instance subrelation_pointwise A `(sub : subrelation B R R') :
  subrelation (pointwise_relation A R) (pointwise_relation A R').
```

These instances allow us to bootstrap the system in a natural way: we can now rewrite inside signatures and under pointwise_relation, by showing that respectful is a morphism for subrelation as well. We can prove compatibility with subrelation and also relation_equivalence (defined as double inclusion of the relations and denoted by $=_{\mathcal{R}}$) of many of the combinators we have seen like inverse, complement and even Proper itself:

```
Instance morphism_proper A : Proper (relation_equivalence ++> @eq _ ++> iff) (@Proper A).
```

It follows that if we can find a Proper instance for m using signature R_2 and a subrelation R_1 of R_2 then m is also a proper element of it: this is exactly what is internalized by the Sub rule. However, we will not directly integrate the rule as it should only be applied once at the top of a search.

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

Indeed, this lemma is way too general to introduce it to the Proper instance search: it could be applied endlessly. Instead, we construct a tactic that restricts its use to the top of the goal when some flag apply_subrelation is set.

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

We add this tactic to the instance database to apply it when the goal is a Proper. Thanks to this control, we can do all the logic programming we want inside Coq using $\mathcal{L}_{\text{tac}}$ and let the user customize the proof search in the same way.

Dual Morphisms Finally, we can construct a tactic to handle the signatures involving inverse in the same way. First, we observe that a term m is a Proper element for a relation R^{-1} if and only if it is for R.

```
Program Instance proper_inverse `(Proper A R m) : Proper R^-1 m.
```

The goal is to make it possible for the user to declare a morphism for R only and automatically infer that it is also a morphism for R^{-1} or any relation equivalent to it with respect to the equational theory generated by:

```
Lemma inverse_invol A (R : relation A) : R^-1-1 =R R.
Lemma inverse_arrow A (R : relation A) B (R' : relation B) : (R ++> R')^-1 =R R^-1 ++> R'^-1.
```

Of course R^{-1} might not be in any kind of normal form: we want to push the inverse relation as far as possible inside the signature. Our strategy is to expand every part of a signature to applications of inverse and add subrelation instances to relate all the signatures in the produced equivalence class. We introduce a new class to normalize signatures, resolution will be based on the first one (m).

```
Class Normalizes {A : Type} (m m' : relation A) : Prop := normalizes : m =_R m'^-1.
```

Our strategy works by adding inverse everywhere in the signatures, going through arrows.

```
Lemma norm1 A R : @Normalizes A R (inverse R).
Lemma norm2 '(Normalizes A R0 R1, Normalizes B U0 U1) : Normalizes (R0 ++> U0) (R1 ++> U1).
```

We implement the strategy by a tactic that figures out if we have an arrow or an atomic type at the head and applies the appropriate lemma. Once we have resolved the inverse signature we can use subrelation to prove that the signature is related to the one declared by the user.

4 Analysis

4.1 Quantitative analysis

The constraint generation algorithm is clearly linear in the size of the rewritten term, so it has got a minor influence on the performance of the whole tactic. On the other hand, the proof search strategy is a (bounded) depth-first search using the instance database whose performance tends to be close to linear in the size of the constraints, when no backtracking is needed. We must always take care that the instances don't loop, that is why we control precisely the application of some lemmas.

The tactic has much better performance than the one by [5] on deep goals, as depth-first search often allows to prove goals directly without much backtracking. In practice the tactic gives immediate responses even on large goals. It should be noted that this tactic unlike the former only returns the first solution of the constraints. We leave the generalization of the search procedure for future work. Regarding the proof term size it is of the order of the rewritten term plus the proof terms for the constraints which are again generally linear in the size of their type.

4.2 Implementation & experiments

The tactic presented here is already available as part of Coq 8.2 where it replaces the previous one [9]. The implementation has been tested on the standard library of Coq as well as all the user contributions of Coq (http://coq.inria.fr/contribs-eng.html) which contains large projects using setoids such as CoRN and CoLoR. It is not clear whether the performance gains on these later examples come from the new `setoid_rewrite` implementation or some other improvement over previous versions of Coq but the standard library's times on setoid-intensive files have dropped significantly (−30%). Also, some other developments that could not be handled previously clearly benefit from the improved performance, e.g. the one done by Benton and Tabareau [1] which provided the impetus to reimplement the whole tactic.

To speed up proof search of instances, we use an enhanced discrimination net that can handle existentials contrary to the one already used in the `eauto` tactic of Coq. We also added a dependency analysis between subgoals to perform so-called green cuts in the search tree when two subgoals become independent (i.e. do not share existential variables).

4.3 Refinements

The tactic extends the previous one by supporting the `at` option which allows to select which occurences of the lemma should be rewritten, in a left-to-right traversal of the term. It should be

noted that the semantic of the tactic is different from the standard `rewrite`'s in that it tries to unify the lemma with each subterm independently and in its local context instead of doing a single unification and rewritting all subterms that match the resulting instantiated lemma. Typically our semantics allows to rewrite with a general lemma and select deep occurences in the goal without having to mention the term, e.g. consider:

$\texttt{Goal} \forall\ x\ y\ z : \texttt{nat}, (x + y) + z = y + (x + z).$

If we want to rewrite with the commutativity lemma for addition, we get four different possible instantiations that can be selected with `at`. This new semantic allows finer-grain control over occurences but it is also essential to be able to rewrite under binders, where unification can capture variables introduced inside subterms. Let's consider the following goal:

$\texttt{Goal} \forall\ H : (\forall\ x, x \times 1 = x), \exists\ x, x \times 1 \neq 0.$

To rewrite under the existential quantifier, we must apply H to x itself, hence do unification in the local context.

5 Conclusion

We have presented a new tactic for generalized rewriting in Coq, based on a constraint generation algorithm generating type class constraints to be solved by a generic but customizable instance search. The tactic extends previous ones in a number of directions, providing support for arbitrary polymorphic relations and morphisms, subrelations, automatic dualization of signatures and rewriting under binders. The new architecture allows for far greater extensibility *via* $\mathcal{L}_{\text{tac}}$ and for finer grain control on performance through its modular implementation. Finally, the choice of a shallow embedding and use of type classes allows easy integration inside user developments.

Acknowledgments I thank anonymous referees and Gregory Malecha for their remarks. I thank Nicolas Tabareau for helping in the design of the dualization instances, glueing to the previous interface and feedback and Arnaud Spiwack for initial discussions that led to the definitions of Proper and respectful.

References

1. Benton, N., Tabareau, N.: Compiling Functional Types to Relational Specifications for Low Level Imperative Code. In: TLDI. (2009)
2. Robinson, P.J., Staples, J.: Formalizing a Hierarchical Structure of Practical Mathematical Reasoning. Journal of Logic and Computation **3** (1993) 47–61
3. Paulson, L.C.: A Higher-Order Implementation of Rewriting. Science of Computer Programming **3** (1983) 119–149 (or 119–150??)
4. Basin, D.A.: Generalized Rewriting in Type Theory. Elektronische Informationsverarbeitung und Kybernetik **30** (1994) 249–259
5. Sacerdoti Coen, C.: A Semi-reflexive Tactic for (Sub-)Equational Reasoning. In Filliâtre, J.C., Paulin-Mohring, C., Werner, B., eds.: TYPES. Volume 3839 of Lecture Notes in Computer Science., Springer (2004) 98–114
6. Sozeau, M., Oury, N.: First-Class Type Classes. In Otmane Ait Mohamed, C.M., Tahar, S., eds.: Theorem Proving in Higher Order Logics, 21th International Conference. Volume 5170 of Lecture Notes in Computer Science., Springer (2008) 278–293
7. Pottier, F.: A versatile constraint-based type inference system. Nordic Journal of Computing **7** (2000) 312–347
8. Sozeau, M.: Un environnement pour la programmation avec types dépendants. PhD thesis, Université Paris 11, Orsay, France (2008)
9. Sozeau, M.: User defined equalities and relations. In: Coq 8.2 Reference Manual. INRIA TypiCal (2008)