A Coq Library for Binary Logical Relations

Jérémie Koenig
Yale University
jeremie.koenig@yale.edu

Abstract

The generalized rewriting system implemented in Coq uses concepts and tools associated with binary logical relations. To this end the Coq standard library provides a rudimentary set of corresponding definitions and tactics. However, our experience suggests that the limitations of this rudimentary implementation restrict its practical applicability.

To address these limitations, I am reimplementing much of this infrastructure in a more general setting. The resulting library is employed in a module system for low-level certified code based on the language infrastructure in Compcert. It may offer a starting point for a more comprehensive treatment of binary logical relations in the Coq standard library. This is a work in progress available at `http://github.com/CertiKOS/coqrel`.

1. Introduction

Since their introduction to the study of programming languages, logical relations have found many applications. Binary logical relations in particular have been used to prove properties such as program equivalence, noninterference, and compiler correctness.

Basic principle The basic idea is as follows. Associate to each basic type a relation over its elements. Associate to each type constructor T a corresponding *relator*, which given a set of type parameters $\tau_1, \ldots, \tau_n$ and corresponding relations $R_1, \ldots R_n$ constructs a relation $T(R_1, \ldots, R_n)$ over $T(\tau_1, \ldots, \tau_2)$. This relator should be compatible with the introduction and elimination rules of T, so that terms built out of related components will themselves be related. This compositionality is the central feature of the logical relations method. Induction on the typing rules is often used to prove that all well-typed terms are related to themselves, by carefully chosen relations and relators which capture a property of interest.

For instance, for a particular application we may want to associate the relation $\leq$ to the type nat. The relator corresponding to the type constructor $\to$ is typically the following one. Given a relation R over the type A, and a relation S over the type B, we construct the relation $R \to S$ over the type $A \to B$ as follows:

$$f\,[R \to S]\,g \overset{\text{def.}}{\Leftrightarrow} \forall x\,y\,.\,x\,R\,y \Rightarrow f(x)\,S\,g(y)$$

In other words, two functions f and g are related if they take related arguments to related results. This definition is compositional in the sense that we can easily prove the following properties, which mirror the elimination and introduction rules for $\to$:

$$\frac{f\,[R \to S]\,g \qquad M\,[R]\,N}{f(M)\,[S]\,g(N)} \qquad \frac{x\,[R]\,y \vdash M\,[S]\,N}{(\lambda x\,.\,M)\,[R \to S]\,(\lambda y\,.\,N)}$$

Now consider the term $\mathsf{plus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$. The corresponding property of interest will be:

$$\mathsf{plus} \quad [(\leq) \to (\leq) \to (\leq)] \quad \mathsf{plus}, \tag{1}$$

which in this case asserts the monotonicity of addition of natural numbers. The fact that $\leq$ is reflexive can be interpreted to mean that all terms of type nat are well-behaved. Because of this, all functions built out of plus and terms of type nat will be monotonic — in the sense that they will satisfy similar properties derived from their types using $\leq$ and $\to$.

Generalized rewriting in Coq The generalized rewriting system [3, 4] implemented in recent versions of Coq follows this approach to extend the `rewrite` tactic to a wide range of user-defined relations. To get a sense of the way this works, suppose we want to use a hypothesis $H : x \leq 5$ for rewriting in the goal:

$$\mathsf{plus}(\mathsf{plus}(x, 2), y) \leq 15$$

The whole process boils down to proving the following lemma:

$$\mathsf{plus}(\mathsf{plus}(5, 2), y) \leq 15 \Rightarrow \mathsf{plus}(\mathsf{plus}(x, 2), y) \leq 15 \tag{2}$$

By transitivity of $\leq$, it suffices to show:

$$\mathsf{plus}(\mathsf{plus}(5, 2), y) \leq \mathsf{plus}(\mathsf{plus}(x, 2), y)$$

At this point, we can apply property (1) to obtain the two subgoals:

$$\mathsf{plus}(5, 2) \leq \mathsf{plus}(x, 2) \qquad y \leq y$$

We can apply (1) again to the first subgoal; the remaining subgoals are taken care of by the reflexivity of $\leq$ and by our hypothesis H. In fact, all of the steps in this proof other than using H can be seen as applying a relational property of some kind. The relation $\leq$ has type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{Prop}$. Its transitivity means that if $N_1 \leq M_1$ and $M_2 = N_2$, then $M_1 \leq M_2 \Rightarrow N_1 \leq N_2$. In other words:

$$(\leq) \quad [(\leq)^{-1} \to (=) \to (\Rightarrow)] \quad (\leq),$$

which applies to (2). The reflexivity of $\leq$ simply means that for any $M : \mathsf{nat}$, the generic relational property $M \leq M$ holds.

The generalized rewriting system uses the `Proper` typeclass to let the user register such monotonicity properties. It is defined as:

$$\mathsf{Proper}\ R\ M \overset{\text{def.}}{\Leftrightarrow} M\,[R]\,M,$$

so that for example (1) could be registered as an instance of:

$$\mathsf{Proper}\ ((\leq) \to (\leq) \to (\leq))\ \mathsf{plus}.$$

Given a rewriting problem, the `rewrite` tactic generates a set of `Proper` typeclass queries by walking the structure of the goal from the subterm of interest to the root. If the queries succeed, the instances found are used to construct an appropriate proof.

2. Limitations

This ingenious use of logical relations and typeclasses yields a very useful and extensible system. Consequently, in my work on deploying a logical relations approach in the context of a module system for certified low-level code [1], I strived to reuse the underlying infrastructure in an effort to make my development compatible in spirit and practice with the existing system. However, in the course of this process a number of limitations appeared which led me to reimplement parts of that infrastructure.

Heterogenous relations The existing implementation is built around the type $\mathsf{relation}(A) := A \to A \to \mathsf{Prop}$. As a consequence, the definitions and tactics cannot be applied to relations which use different types as their domain and codomain.

This is often restrictive. Logical relations are sometimes used to relate executions of programs written in different languages [2]. In the case of our module system, the two languages are two instantiations of a common language which use different memory models. These situations cannot be handled by homogenous relations.

In contrast, our library is built around the type $\mathsf{rel}(A, B)$ of relations with domain A and codomain B. The usual relators are extended correspondingly, so that for instance $\to$ generalizes to:

$$\frac{R : \mathsf{rel}(A_1, A_2) \qquad S : \mathsf{rel}(B_1, B_2)}{R \to S : \mathsf{rel}(A_1 \to B_1, A_2 \to B_2)}$$

Dependent types Logical relations over dependent types are essential to formulate relational parametricity and similar properties. However the lack of support for heterogenous relations necessarily restricts the range of relational properties which can be stated regarding terms with dependent types. Consider $f : \Pi a : A.B[a]$, and two possible values $x, y : A$ of its argument. The types of the results $B[x]$ and $B[y]$ will potentially be different, hence homogenous relations are no longer sufficient to express their relationship.

Use of* rewrite *as the primitive operation rewrite is useful, but is ill-suited as the main operation of a general-purpose library for binary logical relations. The tactic solve_proper defined in Coq.Classes.Morphisms illustrates this. It solves goals of the form:

$$\mathsf{Proper}\ (R_1 \to \cdots \to R_n \to R)\ (\lambda\, x_1 \ldots x_n\, .\, M[\bar{x}]),$$

where M is built out of components already known to be well-behaved. First, it collects into the context the pairs of arguments $x_1, y_1, \ldots, x_n, y_n$, and corresponding hypotheses $H_i : x_i\ [R_i]\ y_i$. To solve the residual goal we need to prove that $M[\bar{x}]$ and $M[\bar{y}]$ are related by R.

One way to proceed would be to follow a process similar to the one outlined in Sec. 1: at each step, find an appropriate relational property, apply it to the goal, then repeat this process on the subgoals, and use the collected hypotheses H_i's to solve the leaves. Arguably, rewrite and solve_proper share a similar structure and should be built out of the same basic components.

However, because rewrite is used as the primitive operation, solve_proper is not actually implemented in this way. Instead, the residual goal is solved by using the H_i's to rewrite each x_i appearing in the left-hand side into the corresponding y_i. If this succeeds, the left-hand side will become identical to the right-hand side, and the reflexivity tactic is applied to the result. This is unnecessarily complex as each one of the n rewrites restarts the whole process described in Sec. 1. Furthermore, it depends on the fact that S and other relations are reflexive, which is necessary for partially rewritten versions of the goal to be used as intermediate steps.

3. A more general framework

To overcome these limitations, I have been working on a basic binary logical relations infrastructure based on the more general heterogenous relation types $\mathsf{rel}(A, B)$. In the following I outline some of the key features of this new library.

The* monotonicity *tactic The library is built around the monotonicity tactic, which implements a single step of the kind proof outlined in Sec. 1: if the goal is of the form

$$f(M_1, \ldots, M_n)\ [R]\ f(N_1, \ldots, N_n),$$

it will locate an instance of

$$\mathsf{Proper}\ (R_1 \to \cdots \to R_n \to R)\ f$$

to use and generate the n subgoals $M_1\ [R_1]\ N_1, \ldots, M_n\ [R_n]\ N_n$.

***Generalizing* Proper *to* Related** While in most situations, the relational property we want to use consists in a term being related to itself, in the context of the monotonicity tactic we can generalize the Proper typeclass to:

$$\mathsf{Related}\ R\ M\ N \quad \overset{\text{def.}}{\Leftrightarrow} \quad M\ [R]\ N,$$

which allows M to be different from N. Of course, instances of $\mathsf{Proper}\ R\ M$ can be promoted to $\mathsf{Related}\ R\ M\ M$, so that Proper remains the typeclass used for declaring most properties.

The* solve_monotonic *tactic Using monotonicity as the basic component, we can implement a more straighforward version of solve_proper. In essence, the solve_monotonic tactic applies monotonicity repeatedly, and attempts to solve the leaf subgoals using hypotheses from the context.

In turn, solve_monotonic can be used by more advanced tactics to discharge relational subgoals. For instance, it could be used by a new version of rewrite to prove an intermediate lemma such as (2). Another example is transport.

Tranporting hypotheses In a context containing a number of hypotheses of the form $H_i : M_i\ [R_i]\ N_i$, the transport tactic will turn a hypothesis built out of the M_i's into an equivalent one built out of the corresponding N_i's. For instance, in a typical situation, we have a hypothesis

$$H : M = \mathsf{Some}(a),$$

where M is a term of type $\mathsf{option}(A)$ which involves the M_i's. Assuming M is only built out of well-behaved components, it will be possible for solve_monotonic to establish a property of the form $M\ [\mathsf{option}(R)]\ N$ for some N of type $\mathsf{option}(B)$, and some R of type $\mathsf{rel}(A, B)$. Here $\mathsf{option}(R)$ relates terms which are either both None, or of the form $\mathsf{Some}(a)$ and $\mathsf{Some}(b)$ with $a\ [R]\ b$. In this situation, we want the tactic $\mathsf{transport}\ H$ to introduce a new variable b, assert that $a\ [R]\ b$, and transform our hypothesis into:

$$H : N = \mathsf{Some}(b).$$

To make the transport tactic user-extensible, I introduce the following typeclass:

$$\mathsf{Transport}\ R\ M\ N\ P\ Q \quad \overset{\text{def.}}{\Leftrightarrow} \quad M\ [R]\ N \Rightarrow P \Rightarrow Q.$$

Instances of Transport correspond to shapes of hypotheses we wish to be able to transport in this way. For instance, the following instance is used for hypotheses of the form $M = \mathsf{Some}(a)$:

$$\forall\, R\ M\ N\ a\, .\, \mathsf{Transport}\ \mathsf{option}(R)\ M\ N\ (M = \mathsf{Some}(a)) \\ (\exists\, b\, .\, a[R]b \wedge N = \mathsf{Some}(b))$$

The transport tactic locates an appropriate instance of Transport, applies it to the hypothesis of interest, discharges the relational premise using solve_monotonic, and finally splits up the resulting existantial quantifiers and conjunctions.

References

[1] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 595–608. ACM, 2015.

[2] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. *ACM SIGPLAN Notices*, 46(1):133–146, 2011.

[3] M. Sozeau. *Un environnement pour la programmation avec types dépendants*. PhD thesis, 2008.

[4] M. Sozeau. A new look at generalized rewriting in type theory. *J. Formalized Reasoning*, 2(1):41–62, 2009.