A semi-reflexive tactic for (sub-)equational reasoning

Claudio Sacerdoti Coen

Project PCRI, CNRS, École Polytechnique, INRIA, Université Paris-Sud.
Claudio.Sacerdoti@inria.fr

Abstract. We propose a simple theory of monotone functions that is the basis for the implementation of a tactic that generalises one step conditional rewriting by "propagating" constraints of the form $x\ R\ y$ where the relation R can be weaker than an equivalence relation. The constraints can be propagated only in goals whose conclusion is a syntactic composition of n-ary functions that are monotone in each argument. The tactic has been implemented in the Coq system as a semi-reflexive tactic, which represents a novelty and an improvement over an earlier similar development for NuPRL.
A few interesting applications of the tactic are: reasoning in type theory about equivalence classes (by performing rewriting in well-defined goals); reasoning about reductions and properties preserved by reductions; reasoning about partial functions over equivalence classes (by performing rewriting in PERs); propagating inequalities by replacing a term with a smaller (greater) one in a given monotone context.

1 Introduction

Equalities are pervasive in mathematics. They are also very easy to reason about since they enjoy the *substitution property*: for each P, whenever $x = y$ and P holds for x, P also holds for y. In type theory, the substitution property is a trivial consequence of the definition of Leibniz equality.

Sometimes in mathematics it is necessary to consider equivalence relations that are not equalities. The usual mathematical practice is to immediately quotient the carrier w.r.t. the equivalence relation in order to use equality between classes and obtain substitutivity. Extensions of intensional type theories to accommodate quotient types have been proposed in the literature [7], but they are not used in practice. Sometimes the problem can be avoided by defining a normalisation function that picks a canonical representative for each class, replacing the equivalence relation with Leibniz equality over the representatives [3]. More often, however, the user avoids building the quotient sets and explicitly works with the equivalence relation. This approach is called the *setoid approach* [1].

The more serious drawback of working with setoids and equivalence relations is that the substitutivity property does not always hold for each context P. When it holds, the context is usually called *compatible* or *equality preserving* and the equivalence relation is said to be a *congruence* w.r.t. the context. When the relation is generalized to be an order, the contexts are usually called *order preserving* or *monotone* and no standard terminology exists when no assumption is made on the relation. Thus I prefer to call *substitutive* the context the substitutivity property holds for independently of the properties of the relations considered. Basin, in a related work [2], calls them *functional*.

In the setoid approach, to substitute y for x in $(P\ x)$ the user needs to prove that P is substitutive. In practice, much of this task can be easily automated [9]: since the composition of substitutive functions is substitutive, it is possible to build a tactic that automatically proves that a context is substitutive whenever the context is syntactically a composition of functions whose substitutive property has been previously shown by the user.

In this paper we provide a generalisation of the setoid approach by considering relations that are weaker than equivalence relations, and we characterise corresponding classes of substitutive contexts. As a result we develop a tactic that from the hypotheses $x\ R\ y$ and $P\ x$ deduces $P\ y$ whenever the context P, seen as a syntactic composition of functions, can be automatically proved to be substitutive.

Our idea is not original: the theory we develop is a reformulation of a restriction of Window Inferencing [6] and an equivalent tactic has been available for years in the NuPRL [2, 8] and HOL systems. However our implementation is the first one to be semi-reflexive and our formulation of the theory paves the way to reflexive implementations. Moreover we take non reflexive relations seriously.

In Sect. 2 we present a few scenarios that motivate the study of the substitutivity property for relations weaker than equivalence relations. Sect. 3 details the relation with previous work, in particular with Windows Inferencing. In Sect. 4 we present a small theory of monotone functions, called *morphisms*. Sect. 5 describes the semi-reflexive tactic that is based on that theory. The tactic has been already integrated in the code base of the proof assistant Coq, and it will be distributed as soon as the next major version of the system is released. To use it now, a user needs to switch to the development version of Coq, available only via anonymous CVS. A sample application of the tactic is given in Sect. 6. We draw a few conclusions in Sect. 7.

2 Motivations

We present now a few scenarios for equational and sub-equational reasoning.

2.1 Equivalence Relations

The classical scenario is the one of equivalence relations. Given the goal

$$\frac{H : x \equiv y}{(P\ (f\ x))}$$

the user wants to replace x with y in the conclusion, obtaining

$$\frac{H : x \equiv y}{(P\ (f\ y))}$$

The substitution can be performed iff $(P\ (f\ \Box))$ is a substitutive context. To prove that $(P\ (f\ \Box))$ is substitutive it is sufficient (but not necessary!) to prove that there exists an equivalence relation $\equiv'$ defined on the codomain of f such that: (1) f is substitutive w.r.t. $\equiv$ and $\equiv'$ (i.e. if $\forall x, x'\ x \equiv x' \Rightarrow (f\ x) \equiv' (f\ x')$) and (2) P is substitutive w.r.t. $\equiv'$ and $\iff$ (i.e. if $\forall x, x'\ x \equiv' x' \Rightarrow (P\ x) \iff (P\ x')$). The properties (1) and (2) are usually proved by the user once and for all when f and P are defined.

2.2 Dropping Symmetry: Preorders

The second scenario considers relations that are transitive and reflexive, but not symmetric. For instance, multi-steps reduction relations belong to this scenario. As an example, the user could be developing a theory of the λ-calculus, and she could face the goal

$$\frac{H : ((\lambda x.x)\ M) \rhd_\beta M}{(Q\ (f\ M))}$$

where $Q \circ f$ is a property preserved by reduction. Thus substituting $((\lambda x.x)\ M)$ in place of M in the goal is a sound operation that yields the new goal

$$\frac{H : ((\lambda x.x)\ M) \rhd_\beta M}{(Q\ (f\ ((\lambda x.x)\ M)))}$$

At first glance, the example given looks similar to the one in the previous scenario. However, the fact that the reduction relations are not symmetric introduce subtle consequences for substitutive contexts: a context C is still substitutive w.r.t. two reduction relations $\rhd$ and $\rhd'$ iff $\forall x, x'\ x \rhd x' \Rightarrow (C\ x) \rhd' (C\ x')$. This means that now C is substitutive iff it is a monotone *increasing* function. Now let $\Rightarrow$ be the reflexive, transitive and asymmetric relation considered on the set of all propositions. In order to substitute x with y in $(C\ x)$ it is necessary to prove that $(C\ y) \Rightarrow (C\ x)$ and, C being a substitutive context by hypothesis, this can be done iff $y \rhd x$. The conclusion is that only the right hand side of the relation can be replaced by the left hand side in the goal! In order to be able to substitute the right hand side for the left hand side the context C must be a monotone *decreasing* function.

From the previous observation we deduce that in the case of asymmetric relation the definition of substitutive context (an increasing vs a decreasing monotone function) depends on the direction chosen for the rewriting of the hypothesis.

As in the previous scenario, monotonicity is preserved by function composition. Moreover, it is easy to deduce whether the compound function is increasing or decreasing from the same knowledge on the arguments of the composition. Thus a tactic can automatically decide if a context is substitutive provided that the functions that syntactically compose the context have been already shown to be monotone.

A different example that belongs to the same scenario is the one of "rewriting" of large inequalities. For instance, given the goal

$$\frac{H : \epsilon \geq 0}{-2 * 0 \geq -2 * \epsilon}$$

the user can use the tactic to propagate the hypothesis H by replacing ϵ with 0 in the goal, obtaining:

$$\frac{H : \epsilon \geq 0}{-2 * 0 \geq -2 * 0}$$

From the usual arithmetic properties an automatic tactic should be able to deduce that the context $-2 * 0 \geq -2 * \Box$ is substitutive (i.e. it is a monotone decreasing predicate w.r.t. the order $\Rightarrow$).

2.3 Dropping Reflexivity: PERs

The third scenario that we consider is about irreflexive relations. For instance, irreflexive, symmetric and transitive relations (also called Partial Equivalence Relations or simply PERs) can be used in type theory to reason about functions over equivalence classes over a subset of a given set X: let $(X, \equiv)$ be a set together with a PER over it. We say that an element $x \in X$ is *proper* if $x \equiv x$. The subset of the proper elements of X forms a setoid. To represent a setoid-compatible unary function over the elements of the setoid we can use a function F whose domain is X such that F respects the relation $\equiv$ *on the proper elements only*. In other words, F is compatible iff $\forall x, x'\ x \equiv x \wedge x \equiv x' \Rightarrow (F\ x) = (F\ x')$. All the theorems that deal with F will have the hypothesis that all the elements of X are proper.

Alternative solutions to the PER approach (also called Partial Setoid Approach) is to first construct the subset of X as a sigma-type and then define an equivalence relation over the sigma-type elements. This alternative approach is strictly more expressive [1], but in practice it can be more cumbersome.

With respect to the previous scenarios the definition of substitutive context must not be changed. Moreover, the composition of substitutive functions (if the relation is symmetric) and the composition of strictly monotone functions (if the relation is asymmetric) preserves the property. However, the property is no longer granted for 0-ary functions (the constants that occur in the goal). An example should easily explain the remark. Consider the context $(f\ x\ \Box)$. We know by hypothesis that f is substitutive in each argument, i.e. $\forall x, x', y, y'\ x \equiv x' \wedge y \equiv y' \Rightarrow (f\ x\ y) \equiv (f\ x'\ y')$ and we want to prove that the whole context is substitutive. The proof is trivial provided that we can show that x is a proper element, i.e. $x \equiv x$. The latter proof cannot be fully automated. Thus we will leave it to the user.

3 Related work

(Sub-)equational reasoning is closely related to Window Inferencing (WI) [6], a style of reasoning proposed by Robinson and Staples [10]. Like natural deduction and sequent calculus, WI is a way of forming and validating proofs in a given logic. A proof is built by manipulating a stack of windows. Each window is a triple Γ, R, E where E is a formula in a context Γ and R is a binary relation over the type of E. The goal of the user is to transform the window Γ, R, E in a new window Γ, R, E' such that $\Gamma \Vdash ERE'$. The only actions the user can take are: 1) closing the window (equivalent to say that $E' = E$, requires ERE); 2) opening a new window Γ', R', E' where E' is a subterm of E such that for all E'', if $E'R'E''$ under the hypotheses Γ' then $ERE\{E''/E'\}$ under the hypotheses Γ; 3) transforming the window to Γ, R, E' in an atomic step (requires ERE'). In the initial proposal for WI all relations R in the windows needed to be equivalence relations; however it was soon observed that symmetry is not required.

WI is surely an interesting style for proof development, expecially for equational proofs, and it can be applied very naturally to some special domains such as program refinement. Moreover effective ad-hoc user interfaces have been developed for WI tools [4]. However, WI is a departure from more traditional proof styles such as natural deduction and sequent calculus and it is surely interesting to accomodate the basic ideas of WI in a more traditional setting.

In particular, we can consider a restriction of WI, equivalent to the original formulation, obtained by restricting the available actions to the sequence: 1) recursively open new windows as needed; 2) apply exactly one transformation; 3) recursively close all the opened windows. The sequence can be applied as many times as needed. Notice that this equivalent formulation does not require the relations to be transitive (since only one transformation can be applied before closing the windows): only the topmost relation must be transitive if the user wants to iterate the sequence more than once.

This alternative formulation can be better understood as a natural language deduction proof by *generalized rewriting*, where a term is substituted for an equivalent (but not necessaritly congruent) one. The terminoly "generalized rewriting" was introduced by Basin in [2]; in the rest of the paper we will adopt *equational reasoning*. To an application of the sequence corresponds the following proof by equational reasoning: the initial goal is $\Gamma \vdash ER?$ where ? is a metavariable to be instantiated during the proof search and $E = C_1[\ldots[C_n[E_{n+1}]]\ldots]$ where $E_i = C_i[\ldots[C_n[E_{n+1}]]\ldots]$ is the expression of the i-th opened window; in a context Γ_{n+1} it is possible to prove the lemma $E_{n+1}R'E'_{n+1}$; the proof proceeds by *rewriting* the lemma, replacing E_{n+1} with E'_{n+1} obtaining the new goal $\Gamma \vdash E'R?$ where $E' = C_1[\ldots[C_n[E'_{n+1}]]\ldots]$ that is proved by reflexivity of R instantiating ? with E'. To complete the proof a new goal $\Gamma_{i+1} \Rightarrow E_{i+1}R_{i+1}E_{i+1}\{E_{n+1}/E'_{n+1}\}$; $\Gamma_n \vdash E_iR_iE_i\{E_{n+1}/E'_{n+1}\}$ is opened for $i = 1, \ldots, n$ and it is solved (hopefully automatically) using the side condition for opening a new window in WI.

We can now see that a usual proof by rewriting is a special case of WI where all the relations are equivalence relations that are congruences w.r.t. the contexts $C_i[\_]$. In particular all the new goals generated by the rewriting step can be closed by using the fact that the appropriate relation is a congruence. Notice that, in this case, $\Gamma_i = \Gamma$ for each window.

Finally we define a proof by *sub-equational reasoning* as the generalization of a proof by equational reasoning where the relations are not required to be equivalence relations. Thus a proof by sub-equational reasoning is a special case of WI where each window opening does not change the context Γ, i.e. where $\Gamma_i = \Gamma$ for each window.

Observation 1 *To summarise the requirements over the relations in WI (the restricted formulation), every relation but that of the initial window does not need to be either symmetric or transitive. Reflexivity is also not required, but it is often necessary to prove the side conditions for opening a window. Thus the implementations of WI usually require the relations to be reflexive (or even pre-orders) [11].*

Observation 2 *(Sub-)equational reasoning is strictly less expressive than WI because of the restriction on the contexts. For instance, when proving $\Gamma \vdash (A \wedge B \to C) \iff ?$ in classical propositional calculus using WI it is possible to open a new window at A adding to Γ the two new hypotheses $\neg C$ and B and then exploiting them to prove $A \iff E$ for concluding $\Gamma \vdash (A \wedge B \to C) \iff (E \wedge B \to C)$. To prove the same result using (sub-)equational reasoning only the user needs first of all to prove the lemma $A \iff E$ in the context Γ, without the two additional hypotheses $\neg C$ and B.*

Although (sub-)equational reasoning is less expressive than WI, it is however a very useful generalization of rewriting and it has been exploited for years in the NuPRL system [2, 8] under the name of generalized rewriting. In this paper we present a tactic for (sub-)equational reasoning that is functionally equivalent to the tactic proposed in NuPRL. However, it differs from the NuPRL tactic in a few important aspects:

1) The NuPRL tactic is completely implemented at the meta-level. The meta level is responsible for finding a proof that the rewriting context is substitutive and generating the corresponding proof term. The latter is huge, its size being at least quadratic in the size of the initial expression. The well known problem, already existent for proofs based on simple rewriting, is that each rewriting step requires an explicit term that is the predicate obtained by abstracting the goal over the term to replace, the term to be replaced and the term to be substitued. Our implementation solves the problem by adopting a semi-reflexive approach. The meta level is still responsible for finding a proof that the rewrite context is substitutive, but instead of generating a proof term that details every step of the proof it generates a proof term that applies a correctness theorem to a trace of the proof found. The trace is roughly linear in the size of the initial expression, being an annotation of it (see Sect. 5 and 6). The correctness theorem says that from each well typed trace there exists a proof of the statement.

The semi-reflexive approach does not hinder flexibility in proof search: since the proof search is done at the meta level it is easy to implement backtracking and heuristics, and it is easy to invoke already existent tactics for proving side conditions. A completely reflexive approach (i.e. implementing proof search in

the logic) would be much more difficult to maintain and less computationally efficient. Moreover, the proof terms generated by (semi-)reflexive tactics are not only more compact, but they also have additional advantages for proof rendering purposes.
2) The tactic for NuPRL described by Basin [2] does not search the whole space for possible proofs of substitutivity, using an heuristic to avoid backtracking but possibly failing to find a proof. The motivation given for the heuristic is the complexity of the complete method, that is exponential in the depth of the rewritten subterms. However, since in practice the goal is small and not very deep, the exponential complexity is not really a problem and we decided to implement the complete method.

A stronger motivation for a simpler implementation is surely that the proof search is quite a complex operation that can be extremely difficult to debug (since a bug results in a complex not well typed proof term). This motivation becomes much less critical in our approach, since the proof term generated is just a trace of the proof — an annotation of the term to be rewritten — that can be inspected much more easily.
3) Finally in our theory we introduce the notion of *variance* for the arguments of a substitutive function. This notion is missing in the WI approach since it can be simulated by adding to the library of the system the inverse relation R^{-1} for each relation R. However, we argue that our approach is someway more natural since it fits better with the notion of monotone functions we are going to develop and for sure it avoids augmenting the library with redundant relations that should later on be considered equivalent during unification, proof search, etc. Just to make a simple example, our approach avoids 1) having to introduce reverse implication $A \Leftarrow B$ in the system; 2) proposing to the user confusing goals that mix the two implications, e.g. $\neg A \Leftarrow \neg B \Rightarrow B \Leftarrow A$; 3) introducing in the system automatic conversions to normalise the goals by expanding every definition of $\Leftarrow$ in a goal during unification and proof search.

4 Morphisms and Signatures

The scenarios provided in Sect. 2 should have convinced the reader that replacing one side of a relation with the other in a goal is a natural operation even when the relation is not an equivalence. To perform the replacement the system has to show that the replacement context is substitutive. This can easily be automated. However, we have seen that according to the properties of the relation the steps required to prove substitutivity slightly change. Thus the implementation of the tactic can become quite tricky in practice. In this section we will present a small theory of monotone functions that will be the basis for a semi-reflexive tactic implementation described in Sect. 5.

4.1 Basic definitions

We characterise n-ary functions that are simultaneously monotone in each argument not only by the types of their arguments and by their output type, but also by the relations associated to each type and by flags that state whether a function is increasing or decreasing in each argument:

Definition 1 (Signature). *A* signature *is an expression*

$$R_1 \Rightarrow^{d_1} \ldots \Rightarrow^{d_{n-1}} R_n \Rightarrow^{d_n} R$$

where $(T_1, R_1), \ldots, (T_n, R_n), (T, R)$ *are pairs whose first component is a type and whose second component is a binary relation*[1] *over it and* $d_1, \ldots, d_n \in \{\rhd, \lhd\}$.

Notice that each relation R in a signature uniquely identifies its carrier T.

Definition 2. *Let* R *be a binary relation.* $R^{\rhd} \stackrel{def}{=} R$ *and* $R^{\lhd} \stackrel{def}{=} R^{-1}$ *(i.e.* $R^{\lhd} \stackrel{def}{=} \lambda xy.yRx$*)*

Definition 3 (Morphism). *A* morphism *of signature*

$$R_1 \Rightarrow^{d_1} \ldots \Rightarrow^{d_{n-1}} R_n \Rightarrow^{d_n} R$$

is a function

$$f : T_1 \to \ldots \to T_n \to T$$

that is simultaneously monotone in its arguments:

$$\forall x_1 x_1', x_1 R_1^{d_1} x_1' \to \ldots \to \forall x_n x_n', x_n R_n^{d_n} x_n' \to (f\ x_1\ \ldots\ x_n) R (f\ x_1'\ \ldots\ x_n')$$

To state that a function f is a morphism of signature Σ we will write $f : \Sigma$.

Definition 4 (Covariance and contravariance). *Let*

$$f : R_1 \Rightarrow^{d_1} \ldots \Rightarrow^{d_{n-1}} R_n \Rightarrow^{d_n} R$$

We say that f *is* covariant *(respectively* contravariant*) in the* i*-th argument if* d_i *is equal to* $\rhd$ *(respectively* $\lhd$*).*

Fact 1 *If the* i*-th relation* R_i *of a signature* Σ *is symmetric, then every morphism of signature* Σ *has also signature* Σ' *where* Σ' *is obtained from* Σ *by changing the variance of the* i*-th argument.*

4.2 Morphism properties

Morphisms can be composed:

[1] We do not require the relations to be orders, even if we still adopt the corresponding terminology talking of monotone functions.

Fact 2 (Morphism composition) *Given*

$$f : R_1 \Rightarrow^{d_1} \ldots \Rightarrow^{d_{n-1}} R_n \Rightarrow^{d_n} R$$

and

$$f_i : R^i_1 \Rightarrow^{d^i_1} \ldots \Rightarrow^{d^i_{n_i-1}} R_{n_i} \Rightarrow^{d^i_{n_i}} R_i \textit{ for } i = 1, \ldots, n$$

the compound function

$$f \circ \langle f_1, \ldots, f_n \rangle$$

is a morphism of signature

$$\begin{array}{l} R^1_1 \Rightarrow^{d^1_1 = d_1} \ldots \Rightarrow^{d^1_{n_1-1} = d_1} R^1_{n_i} \Rightarrow^{d^1_{n_1} = d_1} \\ \quad \ldots \\ \qquad R^n_1 \Rightarrow^{d^n_1 = d_n} \ldots \Rightarrow^{d^n_{n_n-1} = d_n} R^n_{n_n} \Rightarrow^{d^n_{n_n} = d_n} R \end{array}$$

where $d = d'$ is $\triangleright$ when d and d' are the same variance, and $\triangleleft$ when they are not.

Let R be a binary relation over T. The following facts hold.

Fact 3 (Identity morphism) *The identity function $\lambda x : T.x$ is a morphism of signature $R \Rightarrow^{\triangleright} R$.*

Fact 4 (0-ary morphism) *The 0-ary constant function t of type T is a morphism of signature R if and only if tRt.*

Fact 5 *If R is a reflexive relation than every 0-ary function of type T is a morphism of signature R.*

Notice that if (T, R) is a partial setoid, then the 0-ary morphisms T are the proper elements of the setoid.

Fact 6 (Contraction (basic case)) *Given a morphism*

$$f : R \Rightarrow^{\triangleright} \ldots \Rightarrow^{\triangleright} R \Rightarrow^{\triangleleft} R \Rightarrow^{\triangleleft} \ldots R \Rightarrow^{\triangleleft} R'$$

the function $\lambda xy.(f\ x\ \ldots\ x\ y\ \ldots\ y)$ is a morphism of signature

$$R \Rightarrow^{\triangleright} R \Rightarrow^{\triangleleft} R'$$

The generalisation to an arbitrary interleaving of the covariant and contravariant arguments are left as a trivial exercise to the reader.

In the general case using contraction we can always obtain a new morphism equivalent to a given one such that each relation occurs in the signature at most twice, once in covariant and the other in contravariant position.

As a first step towards a semi-reflexive implementation of our tactic in the Coq system we have formalised in Coq the definition of signature and (an equivalent version of) all the previous facts. To obtain a non-reflexive implementation of the tactic such formalisation is not required.

In the formalisation we have heavily exploited dependent types and internal computation to make the formalisation simpler and more elegant. For example, it is possible to define in Coq two functions `signature_to_type` : `signature` $\rightarrow$ `Type` and `signature_to_compatibility` : $\Pi\Sigma$: `signature`, `signature_to_type` $\Sigma \rightarrow$ `Prop` that, given a signature

$$R_1 \Rightarrow^{d_1} \ldots \Rightarrow^{d_{n-1}} R_n \Rightarrow^{d_n} R$$

compute respectively the function type

$$f : T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T$$

and the proposition

$$\begin{array}{l}\lambda f : T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T, \\ \quad \forall x_1 x'_1, x_1 R_1^{d_1} x'_1 \rightarrow \ldots \rightarrow \forall x_n x'_n, x_n R_n^{d_n} x'_n \rightarrow (f\ x_1\ \ldots\ x_n) R (f\ x'_1\ \ldots\ x'_n)\end{array}$$

Another example consists in the function that automatically computes the signature of a morphism composition from the signatures of the arguments and the output type (and relation and variance) of the head function. In this way we reduce the problem of checking whether morphisms have a given signature or whether morphism can be composed to a type-checking problem, that is automatically decided by Coq. Moreover, we obtain proof terms that are relatively small, since we often provide just the signatures in place of long lists of constraints that must be satisfied for instance to compose morphisms.

The current formalisation in Coq is about 664 lines long and required just a few days, despite the fact that Coq automation vis a vis of dependent types (e.g. generation of elimination predicates and unification) is often unsatisfactory.

5 The Tactic

Let us focus now on the implementation of the tactic. We define an *applicative context* to be a term generated by the following grammar:

$$t \ ::= \ \Box \mid x \mid (f\ t\ \ldots\ t)$$

An applicative context is a term with a non-linear placeholder $\Box$ that is a syntactic composition of constants applications. The constants f can be n-ary or 0-ary. Unbound variables x can also occur in the applicative context (but they can not be applied to arguments).

We restrict ourselves to those applicative contexts C such that:

- all the constants have first order non-dependent types
- the constants are fully applied
- the term $\lambda x.C\{x/\Box\}$ is well-typed

Since we are avoiding dependent types we can also deal with function spaces and implications: a formula $A \to B$ can be seen as an applicative context $(\mathtt{Impl}\ A\ B)$ where `Impl` is defined as $\lambda AB.A \to B$.

We suppose that there exists a way for the user to specify for each constant f a set of signatures $\Sigma_1, \ldots, \Sigma_n$ such that f is a morphism of signature Σ_i for each i. Of course the user must provide a proof p_i of the fact that f is a signature of type Σ_i. Notice that the set of signatures is never empty, since every constant f is at least a morphism of signature $\Sigma_{\mathcal{L}}$ where $\Sigma_{\mathcal{L}}$ is the signature $=\Rightarrow^{\triangleright} \ldots \Rightarrow^{\triangleright} = \Rightarrow^{\triangleright} =$ where $=$ is Leibniz equality. Fact 1 can be used to reduce the number of registered signatures: if a morphism is both covariant and contravariant in an argument only one of the two signatures must be declared.

We also suppose that there exists a way for the user to register for each type a set of binary relations over it.

We can now describe the specification of our tactic:

Input 1 *A well-behaved propositional applicative context C, a relation R over T and a rewrite direction $d \in \{\triangleright, \triangleleft\}$.*

Output 1 *A proof of $\lambda x : T.C\{x/\square\}\ :\ R \Rightarrow^{d}$ `Impl` where `Impl` is propositional implication. The tactic fails if it is not able to produce such a proof.*

Before proceeding in describing the implementation, let us see how the tactic is concretely used. Facing the goal

$$\frac{H : E_1 R E_2}{G}$$

the user can ask to replace either E_1 with E_2 or E_2 with E_1 in G. In the first case our tactic is invoked with the arguments $C = G\{\square/E_1\}$, R and $\triangleleft$; in the second case with the arguments $C = G\{\square/E_2\}$, R and $\triangleright$. In both cases the proof built by the tactic can be applied to the hypothesis H to obtain the new goal $G\{E_2/E_1\}$ (respectively $G\{E_1/E_2\}$).

We describe now a naive implementation of the tactic based on the facts given in the previous section. We will then sketch the more efficient implementation that is now integrated in the Coq development release.

Step 1: the first steps output a set of applicative contexts annotated in each possible way according to the following rules:

- annotate each constant with one of its declared signatures; use fact 1 and the proof of symmetry of the appropriate relation to generate every correct signature w.r.t. variance constraints together with the proof that the constant respects the signature
- annotate each placeholder with the identity signature $R \Rightarrow^{\triangleright} R$. This corresponds to interpreting a placeholder as the identity morphism (cfr. fact 3)
- annotate each variable with the appropriate signature for a a 0-ary constant (cfr. fact 4); generate one annotation for each relation declared over the type of the variable; prove that the variable is a 0-ary morphism by

using fact 5 (if the relation is reflexive) or by asking the user to prove that the variable is a proper element for the relation

An annotated applicative context is a syntactic description of a (possibly wrongly typed) hereditary morphism composition where each placeholder is interpreted as the identity morphism (cfr. fact 3).

Step 2: prune out from the set those morphism compositions that are not well-typed according to the constraints given in fact 2 (i.e. the relation associated to each formal argument must match the output relation the actual argument is annotated with)

Step 3: prune out the morphisms whose conclusion is not the relation `Impl` (propositional implication)

Step 4: compute the signature of the obtained morphisms according to the rules given in fact 2

Step 5: apply contraction to each morphism in the set (fact 6); filter out those morphisms whose signature is not $R \Rightarrow^d$ `Impl`

Step 6: if the set is empty than fail; otherwise pick one morphism from the set and build a proof of the fact that the morphism has the expected signature by visiting the annotated applicative context and applying the appropriate fact for each node.

The previous implementation is obviously highly inefficient, since it builds a huge search space first and then prunes it. Better implementations can be obtained by constructing and pruning the search space at the same time. This is roughly the strategy we implemented in the Coq system[2]:

```
let rec decorate C R’ d’ =
  match C with
     [] -> if d=d’ && R = R’ then ok else error
   | x  -> if type x = carrier R’ && (reflexive R’ || ask-user R’ x)
           then ok else error
   | (f C) ->
       let sigset = signatures-of f in
       let sigset’ = filter (fun S -> output S = R’) sigset in
        exists sigset’ (fun S ->
          decorate C (relation (input S))
           (compose-dir R’ (variance (input S))))
```

The function `decorate` must be initially called on the context C, relation `Impl` and direction $\lhd$. Then it proceeds recursively over the applicative context remembering what is the expected output relation of the morphism C and whether the morphism C is supposed to be increasing or decreasing. The function `exists`

[2] To keep the code short only the case of unary morphisms is shown. The n-ary case is more complicated due to the need of backtracking the computation on all the arguments of the application whenever one branch fail. The actual code must also compute the well-typed annotated applicative context to produce the final proof. It is written in OCaml and it is 229 lines long.

performs the necessary backtracking over the set of signatures that have the expected output type. When a placeholder is met it is checked whether the expected relation and variance are coherent with the one the user is interested in. The function `ask-user` is supposed to open a new goal to ask the user to prove that its second argument is proper w.r.t. the relation that is the first argument.

We decided to implement the tactic in the Coq system in a semi-reflexive way (i.e. proof search is implemented at the meta-level; a trace of the proof found is reified at the term level and it is used by the type-checker of Coq to reconstruct the whole proof as a typing problem). The proof-search part (roughly the strategy described by the function `decorate`) is implemented in OCaml and it returns a decorated applicative context. Then the decorated applicative context is reified in a Coq term. Finally a Coq function checks that the decorated applicative context is well-typed, interprets it as a morphism and produces a proof that the morphism respects its signature. In Sect. 6 we present an example of usage of the tactic and we show the decorated applicative context and the proof term generated for that example.

Thanks to dependent types and internal computation in the Coq logic, checking that the decorated applicative context is well-typed boils down to a checking problem that is decided by the system. The construction of the proof is also done on-the-fly by the Coq system during type-checking. Thus the actual proof term generated by the tactic is just the application of a constant to the term that describes the decorated applicative context, that is linear in the size of the goal. If the tactic was written entirely in OCaml, instead, the generated proof goal would have been much bigger and definitely non linear in the size of the goal.

Implementing the tactic as a fully reflexive one would have meant implementing also the proof-search part in Coq. Since execution of code in Coq is made by an interpreter while OCaml code is compiled, this would have meant a much slower tactic.[3] Moreover, it is simpler to add heuristics and optimisations to the OCaml code without having to prove all of them correct: the Coq part of the tactic is responsible of verifying the correctness of the generated solutions and helps in detecting bugs in the OCaml code. Finally, when a 0-ary morphism over a non reflexive relation is found, our tactic needs to open a new goal. To the best of our knowledge, it is impossible for a fully reflexive implementation to open a new goal, whereas this is easy in the semi-reflexive setting: it is sufficient to reify a trace that contains fresh metavariables.

6 An example

The following example shows a situation where the tactic really shines.

Let $=^-$ be the smallest reflexive relation over negative integer numbers (i.e. $x =^- y$ iff $x = y$ and $x < 0$) and $=^+$ be the smallest reflexive relation over positive integer numbers. Suppose that the user has already proved and registered

[3] The next version of the Coq system will be equipped with a compiler [5] that should allow to implement complex proof search procedures as reflexive tactics.

the following signatures for less than, negation and multiplication:

signature:	corresponding compatibility statement:
$<: <\Rightarrow^{\triangleleft} <\Rightarrow^{\triangleright}$ `Impl`	$\forall x, x', y, y'\ x' < x \wedge y < y' \Rightarrow (x < y \Rightarrow x' < y')$
$- : <\Rightarrow^{\triangleleft} <$	$\forall x, x'\ x' < x \Rightarrow -x < -x'$
$* : <\Rightarrow^{\triangleleft} =^{-}\Rightarrow^{\triangleright} <$	$\forall x, x', y, y'\ x =^{-} x' \wedge y' < y \Rightarrow x * y < x * y'$
$* : <\Rightarrow^{\triangleright} =^{+}\Rightarrow^{\triangleright} <$	$\forall x, x', y, y'\ x =^{+} x' \wedge y < y' \Rightarrow x * y < x * y'$

Notice how the compatibility statements are usual preliminary statements found in every arithmetic library about integer numbers, up to the strange notation $x =^{-} y$ that must be unfolded to obtain $x < 0$ and $x = y$.

Notice also that the transitivity of $<$ — that is never used by the tactic — can be exploited to prove that $<$ is a morphism of signature $<\Rightarrow^{\triangleleft} <\Rightarrow^{\triangleright}$ `Impl`. Indeed this is automated in the system for each relation declared to be transitive.

Now consider the goal

$$\frac{\begin{array}{c} n, m, c_1, c_2 : Z \\ H : c_1 < c_2 \end{array}}{c_1 * n * m < -c_1 * n * m}$$

If the user asks to replace c_1 with c_2 by hypothesis H the tactic is automatically able to generate the new goal

$$\frac{\begin{array}{c} n, m, c_1, c_2 : Z \\ H : c_1 < c_2 \end{array}}{c_2 * n * m < -c_2 * n * m}$$

asking the user to prove either the first two following side conditions (i.e. both n and m are negative numbers) or the last two (i.e. both n and m are positive numbers):

$$\frac{n, m, c_1, c_2 : Z}{n =^{-} n} \quad \frac{n, m, c_1, c_2 : Z}{m =^{-} m} \qquad\qquad \frac{n, m, c_1, c_2 : Z}{n =^{+} n} \quad \frac{n, m, c_1, c_2 : Z}{m =^{+} m}$$

Suppose that the user chooses the first possibility (both n and m negative numbers). Then the part of the tactic implemented at the meta level — that implements the annotation function `declare` in Sect. 5 — produces the following annotated applicative context $\mathcal{C}$:

$$\begin{array}{ll} \mathcal{C} \stackrel{def}{=} (\underline{\leq} & (\underline{*}\ (\underline{*}\ \square^{<^{\triangleright}}\ n_{?_1}^{=^{-\triangleleft}})^{<^{\triangleleft}}\ m_{?_2}^{=^{-\triangleright}})^{<^{\triangleright}} \\ & (\underline{*}\ (\underline{*}\ (\underline{-}\ \square^{<^{\triangleright}})^{<^{\triangleleft}}\ n_{?_1}^{=^{-\triangleright}})^{<^{\triangleright}}\ m_{?_2}^{=^{-\triangleleft}})^{<^{\triangleleft}})^{\mathtt{Impl}^{\triangleleft}} \end{array}$$

that is just an annotation of $(<\ (*\ (*\ \square)\ n)\ m)\ (*\ (*\ (-\ \square)\ n)\ m))$ (the applicative context in prefix form obtained by abstracting the goal over c_1). The semantics of the annotations is described in Table 1. The annotation $\square^{<^{\triangleright}}$ says that $\square$ is a monotone increasing context (w.r.t. $<$), according to the user wish to replace the left hand side of the hypothesis H with its right hand side; thus

$\underline{f}$	every function f is annotated with (a reference to) the proof of its compatibility statement; i.e. $\underline{f} = (f, p)$ for some previously user-provided proof term p
$\square^{R^d}$	each placeholder is annotated with its relation and the rewrite direction chosen by the user
$x^{R^d}_{?_i}$	each constant is annotated with its relation R^d and a proof that xRx; a metavariable $?_i$ stands for the proof of a new goal that will be proposed to the user
$(\underline{f}\ C_1^{R_1^{d_1}}\ \ldots\ C_n^{R_n^{d_n}})^{R^d}$	every application is annotated with its relation R and a rewrite direction and it is well typed iff the relations and rewrite directions of its arguments and of the whole application are coherent with the signature of f (that must be $R_1 \Rightarrow^{d_1=d} \ldots R_n \Rightarrow^{d_n=d} R$)

Table 1. Annotated applicative contexts

$(* \ \square^{<^{\rhd}}\ n^{=^{-\lhd}})^{<^{\lhd}}$ says that $(* \ \square\ n)$ is a monotone decreasing context (since $n =^{-} n$); thus $(* \ (* \ \square\ n)^{<^{\lhd}}\ m^{=^{-\rhd}})^{<^{\rhd}}$ says that $(* \ (* \ \square\ n)\ m)$ is a monotone increasing context (since $m =^{-} m$); and so on until we find that the whole applicative context is monotone decreasing (w.r.t. logical implication).

The annotated context is reified as a CIC term and given as an argument to the correctness theorem, obtaining the following proof term t:

$$t \stackrel{def}{=} (\texttt{generalised\_rewrite\_ok}\ Z\ <^{\lhd}\ \mathcal{C}\ c_1\ c_2\ H)$$

Since the annotated context $\mathcal{C}$ is linear in the size of the goal, the proof term t is linear in the size of the goal plus the size of the type of the term to be rewritten (i.e. Z). The correctness theorem `generalized_rewrite_ok` has type

$$\begin{array}{l} \Pi T : \texttt{Type}.\Pi R : T \to T \to \texttt{Prop}.\Pi d.\Pi C : (\texttt{reified\_annotated\_context}\ T\ R\ d). \\ \quad \Pi c_1 c_2 : T.\Pi H : c_1\ R^d\ c_2. \\ \qquad (\texttt{signature\_to\_compatibility}\ (R \Rightarrow^d \texttt{Impl})\ (\texttt{deannotate\_context}\ T\ R\ d\ C)) \end{array}$$

where `reified_annotated_context` is the concrete data type for reified annotated contexts of the specified signature (i.e. an inductive type with one constructor for each line of Table 1) and `deannotate_context` returns the (deannotated) Coq term described by a reified annotated context. The kernel of Coq will type-check the reified annotated context and the proof term and it will infer for t the type

$$\forall c_1, c_2 : Z.c_1 < c_2 \Rightarrow c_2 * n * m < -c_2 * n * m \Rightarrow c_1 * n * m < -c_1 * n * m$$

that asserts that the current goal is substitutive for c_1. Notice also that since t is not a closed term (because the two meta-variables $?_1$ and $?_2$ occurs in C) Coq will accept the proof under the condition that the user will later on instantiate the two metavariables (i.e. she will close the two new sub-goals).

7 Conclusions

We described a small theory of monotone functions closed under composition. The theory can be used to study the properties of applicative contexts – a syntactically restricted term with non-linear occurrences of a placeholder. It leads to the implementation of a tactic that can semi-decide whether a given applicative context is a monotone function when interpreted as a function over the argument marked with a placeholder. The tactic exploits predefined proofs of monotonicity for each constant that are user-provided. The tactic can be applied to the problem of replacing one side of an hypothesis $E_1 R E_2$ with the other side in the current goal, where R is a general relation. The stronger the properties of the relations considered (i.e. reflexivity and symmetry), the fewer the user provided proofs required by the tactic.

The tactic has been implemented as a semi-reflexive one in the Coq proof assistant. It can be applied in several useful situations, such as rewriting in total and partial setoid theories; contraction or expansion of terms according to given reduction relation; propagation of large or strict arithmetical inequalities.

The actual status of the tactic is currently satisfactory. It has been used so far in a few test cases about reductions, greatly reducing the size of the proof script. As a tactic that supports the setoid approach both Lionel Mamane and Bas Spitters have used it in yet unpublished works, respectively about the formalization of surreal numbers and for handling implicit and non-computational arguments using monads. It is also being used by Marco Maggesi in the development of a theory of categories based on the partial setoid approach. Whether it can really be useful in the development of arithmetical libraries (using the approach given in Sect. 6) is still to be understood and currently under investigation.

The current implementation as a semi-reflexive tactic is also satisfactory due to the reasonably small size of the generated proof term. However, the tactic can be fully implemented in any other system for Higher Order Logic using the meta-language only. A fully reflexive implementation of the whole tactic seems difficult because of the need of opening new goals when the relations considered are not reflexive.

The only extension of the tactic and of the underlying theory that seems to be worth considering is that of *subrelations*. A relation R is a subrelation of a relation R' whenever xRy implies $xR'y$. Subrelations can be used as coercions by the "type-checking" algorithm for signatures. The interesting case is that of a morphism whose signature output is R found as an actual argument of a function that expects an argument whose output is R'. Instead of failing, the checker could just verify if R is registered as a subrelation of R', using the witness of the subrelation to build the proof of compatibility for the morphism.

In theory the declaration of subrelations does not add any expressive power to the tactic. In practice it could highly reduce the number of signatures to declare for a given constant. For instance, the "less than" and the equality relations can be both defined as subrelation of "less than or equal". Subrelations are implemented in the NuPRL system [8, 2].

To conclude, the proposed tactic can be an important building block for more complex tactics. Iterating a one step rewriting tactic and backtracking in case of failure yields a powerful simplification tactic for the Coq system, named *autorewrite*. Since autorewrite uses the rewriting tactic as a black box, it was easy to make autorewrite use our new tactic for performing rewriting of hypotheses that are not equalities. As an application, autorewrite is now able to reduce lambda-terms to their normal forms in a development of the theory of residuals for the lambda-calculus that we have used as a test case for the tactic. Once again, the reduction of the proof script size can be remarkable.

References

1. Gilles Barthe, Olivier Pons and Venanzio Capretta. "Setoids in type theory". In Journal of Functional Programming, 13(2), 261–293, 2003.
2. David Basin. "Generalized Rewriting in Type Theory". In Journal of Information Processing and Cybernetics, 30(5/6), 249–259, 1994.
3. Pierre Courtieu. "Normalized Types". In Proceedings of the Annual Conference of the European Association for Computer Science Logic, 554–569, 2001.
4. Christoph Lüth and Burkhart Wolff. "TAS — A Generic Window Inference System". In Theorem Proving in Higher Order Logics: 13th International Conference, LNCS 1869, 405–422, 2000.
5. Benjamin Grégoire. *Compilation de termes de preuves: un (nouveau) mariage entre Coq et OCaml*. PhD thesis. Université Paris 7. 2004.
6. Jim Grundy. 'Transformational hierarchical reasoning". In The Computer Journal, 39(4):291–302, 1996.
7. Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis. LFCS Edinburgh, 1995.
8. Christoph Kreitz. "The NuPRL 5 Manual". 135–145.
9. Clément Renard. *Memoire de DEA : Un peu d'extensionnalité en Coq*. INRIA Rocquencourt, 2001.
10. P. J. Robinson and J. Staples. "Formalizing a hierarchical structure of practical mathematical reasoning". Journal of Logic and Computation, 3, 47–61, 1993.
11. Mark Staples. "Window Inference in Isabelle". In Proceedings of the Isabelle Users Workshop, 18–19 September 1995, Cambridge (UK).