A New Look at Generalized Rewriting in Type Theory

MATTHIEU SOZEAU
Harvard University

Rewriting is an essential tool for computer-based reasoning, both automated and assisted. This is because rewriting is a general notion that permits modeling a wide range of problems and provides a 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 the recently implemented type class mechanism. The new rewrite tactic improves on and generalizes previous versions by natively supporting higher-order functions, polymorphism and subrelations. The type class system inspired by Haskell provides a perfect interface between the user and the tactic, making it easily extensible.

1. INTRODUCTION

In this article, we will develop a new system for supporting generalized rewriting in type theory. 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 (or propositional equality in type theory jargon), this reduces to standard equational reasoning. In Coq for example, we can use the standard `rewrite` tactic to rewrite with an equation in a goal.

Suppose for example that we are proving the following lemma about natural numbers:

```
Goal ∀ x : nat, x + 1 = S x.
```

Addition being defined by recursion on its first argument, we have to use induction to prove it. The case $x = 0$ follows by `reflexivity`, the second goal looks like this after reduction:

```
x : nat
IHx : x + 1 = S x
============================
S (x + 1) = S (S x)
```

It is now possible to rewrite with the induction hypothesis using the tactic `rewrite` *IHx*, obtaining a goal `S (S x) = S (S x)` and finishing the proof using `reflexivity`.

This ability to replace equal terms anywhere is a basic feature of the propositional equality. However, when the considered relation is not Leibniz equality, we have no such high-level support to replace related terms in arbitrary goals. Generalized rewriting allows to do that, greatly simplifying the ubiquitous reasoning with relations in proofs. We will now introduce standard use cases of generalized

rewriting on equivalences like the regular equality (§1.1) but also orders or simply transitive rewrite relations (§1.2).

This article is an hyperlinked literate Coq script, pretty-printed using `coqdoc` with the following lexical conventions: `keywords` are in light red typewriter font, defined constants are in green serif font, inductive types in blue sans serif font, constructors in bordeaux sans serif and *variables* in magenta italic style.

1.1 Extensional notions of equivalence

The standard propositional equality is called intensional, because it relates only objects that are convertible in the system, i.e. reduce to the same normal form. It is a structural equality on terms, but in general one may want to consider objects related up to a larger relation.

Typically, when one works with non-canonical representations of structures, the standard equality is too coarse to capture the intended equivalence on objects. Consider for example representations of the rationals or implementations of sets as AVLs or unsorted lists with no duplicates. In these cases, the natural equivalence relation on the objects is often given by an observation on the datatype: we can compare non-canonical rationals by structurally comparing their canonical forms and consider two representations of sets equivalent if and only if they contain exactly the same elements up to a given equivalence on the elements type. We call these relations extensional, as they relate objects using some of their external properties.

Operations defined on these datatypes often do respect the equivalence relations. For example taking the inverse of two equivalent rationals will always give us equivalent rationals in return, but extracting the denominators will not necessarily give us equal numbers. When we reason on rationals or sets in contexts involving only equivalence preserving operations, we would like to get the same ease of reasoning as for the propositional equality.

For simplicity, we do not detail an implementation of a non-canonical datatype, instead we present an abstract interface for it. A real-world example of this can be found in Coq's finite sets library. Suppose we have a datatype *SET* and its associated equivalence relation *eqset*.

```
Parameter SET : Type.
Parameter eqset : relation SET.
```

We recall that a relation on a type A is modeled as a binary predicate of type $A \to A \to$ `Prop` in Coq. For example the trivial relation that relates everything can be defined as:

```
Definition true_relation (A : Type) : relation A := λ x y, True.
```

We use a standard set of type classes to declare that a relation is reflexive, symmetric or transitive. We also have packages to declare a combination of these properties giving preorders, strict orders, partial orders, partial equivalences and equivalences. We will describe these classes in detail in section 2.1. For now it is sufficient to know that this allows us to refer to proofs of these properties for an arbitrary relation using the overloaded tactics `reflexivity`, `symmetry` and `transitivity`, as long as the proofs were declared in the context.

```
Parameter eqset_equiv : Equivalence eqset.
```

An interface exposes the operations on the structure and lemmas about them, for example:

```
Parameter empty : SET.
Parameter union : SET → SET → SET.

Parameter union_empty : ∀ s : SET, eqset (union s empty) s.
Parameter union_idem : ∀ s : SET, eqset (union s s) s.
```

Naturally, we will also need compatibility lemmas showing that the *eqset* relation is preserved by the operations:

```
Parameter union_compat : ∀ s s' : SET, eqset s s' →
  ∀ t t' : SET, eqset t t' → eqset (union s t) (union s' t').
```

Now consider a proof on these sets using the facts that *empty* is neutral for *union* and that *union* is idempotent:

```
Goal ∀ s, eqset (union (union s empty) s) s.
Proof. intros s. transitivity (union s s);
  [ apply union_compat; [apply union_empty | reflexivity]
  | apply union_idem].
Qed.
```

The `intros` tactic here just introduces the *s* variable in the local context, leaving us with a goal *eqset* (*union* (*union s empty*) *s*) *s* to prove. Without any support for rewriting with user-defined relations, we have to manually apply transitivity of the *eqset* relation with *union s s*, apply the compatibility lemma and resolve the generated subgoals with the appropriate theorems. However, as we know that every operator is compatible with the *eqset* equivalence relation here, two simple rewrites with the *union_empty* and *union_idem* lemmas should suffice to prove the goal, the reasoning about compatibility being done automatically. This functionality is provided by the generalized rewriting tactic. One just has to register the compatibility lemmas using a special command:

```
Instance union_proper : Proper (eqset ++> eqset ++> eqset) union.
Proof. exact union_compat. Qed.
```

The union_proper proof has exactly the same content as the *union_compat* one, but its type is different. We use a concise notation for describing *signatures* of operators, that declare the compatibility of an object with certain relations. The arrows in the signature correspond to the arity of the function, here *union* : *SET* → *SET* → *SET*. The signature asserts that two calls to *union* made with pairwise *eqset* related arguments are always in the relation *eqset*.

The previous goal can then be proved using the higher-level `rewrite` tactic which applies reflexivity, transitivity and compatibility lemmas under the hood, building essentially the same proof term as above.

```
Goal ∀ s, eqset (union (union s empty) s) s.
Proof. intros s. rewrite union_empty, union_idem. reflexivity. Qed.
```

There is no magic here, if the user-provided compatibility lemmas were not sufficient to prove the compatibility conditions, the rewrite would fail.

1.1.1 *Logical equivalence.* A notoriously extensional relation in Coq is the logical equivalence relation.

Indeed, the logical equivalence relation iff is not defined as equality of propositions but as a double implication between them, that is:

```
Definition iff : relation Prop := fun A B ⇒ (A → B) ∧ (B → A).
```

The arrow here is the Coq function space. As we are in a constructive system, a proof of logical equivalence between A and B actually gives us a pair of functions from A to B and from B to A. The same idea of congruence applies here: as all the logical connectives $\wedge$, $\vee$, $\rightarrow$ are compatible with this equivalence, we can use an automatic procedure to verify that replacing a proposition by an equivalent one is valid in any goal built from these connectives:

```
Goal ∀ P Q : Prop, (P ↔ Q) → (Q → P) ∧ (Q → P).
Proof. intros P Q H. rewrite H. split; trivial. Qed.
```

The proof term corresponding to this proof will contain information to show that the goal context $(Q \rightarrow \_) \wedge (\_ \rightarrow Q)$ is congruent for iff-related propositions. This proof is automatically constructed from congruence lemmas on the conjunction and implication connectives.

This support is not restricted to the built-in propositions of type Prop, embedded logics can also profit from the system. A striking example is given by separation logic and its connectives. Formalizations in Coq [BT09, McC09] use the generalized rewriting framework to reason abstractly on separation logic formulae.

1.1.2 *Extensionality on functions.* Another problem with the intensional equality is that it does not give the classical principle of functional extensionality valid in set theory. This principle says that two functions are equal if they behave the same on every input. Leibniz equality does not capture that as it relates only functions which are syntactically the same. It is however impossible to distinguish two functions which behave the same in Coq, as the only way to compare them is to observe their behavior. Hence it is usually the case that a definition taking a function as argument is compatible with extensional equality on this argument, we just do not get it "for free" using the propositional equality.

The definition of extensional equality is tied to the notion of Leibniz equality. In general we may consider functions that are not extensionaly equal but pointwise equal for another relation. We define pointwise equivalence in Coq as a relation on functions that holds if the two functions build related arguments for every input.

```
Definition pointwise_relation A {B} (R : relation B) : relation (A → B) :=
  fun f g ⇒ ∀ a : A, R (f a) (g a).
```

Two functions are extensionally equal if they are pointwise Leibniz-equal:

```
Definition extensionality (A B : Type) : relation (A → B) :=
  pointwise_relation A eq.
```

An example use of this pointwise relation is given by the existential quantifier defined as:

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

The constructive existential is a pair of a witness x of type A and a proof of the proposition $P\ x$. If we have a proof of ex $A\ P$ and a predicate Q pointwise equivalent for iff to P, we can build a proof of ex $A\ Q$ by using the equivalence to build a proof of $Q\ x$ from the proof of $P\ x$. Using signature notation, we have:

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

We read this declaration as follows: if we apply @ex A to two pointwise equivalent predicates we get back two equivalent propositions. We use this compatibility lemma in the following proof, where the notation $\exists\ x : A, P$ is just a shorthand for ex (fun $x : A \Rightarrow P$).

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

When we rewrite in the goal $\exists\ x : A, \neg\ P\ x$ with our proof of $\forall\ x, P\ x \leftrightarrow Q\ x$, we actually go under the lambda abstraction fun $x : A \Rightarrow \neg\ P\ x$ to rewrite its body under the extended context $x : A$. We get back a new body $\neg\ Q\ x$ and a proof that it relates to $\neg\ P\ x$ by iff. From this rewriting proof we can build an extended one showing that fun $x : A \Rightarrow \neg\ P\ x$ and fun $x : A \Rightarrow \neg\ Q\ x$ are related by pointwise_relation A iff. As we know that existential quantification is compatible with this relation, we get a final proof that $\exists\ x : A, \neg\ P\ x$ is logically equivalent to $\exists\ x : A, \neg\ Q\ x$, which is enough to show that the rewrite is valid. The standard `rewrite` tactic does not go under binders like this for technical reasons, hence we use the variant `setoid_rewrite` instead.

Likewise, we can declare that the all combinator representing universal quantification as a constant application is compatible with logical equivalence.

```
Definition all {A} (P : A → Prop) : Prop := ∀ x : A, P x.
Instance all_proper A : Proper (pointwise_relation A iff ++> iff) (@all A).
```

Logical equivalence is not the only relation of interest when reasoning with propositions. It is also useful to show the compatibility of connectives with the more primitive logical implication, and rewrite with it. We can reify logical implication as a constant impl and declare how it interacts with universal quantification for example:

```
Definition impl (P Q : Prop) : Prop := P → Q.
Instance all_impl A : Proper (pointwise_relation A impl ++> impl) (@all A).
```

1.2 General relations

This naturally leads us to consider not only equivalence relations for rewriting but also general relations with which our definitions are compatible.

Logical implication is one example of a non-symmetric relation we can use for rewriting. Another good example in the context of proofs of programs is given by the subset order on sets. Suppose we have defined a subset order on our abstract datatype *SET* and shown that it forms a partial order w.r.t. *eqset*:

```
Parameter subset : relation SET.
Parameter subset_po : PreOrder subset.
Parameter subset_partial : PartialOrder eqset subset.
```

The PreOrder and PartialOrder parameters provide proofs that *eqset* is reflexive, transitive and antisymmetric w.r.t. *eqset*. We can declare that *union* is monotonic for *subset* in the following sense:

```
Instance: Proper (subset ++> subset ++> subset) union.
```

Now rewriting with *subset x y* hypotheses can be done in contexts involving union, e.g:

```
Goal ∀ s t, subset t empty → subset (union s t) s.
Proof. intros s t tempty. rewrite tempty, union_empty. reflexivity. Qed.
```

1.2.1 *Inverse relations.* As the *subset* relation is not symmetric, we may sometimes need to consider its inverse to write signatures. We can define the inverse of a relation R by flipping the arguments applied to R. We use the standard R^{-1} notation to denote it, so $R^{-1}\ x\ y$ is equal to $R\ y\ x$.

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

Take for example the set difference function *diff*, it should respect the following lemma and signature:

```
Parameter diff : SET → SET → SET.
Parameter diff_neutral : ∀ s, eqset (diff s empty) s.
Instance: Proper (subset ++> subset^-1 ++> subset) diff.
```

Unfolding the definitions, this means we have a proof of:

$$\forall\ s\ s', \mathit{subset}\ s\ s' \rightarrow \forall\ t\ t', \mathit{subset}\ t'\ t \rightarrow \mathit{subset}\ (\mathit{diff}\ s\ t)\ (\mathit{diff}\ s'\ t')$$

It models the fact that the second argument has to be larger or equal for the difference to be smaller or equal. As an abbreviation, we might also use a contravariant arrow $\longrightarrow$ to denote the same signature.

```
Instance: Proper (subset ++> subset ⟶ subset) diff.
```

Thanks to these declarations we can prove the following lemma using only rewriting.

```
Goal ∀ s t u, subset s t → subset empty u → subset (diff s u) t.
Proof. intros s t u st eu. rewrite ← eu, st, diff_neutral. reflexivity. Qed.
```

Note that we rewrite in the contravariant position of *diff* using $\leftarrow$ *eu*, so by an hypothesis of type equivalent to $\mathit{subset}^{-1}\ u\ \mathit{empty}$. This corresponds exactly to the relation in the signature.

1.2.2 *Relation inclusion.* Order relations and more generally antisymmetric relations are often related to other, larger relations in practice. Our system also has the ability to take this semantic relation inclusion information into account. For example, the user can declare the following inclusion:

```
Instance: subrelation eqset subset.
```

This declaration proves that any two sets related by *eqset* are also related by *subset*. It allows to factorize declarations of compatibility lemmas when one signature would subsume another. This allows us to rewrite with the *eqset* relation under *diff* even though *diff* has a single compatibility lemma about its behavior w.r.t. the *subset* relation:

```
Goal ∀ s t u, eqset s t → subset (diff s u) (diff t u).
Proof. intros. rewrite H. reflexivity. Qed.
```

We demonstrated how this system can be used to rewrite with orders, but it also works with any user-defined relation like partial equivalence relations that lack reflexivity or inductive relations modeling rewrite systems that lack symmetry and reflexivity. The only kind of relations that hardly make sense for rewriting are intransitive ones.

2. A NEW TACTIC FOR GENERALIZED REWRITING

We will present a new, generalized implementation of generalized rewriting in Coq that blends well with the system, directly supporting polymorphism, higher-order functions and rewriting under binders. Our algorithm is a mix of Basin's [Bas94] and Sacerdoti Coen's [SC04] work that we will present in section 4.

We will split the problem in two parts to get a clear view on the whole system: a constraint generation procedure (in ML, §2.3) and a customizable proof search that is also at the meta-level (in $\mathcal{L}_{\text{tac}}$), based on a set of type class instances [SO08] (§2.4). This simplification follows a current trend in the design of proof search algorithms (e.g. for type inference [Pot00]) to make them more modular: it allows the study and more practically the independent modification of each part.

The resulting system allows to experiment 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.

2.1 Relations

We will begin by defining a number of standard concepts around relations as type classes. We introduce classes that formalize the usual notions of reflexivity, symmetry and transitivity.

```
Class Reflexive {A} (R : relation A) := reflexivity : ∀ x, R x x.
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. A typeclass declaration is essentially a record declaration which registers each projection as a class method. Each method takes as argument an implementation of the class, but this argument is declared implicit and its implementation should be found by a typeclass resolution procedure. Take for example the Reflexive class, its `reflexivity` method has type: $\forall \{A : \texttt{Type}\} \{R : \text{relation } A\} \{R : \text{Reflexive } R\}, \forall x : A, R\ x\ x$. After applying `reflexivity` on a goal of the form $S\ x\ x$, unification leaves us with a constraint of the form Reflexive S. This constraint is solved by launching a proof search for an instance of Reflexive on S using a set of user-declared instances of Reflexive.

We can also package instances, for example to declare equivalences we use the class:

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

The proof search uses methods declared with :> like Equivalence_Reflexive to infer that a relation R declared as an Equivalence also declares a Reflexive R instance. This support allows us to write generic tactics that will work with any independently declared instances of the typeclasses, resolving implicitly the particular structures at hand when we apply them. See [SO08] for a detailed exposition of type classes.

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

```
Instance flip_Reflexive `(Reflexive A R) : Reflexive R^-1 := reflexivity (R := R).
```

Finally we define some instances for the standard logical operators. We use the Program extension [Soz08b] to define these instances. In this mode, each undefined field is turned into an obligation that is automatically proved using a default tactic. We use the `firstorder` tactic here to solve these simple goals. Implication is reflexive and transitive:

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

Both logical equivalence and Leibniz equality have Equivalence instances.

2.1.2 *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. Indeed typeclass resolution is essentially a prolog-like resolution on goals of the form $C\ t$ using the declared instances as clauses.

```
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 impl^-1.
```

2.2 Signatures and morphisms

The central notion of the system 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[1]. Note that this definition is very general and not in any way specialized for functions. We will

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

speak of objects of arrow types having Proper instances as morphisms, 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 m itself.

Clearly, any element in a type accompanied by a reflexive relation is a proper element for it. We add a new logic clause for the Proper R x goal 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.
```

2.2.1 *Signatures.* 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} (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, but will eventually be instantiated by the same object in a Proper proof. For example, to declare that logical negation is compatible with logical equivalence, we define:

```
Program Instance not_iff_morphism : Proper (respectful iff iff) not.
```

The content of not_iff_morphism is a proof of respectful iff iff not not, which unfolds to the expected compatibility statement:

```
∀ P Q, iff P Q → iff (not P) (not Q)
```

We chose a shallow embedding of signatures in the dependent type theory. This means that our signatures are written directly using the theory's term language and in particular its binding structure. This has the disadvantage that we cannot write algorithms on the signatures in Coq itself as we would have to match directly on the syntax of signatures, e.g. to decompose respectful applications. However we can always do so using the $\mathcal{L}_{\text{tac}}$ tactic language which allows to manipulate the syntax directly.

This choice 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.

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. The relation argument of Proper is parsed in this new scope.

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

We define respect arrows using a set of notations, they associate to the right following the arrow type. The notation ++> for covariance is the naked respectful

definition, $R \longrightarrow R'$ is an abbreviation for $R^{-1} \mathrel{++>} R'$. 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.
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.

```
Program Instance contraposed_morphism : Proper (impl --> impl) not.
```

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

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

The signature indicates that for every transitive relation R we have

$$R\ x'\ x \to R\ y\ y' \to R\ x\ y \to R\ x'\ y'$$

Using this morphism instance we will be able to rewrite with any declared transitive relation.

```
Goal ∀ `(Transitive A R) x y z, R x y → R y z → R x z.
Proof. intros A R T x y z H H0.
```

```
A : Type, R : relation A, T : Transitive R
x, y, z : A
H : R x y, H0 : R y z
============================
 R x z
```

```
  rewrite H.
```

```
A : Type, R : relation A, T : Transitive R
x, y, z : A
H : R x y, H0 : R y z
============================
R y z
```

```
  assumption.
Qed.
```

These generic lemmas serve as a basis to do rewriting with relations declared by the user without her having to redeclare structural lemmas that apply to every relation having some standard property like symmetry or transitivity. The standard library contains a set of standard Proper instances used to rewrite with the

predefined constants like iff, impl, etc... We will present them in more detail after describing the constraint generation algorithm which relies only on the small theory we presented here.

2.3 Constraint Generation

The rewriting tactic takes a rewriting lemma and a clause (the goal or an hypothesis) as arguments. It is then in charge to find a subterm unifiable with the left- or right-hand side of the lemma and generate a proof that the rewriting is correct. To do so, the ML algorithm generates a proof skeleton and a set of constraints (or existential variables) corresponding to the compatibility conditions necessary for the replacement to be correct. Once these are solved (if possible) using the type class instances defined in Coq, we get closed terms for all the existential variables in the constraint set. These are substituted in the skeleton to get a complete proof that the rewrite is valid.

The rewriting lemma parameter (denoted ρ) must be an object 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). We only consider left-to-right rewriting here, as we can always convert a rewrite from right to left to the other direction using inverse. We will present the constraint generation system as a set of inference rules from which we will derive a syntax-directed variant in a standard way. 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 $?_x : \Gamma \vdash \tau$ declaring hypothetical objects of a given type τ in context Γ. The type itself may also be open, i.e. contain existentials. As we go under abstractions, we must also extend a context Γ for the locally-bound variables.

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'. Operationally, the new term τ', the proof p and the constraints are outputs of the judgment determined by the input term τ and the rewriting lemma ρ. In this declarative presentation all the occurrences of the lemma are rewritten at once in parallel. We can easily refine the actual algorithm derived from it to fix an order of rewriting and select precisely the occurrences of interest to the user. We could also easily extend the algorithm to rewrite with multiple lemmas at once, but we leave this refinement for future work (§5.1)).

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

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

Once we have 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 of type τ and specialize it with the resulting proof of type $\tau \to \tau'$.

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.*

2.3.1 *Rules.* The inference rules rely on 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

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

UNIFY

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

ATOM

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

LAMBDA

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

APP

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

SUB

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

PI

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

ARROW

$$\frac{\Gamma \mid \psi \vdash \text{impl } \tau_1\ \tau_2 \rightsquigarrow_p^S \text{impl } \tau_1'\ \tau_2' \dashv \psi'}{\Gamma \mid \psi \vdash \tau_1 \to \tau_2 \rightsquigarrow_p^S \tau_1' \to \tau_2' \dashv \psi'}$$

Fig. 1. Rewriting Constraint Generation - declarative version

can never fail. We denote by $t^\uparrow$ the weak head normal form of a term t, again it is always defined as we use it only on well-typed terms. The unification function for a given lemma ρ is denoted $\mathbf{unify}_\rho(\Gamma, \psi, \tau)$. It takes as input a typing environment, a constraint set and a term and tries to unify the left-hand side of the lemma's applied relation with the term. It uses the same unification algorithm as the one used when applying a lemma during a proof. Unification may fail ($\Uparrow$) or succeed ($\Downarrow$), 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 p' for the rewrite from t to some u with respect to some R and returns the extended constraints Ψ'.

—**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 $?_T$.

—**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.
For example, when rewriting the x argument in $f\ x\ y$ using a proof p of $R\ x\ x'$ we will generate a constraint for a compatibility lemma for f of the form Proper $(R \mathrel{+\!\!+\!\!>} ?_S \mathrel{+\!\!+\!\!>} ?_{R'})\ f$. The proof of this lemma will be applied to x, x' and p witnessing the rewriting and then y, y and finally a proof that $?_S\ y\ y$ (or Proper $?_S\ y$), showing that the y argument is unchanged.

—**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.

—**Sub** We add a subsumption rule to the system which allows to assign multiple relations to a single rewrite. The subrelation type class models the lattice of relations ordered by inclusion. It allows for example to show that

$$\text{subrelation}\ (\text{pointwise\_relation}\ \tau\ S)\ (\text{eq}_\tau \mathrel{+\!\!+\!\!>} S)$$

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 were presented earlier.

—**Arrow** The rewrite can happen in the domain only in non-dependent products. In this case we use the impl combinator instead.

2.3.2 *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, and every test is decidable. 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:

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

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 derive an algorithm **rew**(Γ, ρ, τ) 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.

2.4 Resolution

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

The signature relations may be arbitrary, and we want to be able to support some particular signature constructions automatically, notably the respect arrows and 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 $\mathcal{L}_{\text{tac}}$ tactics to handle more complex resolution steps. All of this is defined in a standard Coq script that we will present now. Note that we have no claim of completeness on the set of instances presented here, we leave a detailed study of the unification theory of signatures for future work.

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

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

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 (see Coq.Classes.Morphisms). We will just detail the specific ones that allow to handle the standard constraints generated by the algorithm. The user can always add more signature constructs (e.g. to handle relation conjunction) that may need some special support during resolution.

2.4.1 *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 integrate this into our proof rules by defining the subrelation instance for respect arrows. As usual, 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 mentioned 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, hence any goal provable with the transitivity rule can be proved without it. When extending the subrelation instances, the user should make sure to only add instances that are recursive to preserve the transitivity property. This is trivially true for most user-defined subrelation lattices which are one level deep (e.g, subrelation eq *le* and subrelation *lt* *le*).

We also mentioned that pointwise_relation had to be congruent for subrelation. Indeed it is a covariant morphism for it, just like respectful (pointwise_relation *A* *R* is in fact equivalent to respectful eq *R*):

```
Instance: 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, as we know 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, and even Proper itself (see for example Coq.Classes.Morphisms_Relations):

```
Instance 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 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.

2.4.2 *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 $P \to Q$ 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 instance is declared on $m\ x$. The actual implementation is a bit more refined, allowing the user to declare the number of initial arguments of a constant that should not be considered for rewriting (which is impossible to infer in general). The tactic can then apply the partial instance repeatedly using this information to get a Proper goal that will be matched by the user Proper instances directly.

2.4.3 *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.
```

In practice, the user will have declared a morphism with a signature $R^{-1} \mathrel{++>} S$ and our goal will be to find a signature matching $R \mathrel{++>} ?T$. We first normalize the goal signature to $(R^{-1} \mathrel{++>} ?T')^{-1}$, with $?T := ?T'^{-1}$, apply the proper_inverse lemma and get a goal that is directly matched by the user declaration. We hence have found a instance for the signature $R \mathrel{++>} S^{-1}$.

To normalize the signature, we simply push the inverse relation through respect arrows using the inverse_arrow equivalence. This may create some R^{-1-1} relations that will be handled using the subrelation system using the inclusion subrelation $R^{-1-1}\ R$. We introduce a new class to normalize signatures, resolution will be based on the first one (m).

```
Class Normalizes {A} (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.

3. ANALYSIS

3.1 Quantitative analysis

The constraint generation algorithm is linear in the size of the rewritten term, simply folding through it, so it has a minor influence on the performance of the whole tactic. On the other hand, the proof search strategy is a depth-first search using the instance database whose complexity is potentially exponential in the size of the constraints. In practice however, only a few instances in the database apply to a given constraint and they generate "smaller" subgoals as we have a terminating set of clauses. For successful rewritings, the resolution is often linear in the size of the constraints types, when no backtracking is needed. This search strategy allows to get good performance even on deep goals. In practice the tactic gives immediate responses even on large goals. It should be noted that the tactic only returns the first solution of the constraints unlike the former one by [SC04]. We could easily adapt the implementation to get all results. Regarding the proof term size it is proportional to the rewritten term plus the proof terms for the constraints which are again generally linear in the size of their type. As we rewrite deeper we build more constraints and hence bigger proofs.

3.2 Implementation & experiments

The tactic presented here is already available as part of Coq 8.2 where it replaces the previous one [Soz08a]. 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 [BT09] 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).

3.3 Refinements

The tactic extends the previous one by supporting the `at` option which allows to select which occurrences 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 rewriting all subterms that match the resulting instantiated lemma. Typically our

semantics allows to rewrite with a general lemma and select deep occurrences 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 occurrences but it is also mandatory to be able to rewrite under binders, where unification can capture variables introduced inside subterms. Let's consider the following goal:

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

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

4. 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 [RS93], 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.

4.1 LCF

The idea of combining rewriting tactics appears in the Boyer-Moore and LCF systems, in particular in Lawrence Paulson's work [Pau83] in Cambridge 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 is still the user who combines the tactics himself to create a strategy.

4.2 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 [Bas94] 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:

$$
\begin{aligned}
&+ : \{x = x' \to y = y' \to x + y = x' + y'\} \\
&+ : \{x < x' \to y \leq y' \to x + y < x' + y'\} \\
&+ : \{x \leq x' \to y < y' \to x + y < x' + y'\} \\
&+ : \{x \leq x' \to y = y' \to x + y \leq x' + y'\} \\
&+ : \{x = x' \to y \leq y' \to x + y \leq x' + y'\}
\end{aligned}
$$

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, as is done by the subrelation class in our system: $=$ and $<$ are incomparable here, and both are stronger (smaller) than $\leq$. Choosing the "strongest" signature by considering only the output relation gives the best experimental results, hence choosing one of the first three signatures over the last two (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.

4.3 Coq

Finally, the `setoid_rewrite` tactic developed by Claudio Sacerdoti Coen [SC04] 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 (§2.4.3). It is also using a shallow embedding of the signatures, allowing the use of Coq's logic to extend the set of signature constructs easily.

—The tactic also supports *non-reflexive* relations, generating subgoals for reflexivity on unchanged arguments when needed.

Sacerdoti Coen [SC04] indicates some possible optimizations on the proof search algorithm which is entangled with the recursive search for rewrites. However, in

practice, they are 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 of signatures in supporting polymorphic or dependent relations and functions. Indeed one has to reify the universal quantifier to write signatures in the deep embedding, which is difficult to do in general.

Our implementation is a mix of Basin's and Sacerdoti Coen's, supporting subrelations and having completeness but not building the set of all possible solutions for the compatibility constraints. It is much easier to extend than the previous one and supports the use of polymorphism and dependent types directly, being entirely made up from primitive Coq definitions.

Our implementation is tied to the type classes feature of Coq and its dependent type theory, but the algorithm for generation of compatibility lemmas is relatively agnostic to the underlying theory. The system could hence be adapted with varying levels of difficulty to other proof assistants based on higher-order logic, like Isabelle or Matita. The latter would be an easy target due to its recent extension with unification hints [ARCT09] which can model a subset of the type classes mechanism, altough it currently lacks backtracking.

4.4 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. We plan to extend the support for rewriting with Leibniz equality using the generalized rewriting tactic in future work.

5. CONCLUSION

5.1 Future work

Besides basic improvements like allowing computable relations (whose sort is `Type` instead of `Prop`) and having a more efficient proof search, there are two important directions for future work on the tactic.

5.1.1 *Strategies.* Instead of implementing the tactic by a single monolithic algorithm that applies the rules as described before, we can split it into a set of combinators that can be composed to produce complex rewriting *strategies*, in a manner similar to [LV97]. The standard set of combinators would include:

—Applying a rule, or a set of rules at the toplevel of a term ;
—Applying a strategy on every direct subterm of a term ;
—Trying to apply a strategy and building a reflexivity proof if it fails or doesn't progress.
—Applying a sequence of strategies, choosing the first that works, etc...
—Building a recursive strategy as a fixed point.

From these we can build up higher-order tactics for applying a strategy top-down or bottom-up using the fixed point combinator to apply a strategy until it fails. This gives us the ability to apply the appropriate rewrite strategy given a term rewriting system expressed as a set of rewrite lemmas.

Practically, this allows us to build a more efficient variant of the `autorewrite` tactic. The current version of the tactic is very rudimentary, it simply tries to rewrite with a set of lemmas until no progress can be made. This process produces huge proof terms due to the numerous copies of the same unchanged subterms: each rewrite only changes a small part of the whole term but the proof term contains everything. We can instead do a single fold over the term applying rewrite rules in parallel to distinct, smaller subterms, resulting in a much smaller proof. It also results in a quicker proof search in the case of generalized rewriting as compatibility lemmas have to be searched for only once. Our initial experiments with such an implementation gives an order of magnitude improvement in space and time usage, even when rewriting with Leibniz equalities. We plan to pursue the implementation of this extension to give real control over the rewriting to the user.

5.1.2 *Treating Leibniz equality.* Our system is oblivious to the relations used for rewriting, hence it can also be used with the regular Leibniz equality. As every Coq function is a morphism for this equality, the constraints generated by such a rewrite will always be solved. However, we did not entirely profit from that fact in the tactic: there are some contexts in which we refuse to rewrite (pattern-matching scrutinees and branches, fixpoints and dependent arguments) even if they are valid contexts for rewriting using Leibniz equality. We can specialize the algorithm to generate the appropriate constraints and proofs in these cases to handle Leibniz equality more completely.

5.2 Summary

We have presented a new tactic for generalized rewriting in Coq. This tactic uses a constraint generation algorithm to generate type class constraints which are 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 simplifies integration inside user developments.

ACKNOWLEDGMENTS

I thank the anonymous referees for their useful comments on the presentation of this work.

References

[ARCT09] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In Berghofer et al. [BNUW09], pages 84–98.

[Bas94] David A. Basin. Generalized Rewriting in Type Theory. *Elektronische Informationsverarbeitung und Kybernetik*, 30(5/6):249–259, 1994.

[BNUW09] Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors. *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.

[BT09] Nick Benton and Nicolas Tabareau. Compiling Functional Types to Relational Specifications for Low Level Imperative Code. In *TLDI*, 2009.

[LV97] Sebastiaan P. Luttik and Eelco Visser. Specification of rewriting strategies. In *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97), Electronic Workshops in Computing*. Springer-Verlag, 1997.

[McC09] Andrew McCreight. Practical tactics for separation logic. In Berghofer et al. [BNUW09], pages 343–358.

[Pau83] Lawrence C. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149 (or 119–150??), 1983.

[Pot00] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[RS93] Peter J. Robinson and John Staples. Formalizing a Hierarchical Structure of Practical Mathematical Reasoning. *Journal of Logic and Computation*, 3(1):47–61, 1993.

[SC04] Claudio Sacerdoti Coen. A Semi-reflexive Tactic for (Sub-)Equational Reasoning. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2004.

[SO08] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In César Muñoz Otmane Ait Mohamed and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21th International Conference*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, August 2008.

[Soz08a] Matthieu Sozeau. *Coq 8.2 Reference Manual*, chapter User defined equalities and relations. INRIA TypiCal, 2008.

[Soz08b] Matthieu Sozeau. *Un environnement pour la programmation avec types dépendants*. PhD thesis, Université Paris 11, Orsay, France, December 2008.