Hierarchical Reflection

Luís Cruz-Filipe[1,2] and Freek Wiedijk[1]

[1] NIII, Radboud University of Nijmegen
[2] Center for Logic and Computation, Lisboa
{lcf,freek}@cs.kun.nl

Abstract. The technique of reflection is a way to automate proof construction in type theoretical proof assistants. Reflection is based on the definition of a type of syntactic expressions that gets interpreted in the domain of discourse. By allowing the interpretation function to be partial or even a relation one gets a more general method known as "partial reflection". In this paper we show how one can take advantage of the partiality of the interpretation to uniformly define a family of tactics for equational reasoning that will work in different algebraic structures. The tactics then follow the hierarchy of those algebraic structures in a natural way.

1 Introduction

1.1 Problem

Computers have made formalization of mathematical proof practical. They help getting formalizations correct by verifying all the details, but they also make it easier to formalize mathematics by automatically generating parts of the proofs.

One way to automate proving is the technique called *reflection*. With reflection one describes the desired automation *inside* the logic of the theorem prover, by formalizing relevant meta-theory. Reflection is a common approach for proof automation in type theoretical systems like NuPRL and Coq, as described for example in [1] and [10] respectively. Another name for reflection is "the two-level approach".

In Nijmegen we formalized the Fundamental Theorems of Algebra and Calculus in Coq, and then extended these formalizations into a structured library of mathematics named the C-CoRN library [3, 5]. For this library we defined a reflection tactic called rational that automatically establishes equalities of rational expressions in a field by bringing both to the same side of the equal sign and then multiplying everything out. With this tactic, equalities like

$$\frac{1}{x} + \frac{1}{y} = \frac{x+y}{xy}$$

can be automatically proved without any human help.

The rational tactic only works for expressions in a field, but using the same idea one can define analogous tactics for expressions in a ring or a group. The trivial way to define these is to duplicate the definition of the rational tactic and modify it for these simpler algebraic structures by removing references to division or multiplication. This was actually done to implement a ring version of rational.

However this is not efficient, as it means duplication of the full code of the tactic. In particular the *normalization function* that describes the simplification of expressions, which is quite complicated, has to be defined multiple times. But looking at the normalization function for field expressions, it is clear that it contains the normalization function for rings. In this paper we study a way to *integrate* these tactics for different algebraic structures.

1.2 Approach

In the C-CoRN library algebraic structures like fields, rings and groups are organized into an *Algebraic Hierarchy.* The definition of a field reuses the definition of a ring, and the definition of a ring reuses the definition of a group. This hierarchy means that the theory about these structures is maximally reusable. Lemmas about groups are automatically also applicable to rings and fields, and lemmas about rings also apply to fields.

At the same time, a tactic for proving equalities in arbitrary fields was developed using a partial interpretation relation, as described in [10]. In this paper we show how we can take advantage of this partial interpretation relation to reuse the same tactic for simpler structures. As it turns out, the simplification of expressions done in a field can be directly applied to rings and groups as well. This is quite surprising: the normal forms of expressions that get simplified in this theory will contain functions like multiplication and division, operations that do not make sense in a group.

1.3 Related Work

In the C-CoRN setoid framework, rational is the equivalent of the standard Coq tactic field for Leibniz equality (see [7] and [4, Chapter 8.11]). Both tactics were developed at about the same time. The field tactic is a generalization of the Coq ring tactic [4, Chapter 19], so with the field and ring tactics the duplication of effort that we try to eliminate is also present. Also the ring tactic applies to rings as well as to semirings (to be able to use it with the natural numbers), so there is also this kind of duplication within the ring tactic itself.

Reflection has also been widely used in the NuPRL system as described originally in [1]. More recently, [12] introduces other techniques that allow code reuse for tactics in MetaPRL, although the ideas therein are different from ours. Since the library of this system also includes an algebraic hierarchy built using subtyping (see [15]), it seems reasonable to expect that the work we describe could be easily adapted to that framework.

1.4 Contribution

We show that it is possible to have one unified mechanism for simplification of expressions in different algebraic structures like fields, rings and groups. We also show that it is not necessary to have different normalization functions for these expressions, but that it is possible to decide equalities on all levels with only one normalization function. Presently, both the ring and the field versions of the tactic are used extensively throughout C-CoRN (a total of more than 1.500 times).

Another extension which we present is the addition of uninterpreted function symbols. With it we can now automatically prove goals of the form $|t_1| = |t_2|$, which earlier had to be manually simplified to $t_1 = t_2$.

The whole tactic is about 100kb of code, divided between the ML implementation (17kb), the normalization function (14kb) and the interpretation relation and correctness (23kb for groups, 25kb for rings and 29kb for fields); in Section 6 we discuss why the correctness has to be proved anew for each structure.

We compared the speed of our tactic with that of ring and field, and also with a similar tactic for the HOL Light system [11]. All these tactics have a comparable speed: our tactic is a bit faster than ring, but slower than field.

1.5 Outline

In Section 2 we summarize the methods of reflection and partial reflection. In Section 3 we describe in detail the normalization function of the rational tactic. Section 4 is a small detour where we generalize the same method to add uninterpreted function symbols to the expressions that rational understands. In Section 5 we show how to do reflection in an algebraic hierarchy in a hierarchical way. Finally in Section 6 we present a possibility to have even tighter integration in a hierarchical reflection tactic, which unfortunately turns out to require the so-called K axiom [14].

2 Reflection and Partial Reflection

In this section we will briefly summarize [10]. That paper describes a generalization of the technique of *reflection* there called *partial reflection*. One can give a general account of reflection in terms of decision procedures, but here we will only present the more specific method of reflection with a normalization function, which is used to do equational reasoning.

In the normal, "total", kind of reflection one defines a type E of *syntactic expressions* for the domain A that one is reasoning about, together with an *interpretation function*

$$[\![-]\!]_\rho : E \to A$$

which assigns to a syntactic expression e an interpretation $[\![e]\!]_\rho$. In this, ρ is a *valuation* that maps the variables in the syntactic expressions to values in A. The type E is an inductive type, and therefore it is possible to recursively define a *normalization function* $\mathcal{N}$ on the type of syntactic expressions *inside* the type theory (this is not possible for A; so the reason for introducing the type E is to be able to define this $\mathcal{N}$).

$$\mathcal{N} : E \to E$$

One then proves the correctness lemma stating that the normalization function conserves the interpretation.

$$[\![e]\!]_\rho =_A [\![\mathcal{N}(e)]\!]_\rho$$

Then, to reason about the domain A that $[\![-]\!]$ maps to, one first constructs a valuation ρ and syntactic expressions in E which map under $[\![-]\!]_\rho$ to the terms that one want to reason about, and then one uses the lemma to do the equational reasoning.

For instance, suppose that one wants to prove $a =_A b$. One finds e, f and ρ with $[\![e]\!]_\rho = a$ and $[\![f]\!]_\rho = b$. Now if $\mathcal{N}(e) = \mathcal{N}(f)$ then we get $a = [\![e]\!]_\rho =_A [\![\mathcal{N}(e)]\!]_\rho = [\![\mathcal{N}(f)]\!]_\rho =_A [\![f]\!]_\rho = b$. (Clearly this uses the correctness lemma twice, see Figure 1.) Note that the operation of finding an expression in E that

$$\begin{tikzcd}
e \in E \arrow[rr, "\mathcal{N}"] \arrow[dr, bend right, "{[\![-]\!]_\rho}"'] & & \mathcal{N}(e) = \mathcal{N}(f) \arrow[dl, "{[\![-]\!]_\rho}"'] \arrow[dr, "{[\![-]\!]_\rho}"] & & f \in E \arrow[ll, "\mathcal{N}"'] \arrow[dl, bend left, "{[\![-]\!]_\rho}"] \\
& a \in A \arrow[ul, dashed] \arrow[rr, leftrightarrow, "=_A"] & & b \in A \arrow[ur, dashed] &
\end{tikzcd}$$

Fig. 1. Proving equalities

corresponds to a given expression in A (dotted arrows) is not definable in the type theory, and needs to be implemented *outside* of it. In a system like Coq it will be implemented in ML or in the tactic language $\mathcal{L}_{\text{tac}}$ described in [6] and [4, Chapter 9].

Things get more interesting when the syntactic expressions in E contain *partial* operations, like division. In that case the interpretation $[\![e]\!]_\rho$ will not always be defined. To address this we generalized the method of reflection to *partial reflection*. The naive way to do this is to define a predicate

$$wf_\rho : E \to \mathsf{Prop}$$

that tells whether an expression is *well-formed*. Then the interpretation function takes another argument of type $wf_\rho(e)$.

$$[\![-]\!]_\rho : \Pi_{e:E}.\, wf_\rho(e) \to F$$

The problem with this approach is that the definition of *wf* needs the interpretation function $[\![-]\!]$. Therefore the inductive definition of *wf* and the recursive definition of $[\![-]\!]$ need to be given simultaneously. This is called an *inductive-recursive definition*. Inductive-recursive definitions are not supported by the Coq system, and for a good reason: induction-recursion makes a system significantly stronger. In set theory it corresponds to the existence of a Mahlo cardinal [8].

The solution from [10] for doing partial reflection without induction-recursion is to replace the interpretation function with an inductively defined *interpretation relation*.

$$\mathrel{[\![}_\rho \subseteq E \times A$$

The relation $[\![e]\!]_\rho = a$ now becomes $e \mathrel{[\![}_\rho a$. It means that the syntactic expression e is interpreted under the valuation ρ by the object a. The lemmas that one then proves are the following.

$$e \mathrel{[\![}_\rho a \ \wedge\ e \mathrel{[\![}_\rho b \ \Rightarrow\ a =_A b$$
$$e \mathrel{[\![}_\rho a \ \Rightarrow\ \mathcal{N}(e) \mathrel{[\![}_\rho a$$

The first lemma states that the interpretation relation is *functional*, and the second lemma is again the correctness of the normalization function. Note that it is not an equivalence but just an implication. This is the only direction that is needed. In fact, in our application the equivalence does not hold.[3]

For each syntactic expression e that one constructs for an object a, one also needs to find an inhabitant of the statement $e \mathrel{[\![}_\rho a$. In [10] types $\bar{E}_\rho(a)$ of *proof loaded syntactic expressions* are introduced to make this easier. These types correspond to the expressions that evaluate to a. They are mapped to the normal syntactic expressions by a forgetful function

$$|-| : \bar{E}_\rho(a) \to E$$

and they satisfy the property that for all $\bar{e}$ in the type $\bar{E}_\rho(a)$

$$|\bar{e}| \mathrel{[\![}_\rho a.$$

In this paper we will not go further into this, although everything that we do also works in the presence of these proof loaded syntactic expressions.

3 Normalization Function

We will now describe how we defined the normalization function for our main example of *rational expressions*. Here the type E of syntactic expressions is given by the following grammar.

$$E ::= \mathbb{Z} \mid \mathbb{V} \mid E + E \mid E \cdot E \mid E/E$$

[3] A simple example is $e = 1/(1/0)$, which does not relate to any a. Its normal form is $0/1$, which interprets to 0.

In this $\mathbb{Z}$ are the integers, and $\mathbb{V}$ is a countable set of *variable names* (in the Coq formalization we use a copy of the natural numbers for this). Variables will be denoted by x, y, z, integers by i, j, k. The elements of this type E are just *syntactic* objects, so they are different kind of objects from the *values* of these expressions in specific fields. Note that in these expressions it is possible to divide by zero: $0/0$ is one of the terms in this type.

Other algebraic operations are defined as an abbreviation from operations that occur in the type. For instance, subtraction is defined by

$$e - f \equiv e + f \cdot (-1)$$

We now will describe how we define the normalization function $\mathcal{N}(e)$ that maps an element of E to a normal form. As an example, the normal form of $\frac{1}{x-y} + \frac{1}{x+y}$ is

$$\mathcal{N}(\frac{1}{x-y} + \frac{1}{x+y}) = \frac{x \cdot 2 + 0}{x \cdot x \cdot 1 + y \cdot y \cdot (-1) + 0}.$$

This last expression is the "standard form" of the way one would normally write this term, which is

$$\frac{2x}{x^2 - y^2}.$$

From this example it should be clear how the normalization function works: it multiplies everything out until there is just a quotient of two polynomials left. These polynomials are then in turn written in a "standard form". The expressions in normal form are given by the following grammar.

$$\begin{aligned} F &::= P/P \\ P &::= M + P \mid \mathbb{Z} \\ M &::= \mathbb{V} \cdot M \mid \mathbb{Z} \end{aligned}$$

In this grammar F represents a fraction of two polynomials, P are the polynomials and M are the monomials. One should think of P as a "list of monomials" (where $+$ is the "cons" and the integers take the place of the "nil") and of M as a "list of variables" (where $\cdot$ is the "cons" and again the integers take the place of the "nil").

On the one hand we want the normalization function to terminate, but on the other hand we want the set of normal forms to be as small as possible. We achieve this by requiring the polynomials and monomials to be *sorted*; furthermore, no two monomials in a polynomial can have exactly the same set of variables. Thus normal forms for polynomials will be unique.

For this we have an ordering of the variable names. So the "list" that is a monomial has to be sorted according to this order on $\mathbb{V}$, and the "list" that is a polynomial also has to be sorted, according to the corresponding lexicographic ordering on the monomials. If an element of P or M is sorted like this, and monomials with the same set of variables have been collected together, we say it is *in normal form*.

Now to define $\mathcal{N}$ we have to "program" the multiplying out of E expressions together with the sorting of monomials and polynomials, and collecting factors and terms. This is done simultaneously: instead of first multiplying out the expressions and then sorting them to gather common terms, we combine these two things.

We recursively define the following functions (using the `Fixpoint` operation of Coq).

$$\begin{array}{ll}
- \cdot_{M\mathbb{Z}} - & : M \times \mathbb{Z} \to M \\
- \cdot_{M\mathbb{V}} - & : M \times \mathbb{V} \to M \\
- \cdot_{MM} - & : M \times M \to M \\
- +_{MM} - & : M \times M \to M \\
- +_{PM} - & : P \times M \to P \\
- +_{PP} - & : P \times P \to P \\
- \cdot_{PM} - & : P \times M \to P \\
- \cdot_{PP} - & : P \times P \to P \\
- +_{FF} - & : F \times F \to F \\
- \cdot_{FF} - & : F \times F \to F \\
- /_{FF} - & : F \times F \to F
\end{array}$$

(Actually, these functions all have type

$$E \times E \to E$$

as we do not have separate types for F, P and M. However, the idea is that they only will be called with arguments that are of the appropriate shape and in normal form. In that case the functions will return the appropriate normal form. In the other case they will return any term that is equal to the sum or product of the arguments — generally we just use the sum or product of the arguments.)

For example, the multiplication function $\cdot_{MM}$ looks like

$$e \cdot_{MM} f := \begin{cases} f \cdot_{M\mathbb{Z}} i & \text{if } e = i \in \mathbb{Z} \\ (e_2 \cdot_{MM} f) \cdot_{MV} e_1 & \text{if } e = e_1 \cdot e_2 \\ e \cdot f & \text{otherwise} \end{cases}$$

and the addition function $+_{PM}$ is[4]

$$e +_{PM} f := \begin{cases} i +_{MM} j & \text{if } e = i \in \mathbb{Z},\, f = j \in \mathbb{Z} \\ f + i & \text{if } e = i \in \mathbb{Z} \\ e_1 + (e_2 +_{PM} i) & \text{if } e = e_1 + e_2,\, f = i \in \mathbb{Z} \\ e_2 +_{PM} (e_1 +_{MM} f) & \text{if } e = e_1 + e_2,\, e_1 = f \\ e_1 + (e_2 +_{PM} f) & \text{if } e = e_1 + e_2,\, e_1 <_{\text{lex}} f \\ f + e & \text{if } e = e_1 + e_2,\, e_1 >_{\text{lex}} f \\ e + f & \text{otherwise} \end{cases}$$

where the lexicographic ordering $<_{\text{lex}}$ is used to guarantee that the monomials in the result are ordered.

[4] In the fourth case, the equality $e_1 = f$ is equality *as lists*, meaning that they might differ in the integer coefficient at the end.

Finally we used these functions to recursively "program" the normalization function. For instance the case where the argument is a division is defined like

$$\mathcal{N}(e/f) := N(e) /_{FF} N(f).$$

The base case (when e is a variable) looks like

$$\mathcal{N}(v) := \frac{v \cdot 1 + 0}{1}.$$

To prove that $a = b$, then, one builds the expression corresponding to $a - b$ and checks that this normalizes to an expression of the form $0/e$. (This turns out to be stronger than building expressions e and f interpreting to a and b and verifying that $\mathcal{N}(e) = \mathcal{N}(f)$, since normal forms are in general not unique.)

4 Uninterpreted Function Symbols

When one starts working with the tactic defined as above, one quickly finds out that there are situations in which it fails because two terms which are easily seen to be equal generate two expressions whose difference fails to normalize to 0. A simple example arises is when function symbols are used; for example, trying to prove that

$$f(a + b) = f(b + a)$$

will fail because $f(a + b)$ will be syntactically represented as a variable x and $f(b + a)$ as a (different) variable y, and the difference between these expressions normalizes to

$$\frac{x \cdot 1 + y \cdot (-1) + 0}{1},$$

which is not zero.

In this section we describe how the syntactic type E and the normalization function $\mathcal{N}$ can be extended to recognize and deal with function symbols. The actual implementation includes unary and binary total functions, as well as unary partial functions (these are binary functions whose second argument is a proof)[5]. We will discuss the case for unary total functions in detail; binary and partial functions are treated in an analogous way.

Function symbols are treated much in the same way as variables; thus, we extend the type E of syntactic expressions with a new countable set of *function variable names* $\mathbb{V}_1$, which is implemented (again) as the natural numbers. The index 1 stands for the arity of the function; the original set of variables is now denoted by $\mathbb{V}_0$. Function variables will be denoted u, v.

$$E ::= \mathbb{Z} \mid \mathbb{V}_0 \mid \mathbb{V}_1(E) \mid E + E \mid E \cdot E \mid E/E$$

[5] Other possibilities, such as ternary functions or binary partial functions, were not considered because this work was done in the setting of the C-CoRN library, where these are the types of functions which are used in practice.

Intuitively, the normalization function should also normalize the arguments of function variables. The grammar for normal forms becomes the following.

$$\begin{aligned} F &::= P/P \\ P &::= M + P \mid \mathbb{Z} \\ M &::= \mathbb{V}_0 \cdot M \mid \mathbb{V}_1(F) \cdot M \mid \mathbb{Z} \end{aligned}$$

But now a problem arises: the extra condition that both polynomials and monomials correspond to sorted lists requires ordering not only variables in $\mathbb{V}_0$, but also expressions of the form $\mathbb{V}_1(F)$. The simplest way to do this is by defining an ordering on the whole set E of expressions.

This is achieved by ordering first the sets $\mathbb{V}_0$ and $\mathbb{V}_1$ themselves. Then, expressions are recursively sorted by first looking at their outermost operator

$$x <_E i <_E e + f <_E e \cdot f <_E e/f <_E v(e)$$

and then sorting expressions with the same operator using a lexicographic ordering. For example, if $x <_{\mathbb{V}_0} y$ and $u <_{\mathbb{V}_1} v$, then

$$x <_E y <_E 2 <_E 34 <_E x/4 <_E u(x+3) <_E u(2 \cdot y) <_E v(x+3).$$

With this different ordering, the same normalization function as before can be used with only trivial changes. In particular, the definitions of the functions $\cdot_{MM}$ and $+_{PM}$ remain unchanged. Only at the very last step does one have to add a rule saying that

$$\mathcal{N}(v(e)) := \frac{v(\mathcal{N}(e)) \cdot 1 + 0}{1}.$$

Notice the similarity with the rule for the normal form of variables.

The next step is to change the interpretation relation. Instead of the valuation ρ, we now need two valuations

$$\begin{aligned} \rho_0 &: \mathbb{V}_0 \to A \\ \rho_1 &: \mathbb{V}_1 \to (A \to A) \end{aligned}$$

and the inductive definition of the interpretation relation is extended with the expected constructor for interpreting expressions of the form $v(e)$.

As before, one can again prove the two lemmas

$$e \mathrel{[\![}_{\rho_0,\rho_1} a \;\wedge\; e \mathrel{[\![}_{\rho_0,\rho_1} b \;\Rightarrow\; a =_A b$$

$$e \mathrel{[\![}_{\rho_0,\rho_1} a \;\Rightarrow\; \mathcal{N}(e) \mathrel{[\![}_{\rho_0,\rho_1} a$$

Our original equality $f(a+b) = f(b+a)$ can now easily be solved: $f(a+b)$ can be more faithfully represented by the expression $v(x+y)$, where $\rho_1(v) = f$, $\rho_0(x) = a$ and $\rho_0(y) = b$; the syntactic representation of $f(b+a)$ becomes $v(y+x)$; and each of these normalizes to

$$\frac{v\left(\frac{x \cdot 1 + y \cdot 1 + 0}{1}\right) \cdot 1 + 0}{1},$$

so that their difference normalizes to 0 as was intended.

Adding binary functions simply requires a new sort $\mathbb{V}_2$ of binary function symbols and extend the type of expressions to allow for the like of $v(e, f)$; the normalization function and the interpretation relation can easily be adapted, the latter requiring yet another valuation

$$\rho_2 : \mathbb{V}_2 \to (A \times A \to A).$$

Partial functions are added likewise, using a sort $V_{\not 1}$ for partial function symbols and a valuation

$$\rho_{\not 1} : \mathbb{V}_{\not 1} \to (A \not\to A).$$

As was already the case with division, one can write down expressions like $v(e)$ even when $\rho_{\not 1}(v)$ is not defined at the interpretation of e; the definition of $\mathbb{I}_{\rho_0,\rho_1,\rho_{\not 1},\rho_2}$ ensures that only correctly applied partial functions will be interpreted.

5 Hierarchical Reflection

The normalization procedure described in Section 3 was used to define a tactic which would prove algebraic equalities in an arbitrary field in the context of the Algebraic Hierarchy of [9].

In this hierarchy, fields are formalized as rings with an extra operation (division) which satisfies some properties; rings, in turn, are themselves Abelian groups where a multiplication is defined also satisfying some axioms. The question then arises of whether it is possible to generalize this mechanism to the different structures of this Algebraic Hierarchy. This would mean having three "growing" types of syntactic expressions E_G, E_R and E_F (where the indices stand for groups, rings and fields respectively) together with interpretation relations[6].

$$\begin{tikzcd}
E_F \arrow[r, "\mathbb{I}^F_\rho"] & F : \text{Field} \arrow[d] \\
E_R \arrow[u, "\subseteq"] \arrow[r, "\mathbb{I}^R_\rho"] & R : \text{Ring} \arrow[d] \\
E_G \arrow[u, "\subseteq"] \arrow[r, "\mathbb{I}^G_\rho"] & G : \text{Group}
\end{tikzcd}$$

However one can do better. The algorithm in the normalization function works outwards; it first pushes all the divisions to the outside, and then proceeds to normalize the resulting polynomials. In other words, it first deals with the field-specific part of the expression, and then proceeds working within a ring.

[6] For simplicity we focus on the setting where function symbols are absent; the more general situation is analogous.

This suggests that the same normalization function could be reused to define a decision procedure for equality of algebraic expressions within a ring, thus allowing E_F and E_R to be unified.

Better yet, looking at the functions operating on the polynomials one also quickly realizes that these will never introduce products of variables unless they are already implicitly in the expression (in other words, a new product expression can arise e.g. from distributing a sum over an existing product, but if the original expression contains no products then neither will its normal form). So our previous picture can be simplified to this one.

$$\begin{tikzcd}
E \arrow[r, "\mathbb{I}_\rho^F"] \arrow[dr, bend right, "\mathbb{I}_\rho^R"] \arrow[ddr, bend right, "\mathbb{I}_\rho^G"'] & F : \text{Field} \arrow[d] \\
& R : \text{Ring} \arrow[d] \\
& G : \text{Group}
\end{tikzcd}$$

The key idea is to use the partiality of the interpretation relation to be able to map E into a ring R or a group G. In the first case, expressions of the form e/f will not be interpreted; in the latter, neither these nor expressions of the form $e \cdot f$ relate to any element of the group.

There is one problem, however. Suppose x is a variable with $\rho(x) = a$; then $a + a$ is represented by $x + x$, but

$$\mathcal{N}(x+x) = \frac{x \cdot 2 + 0}{1} \; \mathbb{I}_\rho^G \; a + a$$

does not hold.

In order to make sense of the normal forms defined earlier, one needs to interpret the special cases $e/1$ in groups and rings, as well as $e \cdot f$ with $f = i \in \mathbb{Z}$ in groups (assuming, of course, that e can be interpreted).

The following table summarizes what each of the interpretation relations can interpret.

	$\mathbb{I}_\rho^G$	$\mathbb{I}_\rho^R$	$\mathbb{I}_\rho^F$
$v \in \mathbb{V}$	yes	yes	yes
$i \in \mathbb{Z}$	if $i = 0$	yes	yes
$e + f$	yes	yes	yes
$e \cdot f$	if $f \in \mathbb{Z}$	yes	yes
e/f	if $f = 1$	if $f = 1$	if $f \neq 0$

In the last three cases the additional requirement that e and f can be interpreted is implicit.

Once again, one has to prove the lemmas

$$e \rrbracket^G_\rho a \;\wedge\; e \rrbracket^G_\rho b \;\Rightarrow\; a =_A b$$
$$e \rrbracket^G_\rho a \;\Rightarrow\; \mathcal{N}(e) \rrbracket^G_\rho a$$

and analogous for $\rrbracket^R_\rho$ and $\rrbracket^F_\rho$.

In these lemmas, one needs to use the knowledge that the auxiliary functions will only be applied to the "right" arguments to be able to finish the proofs. This is trickier to do for groups than for rings and fields. For example, while correctness of $\cdot_{MM}$ w.r.t. $\rrbracket^F_\rho$ is unproblematic, as it states that

$$e \rrbracket^F_\rho a \;\wedge\; f \rrbracket^F_\rho b \;\Rightarrow\; e \cdot_{MM} f \rrbracket^F_\rho a \cdot b,$$

the analogue of this statement for $\rrbracket^G_\rho$ cannot be written down, as $a \cdot b$ has no meaning in a group. However, by definition of $\rrbracket^F_\rho$, this is equivalent to the following.

$$e \cdot f \rrbracket^F_\rho a \cdot b \;\Rightarrow\; e \cdot_{MM} f \rrbracket^F_\rho a \cdot b$$

Now this second version does possess an analogue for $\rrbracket^G_\rho$, by replacing the expression $a \cdot b$ with a variable.

$$e \cdot f \rrbracket^G_\rho c \;\Rightarrow\; e \cdot_{MM} f \rrbracket^G_\rho c.$$

This is still not provable, because $\cdot_{MM}$ can swap the order of its arguments. The correct version is

$$e \cdot f \rrbracket^G_\rho c \;\vee\; f \cdot e \rrbracket^G_\rho c \;\Rightarrow\; e \cdot_{MM} f \rrbracket^G_\rho c;$$

the condition of this statement reflects the fact that the normalization function will only require computing $e \cdot_{MM} f$ whenever either e or f is an integer.

The implementation of the tactic for the hierarchical case now becomes slightly more sophisticated than the non-hierarchical one. When given a goal $a =_A b$ it builds the syntactic representation of a and b as before; and then looks at the type of A to decide whether it corresponds to a group, a ring or a field. Using this information the tactic can then call the lemma stating correctness of $\mathcal{N}$ w.r.t. the appropriate interpretation relation.

Optimization

As was mentioned in Section 3, normal forms for polynomials are unique, contrarily to what happens with field expressions in general. This suggests that, when A is a group or a ring, the decision procedure for $a =_A b$ can be simplified by building expressions e and f interpreting respectively to a and b and comparing their normal forms. Clearly, this is at most as time-consuming as the previous version, since computing $\mathcal{N}(e - f)$ requires first computing $\mathcal{N}(e)$ and $\mathcal{N}(f)$.

Also, since the normalization function was not defined at once, but resorting to the auxiliary functions earlier presented, it is possible to avoid using divisions altogether when working in rings and groups by defining directly $\mathcal{N}'$ by e.g.

$$\mathcal{N}'(e+f) = \mathcal{N}'(e) +_{PP} \mathcal{N}(f)';$$

the base case now looks like

$$\mathcal{N}'(v) = v \cdot 1 + 0.$$

Notice that although we now have two different normalization functions we still avoid duplication of the code, since they are both defined in terms of the same auxiliary functions and these are where the real work is done.

6 Tighter Integration

In the previous section we managed to avoid having different syntactic expressions for the different kinds of algebraic structures. We unified the types of syntactic expressions into one type E.

However we still have different interpretation relations $\llbracket_\rho^F$, $\llbracket_\rho^R$ and $\llbracket_\rho^G$. We will now analyze the possibility of unifying those relations into one interpretation relation $\llbracket_\rho^S$. This turns out to be possible, but when one tries to prove the relevant lemmas for it one runs into problems: to get the proofs finished one needs to assume an axiom (in the type theory of Coq).

Every field, ring or group has an underlying carrier. We will write $\hat{A}$ for the carrier of an algebraic structure A. We now define an interpretation relation $\llbracket_\rho^S$ from the type of syntactic expressions E to an *arbitrary* set[7] S, where that set is a parameter of the inductive definition. This inductive definition quantifies over *different* kinds of algebraic structures in the clauses for the different algebraic operations. For instance the inductive clause for addition quantifies over groups.

$$\Pi_{G:\mathrm{Group}}\Pi_{e,f:E}\Pi_{a,b,c:\hat{G}}\,(a +_G b =_G c) \to (e \llbracket_\rho^{\hat{G}} a) \to (f \llbracket_\rho^{\hat{G}} b) \to (e+f \llbracket_\rho^{\hat{G}} c)$$

With this definition the diagram becomes the following.

$$\begin{tikzcd}
E \arrow[dddr, bend right, "\llbracket_\rho^S"'] & F : \mathrm{Field} \arrow[d] \\
& R : \mathrm{Ring} \arrow[d] \\
& G : \mathrm{Group} \arrow[d, "\hat{\cdot}"] \\
& S : \mathrm{Set}
\end{tikzcd}$$

[7] In the formalization we actually have *setoids* instead of sets, but that does not make a difference.

This gives a nice unification of the interpretation relations. However, when one tries to prove the relevant lemmas for it in Coq, the obvious way does not work. To prove e.g.

$$e \, [\![^S_\rho \, a \;\wedge\; e \, [\![^S_\rho \, b \;\Rightarrow\; a =_A b$$

one needs to use inversion with respect to the inductive definition of $[\![^S_\rho$ to get the possible ways that $e \, [\![^S_\rho \, a$ can be obtained; but the inversion tactic of Coq then only produces an equality between dependent pairs where what one needs is equality between the second components of those pairs. In Coq this is not derivable without the so-called K axiom, which states uniqueness of equality proofs [13].

```
forall (A:Set) (x:A) (p:(x=x)), p = refl_equal A x
```

We did not want to assume an axiom to be able to have our tactic prove equalities in algebraic structures that are clearly provable without this axiom. For this reason we did not fully implement this more integrated version of hierarchical reflection.

7 Conclusion

7.1 Discussion

We have shown how the rational tactic (first described in [10]), which is used to prove equalities of expressions in arbitrary fields, can be generalized in two distinct directions.

First, we showed in Section 4 how this tactic could be extended so that it would also look at the arguments of functions; the same mechanism can be applied not only to unary total functions, as explained, but also to binary (or n-ary) functions, as well as to partial functions as defined in the C-CoRN library [3].

In Section 5 we discussed how the same syntactic type E and normalization function $\mathcal{N}$ could be reused to define similar tactics that will prove equalities in arbitrary rings or commutative groups. The work described here has been successfully implemented in Coq, and is intensively used throughout the whole C-CoRN library.

Further extensions of this tactic are possible; in particular, the same approach easily yields a tactic that will work in commutative monoids (e.g. the natural numbers with addition). For simplicity, and since this adds nothing to this presentation, this situation was left out of this paper.

Extending the same mechanism to non-commutative structures was not considered. The normalization function intensively uses commutativity of both addition and multiplication, so it cannot be reused for structures that do not satisfy these; and the purpose of this work was to reuse as much of the code needed for rational as possible.

The correctness of the normalization function w.r.t. the interpretation relation had to be proved three times, one for each type of structure. In Section 6 we showed one possible way of overcoming this, which unfortunately failed because proving correctness of the tactic would then require assuming an axiom which is not needed to prove the actual equalities that the tactic is meant to solve. It would be interesting to know whether this approach can be made to work without needing the K axiom. Though this axiom is required to prove these lemmas using inversion, there might be an alternative way to prove them that avoids this problem.

A different approach to the same problem would be to use the constructor subtyping of [2]. This would allow one to define e.g. the interpretation relation for rings $[\![\,]\!]_\rho^R$ by adding one constructor to that for groups $[\![\,]\!]_\rho^G$; proving the relevant lemmas for the broader relation would then only require proving the new case in all the inductive proofs instead of duplicating the whole code.

Another advantage of this solution, when compared to the one explored in Section 6, would be that the tactic could be programmed and used for e.g. groups before rings and fields were even defined. It would also be more easily extendable to other structures. Unfortunately, constructor subtyping for Coq is at the moment only a theoretical possibility which has not been implemented.

Acknowledgments. The first author was partially supported by FCT and FEDER under POCTI, namely through grant SFRH / BD / 4926 / 2001 and CLC project FibLog FEDER POCTI / 2001 / MAT / 37239.

References

1. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The Semantics of Reflected Proof. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 95–197, Philadelphia, Pennsylvania, June 1990. IEEE, IEEE Computer Society Press.
2. Gilles Barthe and Femke van Raamsdonk. Constructor subtyping in the Calculus of Inductive Constructions. In Jerzy Tiuryn, editor, *Proceedings 3rd Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS'2000, Berlin, Germany, 25 March – 2 Apr 2000*, volume 1784, pages 17–34, Berlin, 2000. Springer-Verlag.
3. Constructive Coq Repository at Nijmegen. `http://c-corn.cs.kun.nl/`.
4. The Coq Development Team. *The Coq Proof Assistant Reference Manual*, April 2004. Version 8.0.
5. L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN: the Constructive Coq Repository at Nijmegen. To appear.
6. David Delahaye. A Tactic Language for the System Coq. In Michel Parigot and Andrei Voronkov, editors, *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955 of *LNCS*, pages 85–95. Springer-Verlag, 2000.
7. David Delahaye and Micaela Mayero. Field: une procédure de décision pour les nombres réels en Coq. *Journées Francophones des Langages Applicatifs*, January 2001.
8. Peter Dybjer and Anton Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124:1–47, 2003.

9. H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. The Algebraic Hierarchy of the FTA Project. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, pages 271–286, 2002.
10. H. Geuvers, F. Wiedijk, and J. Zwanenburg. Equational Reasoning via Partial Reflection. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000*, volume 1869 of *LNCS*, pages 162–178, Berlin, Heidelberg, New York, 2000. Springer Verlag.
11. John Harrison. *The HOL Light manual (1.1)*, 2000. `http://www.cl.cam.ac.uk/users/jrh/hol-light/manual-1.1.ps.gz`.
12. Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A Modular Logical Environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *LNCS*, pages 287–303. Springer-Verlag, 2003.
13. Martin Hoffman and Thomas Streicher. The Groupoid Interpretation of Type Theory. In Giovanni Sambin and Jan Smith, editors, *Proceedings of the meeting of Twenty-five years of constructive type theory, Venice*. Oxford University Press, 1996.
14. Thomas Streicher. *Semantical Investigations into Intensional Type Theory*. LMU München, 1993. Habilitationsschrift.
15. Xin Yu, Aleksey Nogin, Alexei Kopylov, and Jason Hickey. Formalizing Abstract Algebra in Type Theory with Dependent Records. In David Basin and Burkhart Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003). Emerging Trends Proceedings*, pages 13–27. Universität Freiburg, 2003.