Equational Reasoning via Partial Reflection

H. Geuvers, F. Wiedijk, J. Zwanenburg
{herman,freek,janz}@cs.kun.nl

Department of Computer Science, Nijmegen University, the Netherlands

Abstract. We modify the reflection method to enable it to deal with partial functions like division. The idea behind reflection is to program a tactic for a theorem prover not in the implementation language but in the object language of the theorem prover itself. The main ingredients of the reflection method are a syntactic encoding of a class of problems, an interpretation function (mapping the encoding to the problem) and a decision function, written on the encodings. Together with a correctness proof of the decision function, this gives a fast method for solving problems. The contribution of this work lies in the extension of the reflection method to deal with equations in algebraic structures where some functions may be partial. The primary example here is the theory of fields. For the reflection method, this yields the problem that the interpretation function is not total. In this paper we show how this can be overcome by defining the interpretation as a relation. We give the precise details, both in mathematical terms and in Coq syntax. It has been used to program our own tactic 'Rational', for verifying equations between field elements.

1 Introduction

We present a method for proving equations between field elements (e.g. real numbers) in a theorem prover based on type theory. Our method uses the *reflection method* as discussed in [6, 5]: we encode the set of syntactic expressions as an (inductive) data type, together with an interpretation function $[\![-]\!]$ that maps the syntactic expressions to the field elements. Then one writes a 'normalization' function $\mathcal{N}$ that simplifies syntactic expressions and one proves that this function is correct, i.e. if $\mathcal{N}(t) = q$, then the interpretations of t and q ($[\![t]\!]$ and $[\![q]\!]$) are equal in the field. Now, to prove an equality between field elements a and b, one has to find syntactic expressions t_1 and t_2 such that $\mathcal{N}(t_1) = \mathcal{N}(t_2)$ and $[\![t_1]\!]$ is a and $[\![t_2]\!]$ is b. This method has been applied successfully [2] to ring expressions in the theorem prover Coq, where it is implemented as the 'Ring tactic': when presented with a goal $a = b$, where a and b are elements of a ring, the Ring tactic finds the underlying syntactic expressions for a and b, executes the normalization function and checks the equality of the normal forms.

The application of the reflection method to the situation of fields poses one big extra problem: syntactic expressions may not have an interpretation, e.g. $\frac{1}{0}$. So, there is no interpretation function from the syntactic expressions to the actual field ($[\![-]\!]$ would be partial). The solution that we propose here is to write an

interpretation *relation* instead: a binary relation between syntactic expressions and field elements. Then we prove that this relation is a partial function. The precise way of using this approach is discussed below, including the technical details of its implementation in Coq. For the precise encodings in Coq see [4].

The reflection method in general

Reflection is the method of 'reflecting' part of the meta language in the object language. Then meta theoretic results can be used to prove results from the object language. Reflection is also called *internalization* or the *two level approach*: the *meta language level* is *internalised* in the object language. The reflection method can (and it has, see e.g. [7]) be used in general in situations where one has a specific class of problems with a decision function. It is also not just restricted to the theorem prover Coq. If the theorem prover allows (A) user defined (inductive) data types, (B) writing executable functions over these data types and (C) user defined tactics in the meta language, then the reflection method can be applied. The classes of problems that it can be applied to are those where (1) there is a syntactic encoding of the class of problems as a data type, say via the type $\mathsf{Problem}$, with (2) a decoding function $[\![-]\!] : \mathsf{Problem} \to \mathsf{Prop}$ (where Prop is the collection of propositions in the language of our theorem prover), (3) there is a decision function $\mathsf{Dec} : \mathsf{Problem} \to \{0,1\}$ such that (4) one can prove $\forall p{:}\mathsf{Problem}((\mathsf{Dec}(p) = 1) \to [\![p]\!])$. Now, if the goal is to verify whether a problem P from the class of problems holds, one has to find a $p : \mathsf{Problem}$ such that $[\![p]\!] = P$. Then $\mathsf{Dec}(p)$ (together with the proof of (4)) yields either a proof of P (if $\mathsf{Dec}(p) = 1$) or it 'fails' (if $\mathsf{Dec}(p) = 0$ we obtain no information about P). Note that if Dec is complete, i.e. if $\forall p{:}\mathsf{Problem}((\mathsf{Dec}(p) = 1) \leftrightarrow [\![p]\!])$, then $\mathsf{Dec}(p) = 0$ yields a proof of $\neg P$. The construction of p (the syntactic encoding) from P (the original problem) can be done in the implementation language of the theorem prover. Therefore it is convenient that the user has access to this implementation language; this is condition (C) above. If the user has no access to the meta language, the reflection method still works, but the user has to construct the encoding p himself, which is very cumbersome.

In this paper we first explain the reflection method by looking at the example of numbers with multiplication. We point out precisely which are the essential ingredients. Then we extend the example by looking at numbers with multiplication and division. Here the partiality problem arises. We explain how the reflection method can be applied to this example. This is an illustration of what we have implemented in Coq: a tactic for solving equations between elements of a field (a set with multiplication, division, addition, subtraction, constants and variables). The tactic has been applied successfully in a formalization of real numbers in Coq that we are currently working on.

2 Equational reasoning using the reflection method

We explain the reflection method by the simple example of numbers with multiplication. Suppose we have $F : \mathsf{Set}$, $\cdot : F \to F \to F$, $1 : F$ and an equivalence

relation $=_F$ on F (either a built-in equality of the theorem prover or a user defined relation) such that

(i) $=_F$ is a congruence for $\cdot$ (i.e. if $a =_F b$ and $c =_F d$, then $a \cdot c =_F b \cdot d$),
(ii) $\cdot$ is associative and commutative,
(iii) 1 is the unit with respect to $\cdot$.

Phrased differently, $\langle F, \cdot, 1\rangle$ is an Abelian monoid. When dealing with F, we will want to prove equations like

$$(a \cdot c) \cdot (1 \cdot (a \cdot b)) =_F (a \cdot a) \cdot (b \cdot c) \tag{1}$$

where a, b, c are arbitrary elements of F. To prove this equation in a theorem prover each of the properties (i)–(iii) above has to be used (several times). It is possible to write a 'tactic' in the theorem prover that does just that:

> Apply each of the steps (i)–(iii) to rewrite the left and right hand side of equation (1) until the two sides of the equation are literally the same.

Obviously this is not a very smart tactic (e.g. it does not terminate when the equality does not hold) and of course we can do better than this by applying (i)–(iii) in a clever order. For the case of Abelian monoids, this can be done by rewriting all terms into a normal form which has the shape

$$a_1 \cdot (a_2 \cdot (\ldots \cdot (a_n \cdot 1) \ldots))$$

where $n \geq 0$ and $a_1, \ldots, a_n$ are elements of F that can not be decomposed, listed in alphabetic order. So a_i may be a variable of type F or some other term of type F, that is not of the form $- \cdot -$ or 1. A tactic, which is written in the meta language, has access to the code of a_i, hence it can order the a_i according to some pre-defined total order, say the lexicographic one. (Note that a normal form as above can not be achieved via a term rewrite system, because we have to order the variables.) So, a more clever tactic does the following.

> Rewrite the left and right hand side of equation (1) to normal form and check if the two sides of the equation are literally the same.

Following [5], there are three ways to augment the theorem prover with this proof technique for equational reasoning.

1. Add it to the primitives of the theorem prover,
2. Write (in the meta language) a tactic, built up from basic primitive steps, that performs the normalization and checks the equality.
3. Write a normalization function in the language of the theorem prover itself and prove it correct inside the theorem prover; use this as the core of the tactic.

The first is obviously undesirable in general, as it gives no guarantee that the method is correct (one could add any primitive rule one likes). The second and third both have their own pros and cons, which are discussed extensively in [5]. It is our experience (and of others, see [2]) that especially for theorem provers based on type theory, the third method is the most convenient one if one wants to verify a large numbers of problems from one and the same class. We will motivate why.

Reflection in type theory

We still work with the Abelian monoid $\langle F, \cdot, 1\rangle$ from before and we want to verify equation (1). The equality on this monoid will be denoted by $=_F$, which may be user defined or not, as long as it is an equivalence relation and a congruence for $\cdot$. Note that there is also the definitional equality, built-in into Coq. This is usually denoted as $=_{\beta\delta\iota}$, as it is generated from the literal (α-) equality by adding the computation steps β, δ (for unfolding definitions) and ι (for recursion). Definitional equality is decidable and built into the type checker; it is included in the equality $=_F$ (if two terms are definitionally equal, they are equal in any respect).

We introduce an inductive type of *syntactic expressions*, E, by

$$E ::= V \mid C \mid E * E$$

where V is the type of variables, let's take

$$V ::= \mathbb{N}$$

and C is the type of constant expressions, containing in this case just one element, `u`. In type theory (using Coq syntax) the definition of V and E would be as follows.

```
Definition V : Set := nat.

Inductive E : Set :=
   evar  : V->E
 | eone  : E
 | emult : E->E->E.
```

To define the semantics of an expression $e : E$, we need a valuation $\rho : V \to F$ to assign a value to the variables. The interpretation function connecting the level of the syntactic expressions E and the semantics F is then defined as usual by recursion over the expression.

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

In Coq syntax the interpretation function `I` is defined as follows, given the Abelian monoid `<F, fmult, fone>`:

```
Variable rho : V->F.

Fixpoint I [e:E] : F :=
 Cases e of
    (evar v)      => (rho v)
 |  eone          => fone
 |  (emult e1 e2) => (fmult (I e1) (I e2))
 end.
```

Now we write a 'normalization function':

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

that sorts variables, removes the unit (apart from the tail position) and associates brackets to the left. We don't give its encoding `N : E -> E` in Coq, but give the following examples.

$$\mathcal{N}((v_0 * \mathtt{u}) * (v_1 * v_2)) =_{\beta\delta\iota} (v_0 * (v_1 * (v_2 * \mathtt{u}))),$$
$$\mathcal{N}((v_2 * v_0) * v_1) =_{\beta\delta\iota} (v_0 * (v_1 * (v_2 * \mathtt{u}))).$$

The equality $=_{\beta\delta\iota}$ is the internal (computational) equality of the theorem prover: no proof is required for its verification; a verification of such an equality is performed by the type checker.

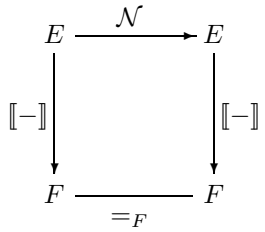
We prove the following key lemma for the normalization function.

$$\mathit{normcorrect}: \quad [\![e]\!]_\rho =_F [\![\mathcal{N}(e)]\!]_\rho$$

In Coq terminology: we construct a proof term

```
normcorrect : (rho: V -> F)(e:E)((I rho e) = (I rho (N e))).
```

The situation is depicted in the following diagram; *normcorrect* states that the diagram commutes.



Solving equation $f =_F f'$ with f and f' elements of F now amounts to the following.

- Find (*by tactic*) e, e' and ρ with

$$[\![e]\!]_\rho =_{\beta\delta\iota} f \text{ and } [\![e']\!]_\rho =_{\beta\delta\iota} f'$$

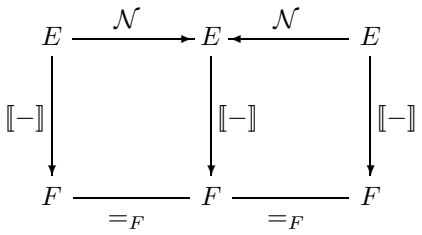
- Check (*by type checker*) whether

$$\mathcal{N}(e) =_{\beta\delta\iota} \mathcal{N}(e')$$

The proof of $f =_F f'$ is then found by

$$f =_{\beta\delta\iota} [\![e]\!]_\rho =_F [\![\mathcal{N}(e)]\!]_\rho =_{\beta\delta\iota} [\![\mathcal{N}(e')]\!]_\rho =_F [\![e']\!]_\rho =_{\beta\delta\iota} f'$$

from *normcorrect* for e and e' and *trans* of $=_F$. In a diagram:



In Coq this means that we have to construct a proof term of type

```
f = f'
```

This is done from `normcorrect` using the proofs of symmetry and transitivity of $=_F$, `sym` and `trans`.

```
sym   : (x,y:F)   (x = y) -> (y = x).
trans : (x,y,z:F) (x = y) -> (y = z) -> (x = z).
```

The crucial point is that

```
(normcorrect rho e)     : ((I rho e)  = (I rho (N e))).
(normcorrect rho e')    : ((I rho e') = (I rho (N e'))).
```

can *only* be fitted together using `trans`, when `(N e)` and `(N e')` are $\beta\delta\iota$-convertible. In that case we find that `(I rho (N e))` is $\beta\delta\iota$-convertible with `(I rho (N e'))` as well, so if we call that `g` by defining:

```
g := (I rho (N e))
```

then we find that:

```
(normcorrect rho e)     : (f = g).
(normcorrect rho e')    : (f' = g).
```

So using this, we can construct a proof term

```
(trans f g f' (normcorrect rho e) (sym f' g (normcorrect rho e')))
: f = f'.
```

The important points to note here are

(1) This proof term of an equality has a relatively small size, compared to a proof term that is spelled out completely in terms of congruence (of $=_F$ w.r.t. $\cdot$) and reflexivity, symmetry and transitivity (of $=_F$). The terms `refl`, `sym`, `trans`, and `normcorr` are just defined constants. The terms `rho`, `e` and `e'` are generated by the tactic; `rho` being of size linear in `f` and `f'` with a rather small constant. A proof term that is completely spelled out has a polynomial size in `f` and `f'`.

If we unfold the definitions, we observe that the bulk of the proof term is in `normcorr`. This will be rather large but it only has to be extended with a part

of – roughly – the size of the input elements themselves. So, then the proof term is still linear in the size of the input terms.

(2) *Checking* this proof term (i.e. verifying whether it has the type `f = f'`) can in general take rather long. This is because type checking now involves serious computation, as we use the language of the theorem prover as a small programming language. The bulk of the work for the type checker is in verifying whether `(N e)` and `(N e')` are $\beta\delta\iota$-convertible.

We compare this to the approach of using a tactic that is written completely in the meta laguage. This tactic will do roughly the same thing as our reflection method: reduce expressions to normal form and generate step by step a proof term that verifies that this reduction is correct. Checking such a proof term will take about the same time. Some increase in speed may only be gained if we check a *user generated* proof term, because this will (in general) avoid reducing to full normal form (assuming the user sees the possible 'shortcuts').

(3) *Generating* the proof term is very easy, both for the reflection method as for the tactic written in the meta language. The tactics generate the full proof term without further interaction. Note that a completely user generated proof term of an equality (which may be fastest to type check, see above), is not realistic.

Here we also see why the reflection approach is particularly appealing for theorem provers based on type theory: one has to construct a proof term, which remains relatively small using reflection. Moreover, these theorem provers provide the required programming language to encode the normalization and interpretation functions in.

Looking back at the example from the beginning, encoded in Coq, we have as goal

```
Goal
((fmult (fmult a c) (fmult fone (fmult a b)))
                                = (fmult (fmult a a) (fmult b c))).
```

Now the tactic generates

```
(emult (emult (evar 0) (evar 2))
       (emult eone (emult (evar 0) (evar 1)))).
(* the e : E * )
(emult (emult (evar 0) (evar 0)) (emult (evar 1) (evar 2))).
(* the e' : E * )
```

and a function `rho : V -> F` which is defined in such a way that

```
(rho (evar 0)) = a
(rho (evar 1)) = b
(rho (evar 2)) = c
```

Then it constructs a term as above,

```
(trans f g f' (normcorrect rho e) (sym f' g (normcorrect rho e')))
```

where `g` is `(I rho (N e))`. Note that `(I rho (N e))` $=_{\beta\delta\iota}$ `(I rho (N e'))` $=_{\beta\delta\iota}$

```
(I rho (emult (evar 0) (emult (evar 0)
          (emult (evar 1) (emult (evar 2) eone)))))
```

$=_{\beta\delta\iota}$ `(fmult a (fmult a (fmult b (fmult c fone))))`. This term is given to the type checker. If it type checks with as type the goal, the tactic succeeds (and it has constructed a proof term proving the goal); if the type check fails, the tactic fails.

3 Reflection With Partial Operations

We explain partial reflection by adapting the example to include division. We view division as a ternary operation:

$$a \div b \,//\, p \text{ with } p \text{ a proof of } b \neq_F 0.$$

This is very much a type theoretic view. One may alternatively write

$$a \div b \text{ for } b \in \{z \mid z \neq_F 0\},$$

but note that this also requires a proof of $b \neq_F 0$, before $\div$ can be applied to it.

As a side remark, we note that we use the principle of *irrelevance of proofs* when extending the equality on F to expressions of the form $a \div b \,//\, p$. That is, if p and p' are both proofs of $b \neq_F 0$, then $(a \div b \,//\, p) =_F (a \div b \,//\, p')$. In our encoding in Coq, this is achieved by representing $\{z \mid z \neq_F 0\}$ by the type Pos of pairs $\langle b, p\rangle$ with $p : (b \neq_F 0)$ *with the equality on* Pos *the one inherited from* F. Then we let $\div$ be a function from $F \times \mathsf{Pos}$ to F.

If we extend our structure with a zero element and a division operator, like in fields, we encounter the problem of undefined elements. These cause trouble in various places. First of all, there is the question of which syntactic expression one allows: if $1/0$ is accepted, which interpretation does it have (one has to choose one). This is of course related to the question whether the theorem prover allows to write down $\frac{a}{0}$ (whatever its meaning may be). The second problem is that a naive normalization function might rewrite $0/(0/v)$ to v (just because $x/(x/v) = x/x*v = 1*v = v$). But then, $\frac{0}{\frac{0}{a}} = a$, which is undesirable. Note that the 'division by 0' problem can occur in a more disguised form, e.g. in $\frac{y}{\frac{y}{a}} = a$, with y a variable, which is correct under the *side-condition* that $y \neq_F 0$. So, it seems that, when normalizing an expression e, one would have to take the *interpretation* $[\![e]\!]_\rho$ into account (and the interpretation of subexpressions of e) to verify that the normalization steps are correct.

We have solved the problems just mentioned by

- Allowing syntactic expressions (like $1/0$) that have no interpretation. So $[\![-]\!]_\rho$ is defined as a relation, for which it has to be *proved* that it is a partial function.
- Writing the normalization function N in such a way that, if expression e has an interpretation, then expression $N(e)$ has the same interpretation as e.

Syntactic expressions We now define the inductive type of *syntactic expressions*, E, by

$$E ::= V \mid C \mid E * E \mid E/E$$

where V is again the type of variables, for which we take $V ::= \mathbb{N}$ again. C is the type of constant expressions, now containing a zero, `z`, and a one expression, `u`. In type theory (using Coq syntax):

```
Inductive E : Set :=
   evar  : V->E
 | eone  : E
 | ezero : E
 | emult : E->E->E
 | ediv  : E->E->E.
```

Note that E doesn't depend on F and ρ; we have 'light' syntactic expressions (without any semantic information). This implies that $1/0$ is allowed in E: it is a well-formed expression.

Interpretation relation The semantics of an expression is now not given by a function but an *interpretation relation*:

$$[\![]\!]_\rho \subseteq E \times F$$

Again, we need a valuation $\rho : V \to F$ to assign a value to the variables. The interpretation relation can then be defined inductively as follows.

$$\begin{aligned}
v_n \,[\![]\!]_\rho\, f &\text{ iff } \rho(n) =_F f,\\
\mathtt{u} \,[\![]\!]_\rho\, f &\text{ iff } f =_F 1,\\
\mathtt{z} \,[\![]\!]_\rho\, f &\text{ iff } f =_F 0,\\
(e_1 * e_2) \,[\![]\!]_\rho\, f &\text{ iff } \exists f_1, f_2 \in F\ (e_1 \,[\![]\!]_\rho\, f_1) \wedge (e_2 \,[\![]\!]_\rho\, f_2) \wedge (f =_F f_1 \cdot f_2),\\
(e_1/e_2) \,[\![]\!]_\rho\, f &\text{ iff } \exists f_1, f_2 \in F\ (e_1 \,[\![]\!]_\rho\, f_1) \wedge (e_2 \,[\![]\!]_\rho\, f_2) \wedge (f_2 \neq_F 0) \wedge (f =_F f_1 \div f_2).
\end{aligned}$$

In Coq let there be given a structure `<F, fmult, fdiv, fone, fzero>`, with

```
fdiv: (x,y:F)(~(y =_F fzero))->F
```

and the other operations and the equality as expected. The inductive definition of $[\![]\!]_\rho$ is as follows.

```
Inductive I : E->F->Prop :=
   ivar  : (n:V)(f:F) ((rho n) = f) -> (I (evar n) f)
 | ione  : (f:F)      (fone = f)    -> (I eone f)
 | izero : (f:F)      (fzero = f)   -> (I ezero f)
 | imult : (e,e':E)(f,f',f'':F)
   ((fmult f f') = f'') -> (I e f) -> (I e' f')
```

```
                                          -> (I (emult e e') f'')
| idiv  : (e,e':E)(f,f',f'':F)(nz:~(f' = fzero))
  ((fdiv f f' nz) = f'') -> (I e f) -> (I e' f')
                                          -> (I (ediv e e') f'').
```

Note that we do not just let `ione : (I eone fone)`, but take `fone` modulo the equality on `F`, and similarly for the constant, the variables and the two operators. This is because `I` should be a partial function *modulo* the equality on `F`. In more technical terms: correctness of normalization can only be proved with this version of `I`.

Normalization and correctness The 'normalization function':

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

now brings the expressions that have an interpretation in one of the following two normal forms

$$(v_1 * (v_2 * \ldots (v_n * \mathsf{u}) \ldots)) \,/\, (w_1 * (w_2 * \ldots (w_m * \mathsf{u}) \ldots)),$$
$$\mathsf{z} \,/\, \mathsf{u},$$

with $v_1, \ldots, v_n, w_1, \ldots w_m$ variables and the two lists $v_1, \ldots, v_n$ and $w_1, \ldots w_m$ disjoint. So, $\mathcal{N}$ creates two mutually exclusive lists of sorted variables, one representing the enumerator and one representing the denominator. The sorting of these lists is the same as for multiplicative expressions. In case $\mathcal{N}$ encounters a z in the enumerator, the whole expression is replaced by z/u (which has interpretation 0). For the expressions that do not have an interpretation (those $e \in E$ for which there are no $\rho : V \to F, f \in F$ with $e \,[\![_\rho\, f$), the normalization function can return anything.

We don't give the encoding `N : E -> E` in Coq, but restrict ourselves to some examples.

$$\mathcal{N}(v_0/(v_1/v_3)) * v_1 =_{\beta\delta\iota} (v_0 * (v_3 * \mathsf{u}))/\mathsf{u},$$
$$\mathcal{N}((v_0/(v_1 * v_2))/(v_3/v_2)) =_{\beta\delta\iota} (v_0 * \mathsf{u})/(v_1 * (v_3 * \mathsf{u})).$$

We can understand the way $\mathcal{N}$ actually works as follows.

1. From an expression e, two sequences of variables and constants are created s_1 and s_2, the first representing the enumerator and the second the denominator. The intention is that, if e has an interpretation, then s_1/s_2 has the same interpretation.
2. These two sequences are put in normal form, following the normalization procedure for multiplicative expressions.
3. Variables that occur both in s_1 and s_2 are canceled, units are removed and s_1 is replaced by z if it contains a z.

Note that we tacitly identify a sequence s_1 with the expression that arises from consecutively applying $*$ to all its components. This is also the way we have implemented it in Coq: we do not use a separate list data structure, but encode it via $*$ and u. On these lists, we define an 'append' operation, which we denote by @. So, if s_1 and s_2 denote two expressions in multiplicative normal form, $s_1@s_2$ is the multiplicative normal form of $s_1 * s_2$. As a matter of fact, $\mathcal{N}$ doesn't do each of these steps sequentially, but in a slightly smarter (and faster) way.

In proving the correctness of $\mathcal{N}$, one has to preserve the property that all denominators are $\neq_F 0$. In that, the first step is the crucial one. (The second step is only a reordering of variables; one has to prove that this reordering preserves the $\neq_F 0$ property, which is easy. In the third step one has to prove that $\neq_F 0$ is preserved under cancellation, which is the case: if $a \cdot b \neq_F 0$, then $a \neq_F 0$.) The first step has a nice recursion: if $\mathcal{N}(e) = (s_1, s_2)$ and $N(e') = (s_1', s_2')$, then

$$\begin{aligned} \mathcal{N}(e * e') &:= (s_1@s_1', s_2@s_2'), \\ \mathcal{N}(e/e') &:= (s_1@s_2', s_2@s_1'). \end{aligned}$$

Now, if $e * e'$ has an interpretation, then (by induction) s_2 and s_2' have an interpretation different from 0 and hence the interpretation of $s_2@s_2'$ is different from 0. Similarly, if e/e' has an interpretation, then (by induction) s_2, s_1' and s_2' have an interpretation different from 0 and hence the interpretation of $s_2@s_1'$ is different from 0.

This is also how the correctness proof of $\mathcal{N}$ works: $\mathcal{N}$ itself doesn't have to bother about the interpretation of the expressions it operates on, because it is written in such a way that, the fact that e has an interpretation implies (in a rather simple way, sketched above) that $\mathcal{N}(e)$ has an interpretation (which is the same as for e).

Again we note that $\mathcal{N}$ cannot be found as a term rewriting system, for one because it orders variables, but more importantly because it only works properly for expressions that have an interpretation. We can use this information, because the expression we start from is derived from an existing $f : F$, which is well-defined (otherwise we couldn't write it down in the theorem prover). So, we already know that the first e has an interpretation (namely f) and by virtue of the construction of $\mathcal{N}$, this property is preserved.

We prove the following key lemmas.

$$\begin{aligned} \textit{normcorrect}: &\quad e \,[\![_\rho\, f &&\Rightarrow \mathcal{N}(e) \,[\![_\rho\, f \\ \textit{extensionality}: &\quad (e \,[\![_\rho\, f) \wedge (e \,[\![_\rho\, f') &&\Rightarrow f =_F f'. \end{aligned}$$

Extensionality states that $[\![_\rho$ is really a partial function (w.r.t. the equality $=_F$).

Reflection The reflection method for solving $f =_F f'$ is now:

- find (*by tactic*) e, e' and ρ with

$$e \,[\![_\rho\, f \text{ and } e' \,[\![_\rho\, f'$$

– construct (*see below*) proof terms for these two statements
– check (*by type checker*) whether

$$\mathcal{N}(e) =_{\beta\delta\iota} \mathcal{N}(e')$$

($=_{\beta\delta\iota}$ means $\beta\delta\iota$-*convertible*)

The proof of $f =_F f'$ is then found by:

$$\left.\begin{array}{ll} e \,[\![_\rho\, f & \Rightarrow \mathcal{N}(e) \,[\![_\rho\, f \\ e' \,[\![_\rho\, f' & \Rightarrow \mathcal{N}(e') \,[\![_\rho\, f' \end{array}\right\} \Rightarrow f = f'$$

from *normcorrect* (applied to (e, f) and (e', f'), respectively) and *extensionality* (applied to $(\mathcal{N}(e), f, f')$).

Just as in the case for reflection in Section 2, a precise proof term can be constructed, which type checks with type $f =_F f'$ if and only if these terms are can be shown to be equal in the equational theory. In the next Section we will exhibit such a proof term. The main work in type checking this proof term lies in the execution of the algorithm $\mathcal{N}$ (but this is done by the type checker).

One problem remains. As we now have an interpretation *relation*, there arise some *proof obligations*: it is not just enough to *find* encodings e and e' of f and f'; we have to *prove* that they are encodings indeed. That is, we have as new goals

$$e \,[\![_\rho\, f \text{ and } e' \,[\![_\rho\, f'$$

Of course, we don't want the user to have to take care of these goals; the tactic should solve them. This problem is dealt with in the next Section.

4 Proof Loaded Syntactic Objects

At the second step of the partial reflection method, we need proofs of $e \,[\![_\rho\, f$. One way is to let the tactic construct these; so from $f : F$, the tactic extracts both $e : E$ and ρ and a proof term p with $p : e \,[\![_\rho\, f$. This is possible, but it is not what has been implemented. We have chosen to have one data type for both expressions and proofs. The strategy for doing so (and which fits very well with the type theoretic approach) is to create *syntactic expressions with proof objects inside*

$$\bar{E}$$

with a *forgetful function* $|-|$ and an *interpretation function* $[\![-]\!]_\rho$,

$$\begin{array}{c} |-| : \bar{E} \to E \\ [\![-]\!]_\rho : \bar{E} \to F \end{array}$$

The key property to be proved is then

$$|\bar{e}| \,[\![_\rho\, [\![\bar{e}]\!]_\rho$$

But note that $\bar{E}$ depends on F and ρ (it should 'know' about semantics), so $\bar{E}$ is a type of 'heavy' syntactic expressions (including proof terms). This can only work if we let $\bar{E}$ be a *dependent* type over F:

$$\bar{E}_f$$

which in Coq terms is defined as:

```
Inductive xE : F -> Set :=
   xevar  : (i:V)(xE (rho i))
 | xeone  : (xE fone)
 | xezero : (xE fzero)
 | xemult : (f,f':F)(e:(xE f))(e':(xE f'))(xE (fmult f f'))
 | xediv  : (f,f':F)(e:(xE f))(e':(xE f'))(nz:~(f' = fzero))
              (xE (fdiv f f' nz)).
```

The type $\bar{E}_f$ represents the type of 'heavy' syntactic expressions whose interpretation is f. The interpretation function is now

$$[\![-]\!]_\rho : \bar{E}_f \to F$$

for which it should hold that

$$[\![\bar{e}]\!]_\rho =_{\beta\delta\iota} f$$

so $[\![-]\!]_\rho$ is constant on its domain. In Coq terms we define:

```
xI := [f:F][e:(xE f)]f  : (f:F)(xE f) -> F.
```

Note that we do not define the interpretation by induction on `e` : (`xEf`), but we just return f (the *intended* interpretation). The obligation is now to prove that the underlying 'light' syntactic expression has indeed f as interpretation. The forgetful function, extracting the 'light' syntactic expression, now is

$$|-| : \bar{E}_f \to E$$

It maps the 'heavy' syntactic expressions to the 'light' ones. In Coq terms:

```
Fixpoint xX [f:F; e:(xE f)] : E :=
  Cases e of
    (xevar i)               => (evar i)
  | xeone                   => eone
  | xezero                  => ezero
  | (xemult f' f'' e' e'')  => (emult (xX f' e') (xX f'' e''))
  | (xediv f' f'' e' e'' p) => (ediv (xX f' e') (xX f'' e''))
  end.
```

which is defined by induction over `(xE f)`. The maps $[\![-]\!]_\rho$ and $|-|$ 'extract' the two components (syntactic expression and semantic element) from the 'heavy' encoding. The key result now is that the second extraction is an interpretation of the first:

$$\mathit{extractcorrect}: \quad \forall x \in \bar{E}_f(|x|\,[\![x]\!]_\rho\,[\![x]\!]_\rho)$$

which is just $\forall x \in \bar{E}_f(|x|\,[\![x]\!]_\rho\, f)$.

The tactic now works as follows, given a problem $f =_F f'$.

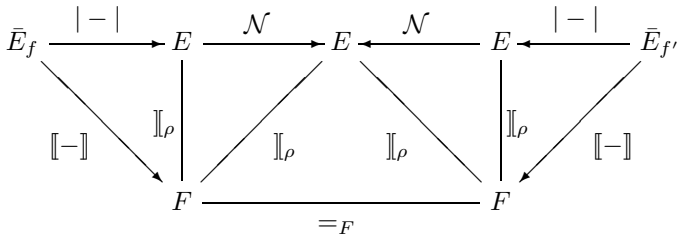
– find (*by tactic*) $\bar{e} \in \bar{E}_f$, $\bar{e}' \in \bar{E}_{f'}$ and ρ with

$$|\bar{e}| \, [\![]_\rho \, f \text{ and } |\bar{e}'| \, [\![]_\rho \, f'$$

– obtain (*from extractcorrect*) proof terms for these two statements
– check (*by type checker*) whether

$$\mathcal{N}(|\bar{e}|) =_{\beta\delta\iota} \mathcal{N}(|\bar{e}'|)$$

So, the tactic creates e, e' of type E indirectly by creating $\bar{e}, \bar{e}'$ of types $\bar{E}_f, \bar{E}_{f'}$. In a diagram the situation is now as follows.



The outside triangles commute due to *extractcorrect*; the large middle triangle commutes due to *extensionality*; the other two triangles commute due to *normcorrect*. If we make the proof term given by this method explicit, it is

```
(extensionality rho ne f f'
    (normcorrect rho e  f  (extractcorrect rho f  xe ))
    (normcorrect rho e' f' (extractcorrect rho f' xe')))
: f = f'
```

where `xe` and `xe'` correspond to $\bar{e}$ and $\bar{e}'$, and where we have defined

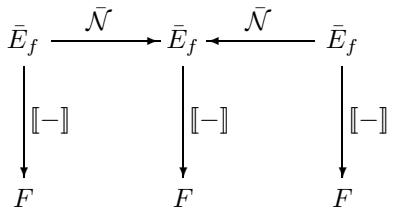
```
e  := (xX f  xe).
e' := (xX f' xe').
ne := (N e).
```

This term is only well-typed when `(N e)` is $\beta\delta\iota$-convertible with `(N e')`.

Normalizing Proof Loaded Objects

In presence of the type $\bar{E}_f$, we could do without the type E all-together. Then we would define a normalization function $\bar{\mathcal{N}}$ to operate on the 'heavy' syntactic expressions of type $\bar{E}_f$. This is possible (and it yields a simpler diagram), but it is not desirable, because then the computation (reducing $\bar{\mathcal{N}}(\bar{e})$ to normal form) becomes much heavier. Moreover, it would be more difficult to program $\bar{\mathcal{N}}$ (having to take all the proof terms into account) and the two levels in the reflection approach would be less visible, therefore slightly blurring the exposition.

Nevertheless, for reasons of completeness we have also constructed (see [4]) the function $\bar{\mathcal{N}}$ together with proofs that it is correct. Ideally, this would amount to the following diagram



However, $\bar{\mathcal{N}}$ can not have the dependent type $\bar{E}_f \to \bar{E}_f$ (for $f : F$), because the value (in F) of the output of the normalization function is not *literally* the same as its input value, but only *provably* equal to it. So, we can *not* construct $\bar{\mathcal{N}}$ as a term `xN : (z:F) (xE z) -> (xE z)`. Instead we construct `xN : fE -> fE`, where `fE` is the type of pairs `< f, e >`, with `f : F` and `e : (xE f)`. (In type theoretic terms, this is the *Σ-type* of *dependent pairs* $\langle f, e\rangle$ with $f : F$ and $e : \bar{E}_f$.) Then we have to prove that if $\bar{\mathcal{N}}(\langle f, e\rangle)$ yields $\langle f', e'\rangle$, then f and f' are (provably) equal in F.

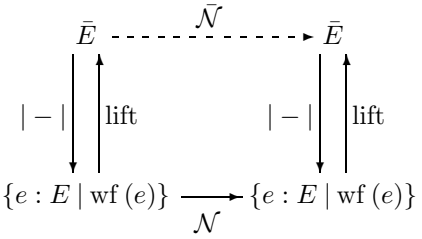
If we cast this in purely mathematical terms, the situation is as follows. Define $\bar{E} := \Sigma f{:}F.\bar{E}_f$ and let wf be the predicate on syntactic expressions stating that it has an interpretation (it is *well-formed*). It is defined as follows (for $e : E$).

$$\mathrm{wf}\,(e) := \exists f{:}F(e \,[\![_\rho\, f).$$

Now there are maps $\mathrm{lift} : \{e : E \mid \mathrm{wf}\,(e)\} \to \bar{E}$ and $|-| : \bar{E} \to \{e : E \mid \mathrm{wf}\,(e)\}$. Furthermore, we can construct a proof-object

$$\mathit{normwf} \quad : \quad \forall e{:}E(\mathrm{wf}\,(e) \to \mathrm{wf}\,(\mathcal{N}(e)))\}.$$

Then we can read off the normalization function $\bar{N} : \bar{E} \to \bar{E}$ from the following diagram.



The proof term *normwf* shows that $\mathcal{N}$ is indeed a function from the set of well-formed expressions to itself. The correctness of $\bar{\mathcal{N}}$ is given by

$$\mathit{normcorrect} \quad : \quad \forall \bar{e}{:}\bar{E}([\![\bar{e}]\!] =_F [\![\bar{\mathcal{N}}(\bar{e})]\!]).$$

Here $[\![-]\!] : \bar{E} \to F$ is the interpretation function mapping (heavy) syntactic expressions to elements of F. (As a matter of fact, it is just the first projection.)

5 Partial Reflection in Practice

The approach of partial reflection is successfully used in our current FTA project (Fundamental Theorem of Algebra). First of all, we have a tactic called `Rational` for proving equalities. This tactic is implemented as outlined above.

But often we do not just want to prove an equality, but rather to use an equality to *rewrite* a goal in a different form. In order to explain how we have implemented rewrite tactics, we first say something about the equality in the FTA project. Our equality is just a congruence relation, respected by operations (such as `+` and `*`) and certain predicates (such as `<`). This means we cannot just replace equals by equals in *any* expression, but only those built-up from terms respecting our equality. (This stands in contrast to the standard Leibniz-equality in Coq; Leibniz-equals may be replaced in any proposition.) For instance, we have the following lemma:

```
less_wd_left : (a,b,c:F)(a=b) -> (b<c) -> (a<c).
```

Hence, we have defined rewriting tactics for each important predicate that respects our equality. For instance, the tactic `Step_less_left t` applies to a goal `p<q`: it lets `Rational` solve the equation `t=p` and returns the new goal `t<q`. It is defined for each `t` as

```
(Apply less_wd_left with b:=t) ;
[ Rational | (* Use Rational tactic to prove equality *)
  Idtac ]    (* Do nothing with new inequality *)
```

The following example illustrates its use. (Note that `1/z//H2` denotes `1` divided by `z` with as proof of the side condition `z#0` – $z \neq_F 0$ – the variable `H2`.)

```
  H1 : 0 < z
  H2 : z # 0
  H3 : x*z < y*z
  ==============
  x < y
< Step_less_left x*z*(1/z//H2)
  H1 : 0 < z
  H2 : z # 0
  H3 : x*z < y*z
  ==============
  x*z*(1/z//H2) < y
```

6 Conclusion

We have extended the reflection method to include partial functions. The power of the method lies in the fact that no new proof obligations arise. So, if the user wants to prove a simple equation involving partial functions, the system does not

(have to) generate a new set of goals (in order to prove that all partiality side conditions are fulfilled). That the necessary side conditions are fulfilled is already proven by the correctness of the normalization function. Phrased differently: normalization preserves well-definedness. The other crucial point is the fact that, although some syntactic expressions may be undefined, the ones that our tactic generates never are, for the simple reason that they are encodings of *well-defined* semantic objects in the theorem prover. So, the normalization function starts off from a syntactic expression that is well-defined (for the simple reason that the semantic object is its interpretation) and the well-definedness is preserved under normalization.

As a side remark, we point out that the fact that *the encoding always yields a well-defined syntactic expression* is a statement on the meta-level. As the encoding function is a meta-function we can not expect to state this literally in the theorem prover. We can state $\forall f : F \exists e : E \exists \rho (e \, [\![_\rho \, f)$, but this does not capture what we want to say: it is trivially true, taking a variable v for e and $\rho(v) = f$, and it does not say anything about the encoding function.

The actual implementation of the method as a tactic for solving equations between field elements has shown that this is a very useful technique. We believe it is very generally applicable in situations where partiality occurs.

Acknowledgements We thank Henk Barendregt for the many useful discussions on the subject and the anonymous referees for their comments on the paper.

References

1. G. Barthe, M. Ruys and H. Barendregt (1996), A Two-Level Approach towards lean Proof-Checking.
2. S. Boutin, Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, TACS'97, volume 1281. LNCS, Springer-Verlag, 1997.
3. G. Huet et al. (1997), The Coq Proof Assistant, Reference Manual, Version 6.1, INRIA-Rocquencourt — CNRS-ENS Lyon.
4. The "Fundamental Theorem of Algebra" Project, Departmet of Computer Science, University of Nijmegen, the Netherlands. See `http://www.cs.kun.nl/gi/projects/fta/`
5. J. Harrison (1995), Meta theory and Reflection in Theorem Proving: a Survey and Critique, Technical Report CRC-053, SRI International Cambridge Computer Science Research Center.
6. D. Howe (1988) Computational Meta theory in Nuprl, The Proceedings of the Ninth International Conference of Automated Deduction, eds. E. Lusk and R. Overbeek, LNCS 310, pp. 238–257.
7. M. Oostdijk and H. Geuvers (2000), Proof by Computation in Coq, to appear in TCS.