Using reflection to build efficient and certified decision procedures

Samuel Boutin
Samuel.Boutin@inria.fr

January 9, 1997

Abstract

In this paper we explain how computational reflection can help build efficient certified decision procedure in reduction systems. We have developped a decision procedure on abelian rings in the Coq system but the approach we describe applies to all reduction systems that allow the definition of concrete types (or datatypes). We show that computational reflection is more efficient than an LCF-like approach to implement decision procedures in a reduction system. We discuss the concept of total reflection, which we have investigated in Coq using two facts: the extraction process available in Coq and the fact that the implementation language of the Coq system can be considered as a sublanguage of Coq. Total reflection is not yet implemented in Coq but we can test its performance as the extraction process is effective. Both reflection and total reflection are conservative extensions of the reduction system in which they are used. We also discuss performance and related approaches. In the paper,we assume basic knowledges of ML and proof-checkers.

1 Introduction

Aim of the paper

In this paper we show how to build efficient, certified decision procedures in a broad range of applications. Computer algebra systems attempt to answer the problem of efficient symbolic computation but ignore the goal of reliability. The same aim is followed by systems allowing the checking of tautologies using a broad range of techniques among which we find binary decision diagrams. On the other hand, a large number of systems try to prove the reliability of programs and mathematical results but seem to be too weak to be used for real world examples. This paper addresses the question of how we could fill the gap between these communities with a technique that yield both efficency and high level of reliability. We show how computational reflection in reduction systems seems to be a valuable answer to the problem of writing efficient and reliable decision procedures.
As a concrete application, we have implemented a certified decision procedure on the first order theory of abelian rings in the Coq proof developpement system, but no previous knowledge of Coq is required to read this paper. The decision on the theory of abelian rings is at the same time a simple and powerful test. It is not far from decision on boolean rings allowing the checking of tautologies and it is implemented in computer algebra systems. It is also a non-trivial example as it addresses the question of dealing with associative commutative theories.
Systems like HOL or PVS already offer powerful means to define decision procedures. Concerning HOL for instance, we think that following our approach, it is possible to build decision procedures at least an order of magnitude faster with the same level of reliability and in a quite easy manner.

Overview

In the second section we recall basic notions concerning theorem-provers and explain basic means of comparison between a wide range of provers. The third section addresses the usual mechanism of goal-directed theorem proving using tactics and tacticals and explains why this approach is problematic

in systems which keep proof-objects. The fourth section addresses the mechanism of computational reflection in the Coq theorem prover but no particular knowledge about Coq is needed to read it. The fifth section addresses some hints about what we call total (or complete) reflection and the reliability problems it raises. The sixth section addresses the performance of the method and the seventh section discusses numerous related approaches concerning computational reflection.

2 Reduction systems and fully expansive provers

It seems possible to divide current theorem-proving systems into two broad categories. In the first category are automatic theorem-provers where a user can just tell the system to prove a new lemma given a bunch of already checked lemmas. So in such systems, the user has to guess the intermediate lemmas that the prover possibly needs to solve a problem. In these systems, the user has nearly no means to design the proof of the theorem. The point in such systems is the power of the decision procedures available in the system and the exact way the proof is gotten is irrelevant. Reliability in such system amounts to the reliability of all the decision procedures used to perform a proof. In order to extend dynamically the power of such a system, there ought to exist a mechanism to safely add new decision procedures. Such a mechanism exists for instance in the Boyer-Moore theorem prover [3] which belongs to the first category.

In the second category are computer aided proof-checkers, inheriting the LCF approach, where any object, proof or decision procedure has to split in a sequence of application of a finite set of primitive rules. In these systems, proofs can be constructed in a top-down manner by the application of *tactics*. In this second camp, we can still distinguish fully expansive theorem provers from reduction systems. This distinction concerns merely the implementation style but we believe that in fact it relies on a very different approach to the process of building proofs. We consider that the respective treatments of identity merely explain the divergences: there exists indecidable problems in mathematics so, a proof checker needs the user in the process of building a proof. Roughly, Fully Expensive Theorem Provers (FETP), always rely on extensional type theory so that identities managed by the prover can be undecidable and must be under the control of the user. Reduction systems rely on intensional type theory so that identity is decidable (and not under the user control) but weak so that this raises problems when dealing with extensional concepts but we do not discuss this topic in the paper.

Another way to say this is that, in FETPs, definitional identity (how the system identifies terms) is composed of renaming of variables and abreviation and the user proves identities through primitive **judgmental** equality which allows **derived** identities; while in reduction systems, definitional equality is extended with β and ι-reduction (the reduction of primitive recursive pattern matching) so that it is decidable and no judgmental equality is available. If the user wants to prove identities in a reduction system, he must use a defined identity and for instance it is possible to define an equality in the object language of Coq which behaves nearly like Leibniz equality.

Yet another way to explain this difference is to say that all the rewriting steps are **explicit** in a FETP while there are **implicit** proof steps in a reduction system (namely β and ι reductions). A side effect of this difference is that there exists means of computation in reduction systems, similar to usual functional languages while this is not the case in pure FETPs like LCF or HOL. The price for this is that the critical part of the code of a FETP consists of the implementation of the logic, while the critical part of the code of a reduction system must also include the code that implements the extended definitional equality.

Of course this description is simplistic but it sheds light over the differences between modern provers and allows to say in a few words the leitmotiv of the paper: *using computational reflection, we translate theorem proving into computation* and at the same time *we translate explicit proof steps into implicit ones* and this is possible because we use a reduction system: Coq.

The following simple examples illustrate what is an explicit proof step and what is an implicit one. Consider the following sequence performed in Coq:

```
============================
 (plus O O)=O

Unnamed_thm < Simpl;Apply refl_equal.
Subtree proved!
```

Let's roughly explain what happens in this small sequence. The function `plus` is defined by primitive recursion on its first argument so that `(plus O O)` computes to `O` in the definitional identity of the system. The symbol `=` stands for a user-defined relation which behaves roughly like Leibniz identity. To solve the goal `(plus O O)=O`, we first put it in normal form for definitional identity using the tactic `Simpl` and so we obtain the new goal `O = O`, but this is just an instance of the reflexivity of the `=` relation and we can conclude by applying the `refl_equal` axiom which asserts this reflexivity. We can ask for the proof generated by this sequence of tactics by using the Coq command `Show Proof`; it yields "`(refl_equal nat O)`". So the important point we show is that the rewrite of `(plus O O)` into `O` does not appear in the proof term: it is what we call an **implicit proof step** or **implicit rewriting step** or **computation** (the reduction of a primitive recursive pattern matching).
Consider now this other sequence:

```
============================
 (a,b,c:nat)a=b->b=c->a=c

Unnamed_thm < Intros a b c a_eq_b b_eq_c;Rewrite -> a_eq_b;Exact b_eq_c.
Subtree proved!
Unnamed_thm < Show Proof.
Proof: [a,b,c:nat][a_eq_b:a=b][b_eq_c:b=c](eq_ind_r nat b [n:nat]n=c a_eq_b a b_eq_c)
```

Here, the notation `(a,b,c:nat)a=b->b=c->a=c` means informally: "for all natural numbers `a`, `b`, `c`, `a=b` implies `b=c` implies `a=c`". We directly construct a sequence of tactics (see the following section for more details) which proves this goal. Using `Intros a b c a_eq_b b_eq_c`, we say that this goal can be proved if we are able to prove `a=c` in the context where `a`, `b`, `c` are integers and where we know that `a=b` and `b=c`. This operation is called the discharging of hypotheses and it also gives names to hypotheses. Here we call `a_eq_b` the hypothesis `a=b` for instance. With `Rewrite -> a_eq_b`, we rewrite a to b in the goal so that the new goal is `b=c` but this is an hypothesis and we conclude with `Exact b_eq_c`. We now ask for the proof term and obtain

```
[a,b,c:nat][a_eq_b:a=b][b_eq_c:b=c](eq_ind_r nat b [n:nat]n=c a_eq_b a b_eq_c)
```

The `[]`-abstractions correspond to discharged hypotheses and the rest of the proof term, `(eq_ind_r nat b [n:nat]n=c a_eq_b a b_eq_c)` corresponds to the rewrite of hypothesis `a_eq_b`. This term is an application of Leibniz principle. Without entering into details, just remark that to perform the rewrite of `a` into `b` on the goal `a=c`, we have to **build explicitely the predicate** `[n:nat]n=c`. More generally, the point is that such **an explicit rewrite requires to build a predicate of the size of the goal in a reduction system**.

3 The goal-directed LCF approach

For the sake of completeness we recall the usual way of defining decision procedures using an LCF like approach. We show that this approach does not fit in the case of reduction systems. As this was described through the two small examples of the previous section, tactics allow to construct the derivation of a theorem during an interactive session where the user of the system applies a tactic to a goal and then has to solve the subgoals generated by this application. Most of the type checkers support an LCF like set of tactics and tacticals; and this tools allow to define decision procedures. Here are the main tacticals available in the Coq system.

- `Repeat` T applies the tactic T on the current goal until T fails.
- T_1 `;` T_2 first applies T_1 then T_2 to the subgoals generated by the application of T_1.
- T_1 `Orelse` T_2 applies T_1 and if the application of T_1 fails, it applies T_2.
- `Try` T tries T and does nothing if T fails.
- T `;` `[`T_1`;`.....`;`T_n`]` applies T and then T_i to the i-th subgoal generated by T.
- expressive power of ML pattern matching

As an example of how this work, let's look at the problem of deciding identity on the first order theory of monoids. Suppose for instance that the type A [1] in Coq supports a monoid structure. This means that there exists constants `unit_A` and `mult_A` such that the following constants[2] are available:

```
assoc_mult_A : (x,y,z:A) (mult_A (mult_A x y) z) = (mult_A x (mult_A y z)).
neutral_one_A_left: (x:A) (mult_A one_A x) = x.
neutral_one_A_right: (x:A) (mult_A x one_A) = x.
```

"`(_:A)`" means "for all `_` in `A`". Now the problem is how to define a decision procedure that decide identities that are deducible from `assoc_mult_A`, `neutral_one_A_left`, and `neutral_one_A_right` only. To perform this, a simple solution is to put the two members of the identity in canonical form w.r.t the monoid structure; for instance a possible canonical form is the elimination of unnecessary occurences of `one_A`, and systematic association to the right. For more details about canonical forms w.r.t first order theories, refer to [13]. We can write a naive decision procedure on this theory using Coq, call it `Monoid_dec` exactly as it would be possible in HOL:

```
Repeat ((Rewrite -> neutral_one_A_left)
       Orelse
       (Rewrite -> neutral_one_A_right)
       Orelse
       (Rewrite -> neutral_one_A_left));
Auto.
```

The tactic `Rewrite ->t`, where `t` is a universally quantified identity rewrites all the occurences of the left member of `t` that the system is able to match into the correctly instanciated right member of `t`. Suppose now that you want to check an identity on the theory of monoids. Then apply the tactic above. Obviously, both left and right members will be put in canonical form as this tactic eliminates the three kind of "monoid redexes", and then the `Auto` ought to conclude if the goal is now an instance of the reflexivity of equality.
This seems a very simple way of defining decision procedures! In fact, in the context of a reduction system, it is also very inefficient. The size of proof terms in a reduction system, using this approach can be estimated: given a goal F, the size of the proof of F using decision procedure D is approximatively $size(F) * n(D, F)$ where $size(F)$ is the size of the formula F and $n(D, F)$ is the number of rewrites to prove F using D. This is directly related to our example about explicit proof steps at the end of section 2. This can be improved if, by some means, the decision procedure is able to perform local actions on the goal but anyway it remains essentially bad in the treatment of big problems. So this approach can be used for middle size problems involving at most a few hundreds of rewrites. In case of large problem solving we recommend the approach described in the next section.

[1] if you do not like the word type, call it a set
[2] if you do not like the constant, call it an axiom

4 Computational reflection in Coq

Following Harrison [8], "computational reflection principles do not extend the power of the logic, but may make deductions in it more efficient". We can be more precise here as what we do can be summarized as follows: rather than rewriting tactics as this was described in the previous section, we perform this rewriting with the definitional identity of the system. Another way to say this is we translate explicit proof steps (rewrites) into implicit ones. To do so, we need what we call metaification. We have implemented the decision on the theory of abelian rings using the method described here but for the sake of simplicity we use the theory of monoids as sample. We discuss the performance of this approach in a further section.
In the previous section we introduced a type `A` and a set of constants that fit `A` with the structure of monoid. In this section, we still want to decide identites on the theory of monoids that we call the target theory. Our general approach to the building of a reflection tactical deciding for a first order theory is as follows (we illustrate the general scheme using the case of monoids):

1. define an **inductive type** [16] the constructors of which are the constants of the target theory. In the case of monoids this corresponds in Coq to the following inductive definition:

   ```
   Inductive Set monoid :=
     r1 : monoid
   | rmult : monoid -> monoid -> monoid
   | atom : nat -> monoid.
   ```

   This definition is exactly like the definition of a concrete types [3] in ML. So `r1` and `rmult` are inductive constructors corresponding to constants `one_A` and `mult_A`. The extra constructor `atom` is used to encode variables corresponding to objects of type `A` which are neither `one_A` nor `mult_A` and its use will be cleared later on. We call `atom`, `r1` and `rmult` the signature of the source theory, and `monoid` is the source theory.

2. define a translation of the source theory into the target theory by primitive recursive pattern matching on the source theory. This is the **canonical map** from the source theory to the target theory. In the case of monoids, this canonical map is the following `T_A` primitive recursive function; the keyword `Fixpoint` is for Coq something like `let rec` for ML and the primitive pattern matching begins with keyword `Cases` and ends with `end`:

   ```
   Fixpoint T_A[map:assoc_list;x:monoid] : A :=
     Cases x of
      (atom q)      => (assoc q map)
     | r1           =>  one_A
     | (rmult l r)  => (mult_A (T_A map l) (T_A map r))
    end.
   ```

   The variable `map` denotes an association list to interpret variables constructed with `atom` and `assoc` is the corresponding association function.

3. define in the meta-language (or implementation language) the inverse of `T_A` that we call `meta_A`. This can be performed by a simple inspection of the abstract syntax. For instance this progam, given the expression `(mult_A (f x) y)` in the target theory will be able to build the term `t≡ (rmult (atom 0) (atom 1))` and the list `l≡(0,(f x));(1,y)` such that `(T_A l t)` **reduces** (or **computes**) in an implicit proof step to `(mult_A (f x) y)`. We call this step the **metaification** as we use the meta-language to implement the reflection from the target theory into the source theory.

[3] or datatypes if you are an SML user

4. implement in the source theory the decision procedure that puts terms of the target theory in canonical form. So here we suppose that there exists a canonical form. For instance, the function `delete_neutral` below eliminates the extraoccurences of constructor `r1` in a term of type `monoid`, think that these functions are written nearly as they would be in ML as we use Coq as a programming language here:

```
Fixpoint del [x:monoid] : monoid :=
  [x:monoid]
     Cases x of
     | (rmult l r1)  => l
     | (rmult r1 l)  => l
     | t             => t
     end.
Fixpoint delete_neutral [x:monoid] : monoid :=
  Cases x of
  | (rmult l r1)  => (delete_neutral l)
  | (rmult r1 l)  => (delete_neutral l)
  | (rmult l r)   => (del (rmult (delete_neutral l) (delete_neutral r)))
  | t             => t
  end.
```

   This function performs in the source theory the same kind of action as tactic `Monoid_dec` defined in the previous section. We then build `R`, a **normalising function** in the same style as `delete_neutral`, which also performs association to the right.

5. check the correctness of the normalising function on the source theory. What we mean by checking the correctness of `R` is proving the following lemma where "`(l:assoc_list)(x:A)`" means informally "forall association list `l` and element x of `A`".

```
Lemma R_correctness : (l:assoc_list)(x:A) (T_A l (R x)) = (T_A l x).
```

   Hence we prove correctness of `R` with respect to the canonical map `T_A`.

6. define the reflection tactic, `Monoid_refl`, using the `meta_A` function and the `R_correctness` lemma. Rather than entering technical details concerning the Coq system, we describe informally how the tactic works:
   Suppose we want to decide `M = N` on `A`,the target theory

   - using the metaification function, we generate `m`, `n` and the association list `l` such that `(T_A l m) = M` and `(T_A l n)=N`.
   - then by conversion only, current goal is now `(T_A l m) = (T_A l n)`.
   - by application of the correctness lemma for the decision procedure `R` the goal becomes `(T_A l (R m)) = (T_A l (R n))`.
   - it is easy to translate this goal to: `(R m) = (R n)`.
   - by reduction we obtain an instance of the reflexivity of Leibniz equality as the new goal is `t = t` and we are done.

The more difficult step is to prove the `R_correctnes` lemma. To prove this lemma, we can use an LCF like tactic which is generally quite simple to write. By doing it we perform a "bootstrap" of the LCF like tactic when building the reflection tactic. For instance, we build an LCF-like tactic, called `Ring_dec`, which performs decision on the first order theory of abelian rings. Then, we build the reflection tactic `Ring_refl`, using the process of computational reflection described above. And

in the process of proving the correction of the corresponding normalisation function, we used 73 times the tatic `Ring_dec`. Most of the time, this was on quite complicated identities so that `Ring_dec` saved a lot of work. The bootstrap of LCF like tactics into reflected ones is not only a nice concept: it is really useful in practice.

Remarks

1. The building of a decision procedure is based on the following remark: most of the time, the process of putting an element of a first order theory in canonical form can be performed by primitive recursion on an inductive type that encode the first order theory. However, if primitive recursion is not enough, it is still possible to use a more general well founded induction in Coq; there are no limitations in this direction. Finally, putting a term in canonical form for a rewriting system is always performed by induction on the signature in some sense and we can use reflection to perform it.

2. In the case of monoids, you could take the type "`A -> monoid`" for constructor `atom`. However, most of the time, and particularly in the case of abelian rings, we need a complete ordering of atoms so that the use of integer is generally justified. For instance, decision on the first order theory of abelian rings amounts to sorting list and lists of lists so we need to compare the elements of these lists (which are atoms).

3. The method we describe here is rather general and the scheme of building a reflection tactic given above is not rigid. The main idea is that we reflect a theory in the object language into an inductive type in the object language. The metaification from the target theory to the source theory is performed in the meta-language but all of our approach is conservative. For example, here is work on progress to reflect propositions as types in the Coq system in order to check intuitionistic and classical tautologies; in this context, the canonical map from source theory to propositions as types looks like the following:

```
Fixpoint T_Prop[map:assoc_list;x:prop] : Prop :=
  Cases x of
   (atom q)   => (assoc q map)
  | p0        => False
  | p1        => True
  | (por l r) =>
     (T_Prop map l)\/(T_Prop map r)
  | (pand l r)=>
     (T_Prop map l)/\(T_Prop map r)
  | (pnot l)  =>
     ~(T_Prop map l)
  | (pxor l r)=>
        (T_Prop map l)/\ ~(T_Prop map r)
     \/ ~(T_Prop map l)/\ (T_Prop map r)
  | (pimp l r)=>
     (T_Prop map r) \/ ~(T_Prop map l)
  end.
```

and the corresponding correctness lemma of a decision function, say `R`, for tautologies is:

```
Lemma R_is_correct :
  (x:prop)(l:assoc_list)
    (T_Prop l x) <-> (T_Prop l (R x)).
```

We hope that this gives the taste of the wide range of application of computational reflection in Coq.

5 Total reflection

This section concerns an important issue arising from our work. The previous approach is very efficient as compared to LCF-approach[4] to build decision procedures , but is still too far from the performance of computer algebra systems or hand-coded decision procedures in a programming language. We now describe a means to fill most of the gap. We discuss performance in the next section. When applying a reflection tactic, most of the time is used in the normalisation process: if we keep the notations of the previous section, the long step is the computation of `(R n)=(R m)`, where we evaluate `(R n)` and `(R m)`. What we want to do here, is to use a very nice property of the Coq system: its implementation language is almost a sublanguage of the metalanguage. Coq is implemented in Objective Caml [15] a dialect of ML; and Coq has an extraction process [17] from the Coq object language to Objective Caml. So we can use this extraction process to translate `delete_neutral` and `R` from Coq to Objective Caml. This extraction process is automatic and its correctness is gauranteed. So, using the extraction process, we want to perform the computation of `(R m)` in the language Objective Caml rather than using the computation function of the Coq system. The evaluation of Objective Caml is 500 to 1000 times faster than the one of Coq for the following reasons. First Coq is not compiled but interpreted, second Coq's encoding of concrete types is more sophisticated than ML's because the notion of inductive type is more general than concrete types. Third Coq performs strong computation (reducing under abstractions) while Objective Caml uses call-by-value.

We call **total reflection** this process of combining reflection and extraction. Using total reflection means that the extraction process becomes critical to correctness of the system. Total reflection is not implemented yet but we can trace its efficiency as program extraction is implemented.

6 Application and performances

We have implemented a decision procedure on abelian rings using an LCF like approach and then using reflection. The algorithms under these tactics are different but we think that the following examples give the taste of the relative efficiency of these approaches. But this is in the context of a redution system where we keep proof objects. In this section, `A` endows a structure of abelian rings. Test are performed on a PC pentium pro 150.

- This is checked in 3s cpu using the LCF-like tactic.

```
Goal (n,m,p,q:A)[| n*(m*(p*q)) + m*p = p*m + q*(m*(n*p)) |].
Ring_dec.
Save Test4.
```

- the following decision is performed in 14s cpu.

```
Goal  (n,m,p,q:A)
  [|m*q*p + m*n*m*p + p*q*p + p*n*m*p + p*q = p*m*n*m + p*p*q + p*p*n*m + p*q*m + p*q |].
Ring_dec.
Save Test5.
```

- now 75s cpu.

```
Goal (n,m,p,q,r,s:A) [| (n + m + -p)*(q + (-r) + p) =(p + q + -r)*(m + (-p) + n) |].
Ring_dec.
Save Test9.
```

[4] in reduction systems

- The following example cannot be solved in the "fully expansive" or LCF approach. The decision procedure on rings defined by reflection performs the computation in 20s cpu on a pentium pro150 in constant space.

```
Goal (n,m,p,q,r,s:A)
[|(n + m + -p)*(q + (-r) + p)*(n + (-m) + -p)= ((-p) + n + -m)*(p + q + -r)*(m + (-p) + n) |].
```

- The following example is used to prove that the product of sums of eight squares is a sum of eight squares. To perform the computation corresponding to this goal, Coq needs 5 minutes in constant memory space.

```
Goal (p1,q1,r1,s1,t1,u1,v1,w1,p2,q2,r2,s2,t2,u2,v2,w2:A)
[| (p1*p1 + q1*q1 + r1*r1 + s1*s1 + t1*t1 + u1*u1 + v1*v1 + w1*w1)*
     (p2*p2 + q2*q2 + r2*r2 + s2*s2 + t2*t2 + u2*u2 + v2*v2 + w2*w2)
=  (p1*p2 + (-q1*q2) + (-r1*r2) + (-s1*s2) + (-t1*t2) + (-u1*u2) + (-v1*v2) + (-w1*w2))*
   (p1*p2 + (-q1*q2) + (-r1*r2) + (-s1*s2) + (-t1*t2) + (-u1*u2) + (-v1*v2) + (-w1*w2))
+  (p1*q2 + q1*p2 + r1*s2 + (-s1*r2) + t1*u2 + (-u1*t2) + (-v1*w2) + w1*v2)*
   (p1*q2 + q1*p2 + r1*s2 + (-s1*r2) + t1*u2 + (-u1*t2) + (-v1*w2) + w1*v2)
+  (p1*r2 + (-q1*s2) + r1*p2 + s1*q2 + t1*v2 + u1*w2 + (-v1*t2) + (-w1*u2))*
   (p1*r2 + (-q1*s2) + r1*p2 + s1*q2 + t1*v2 + u1*w2 + (-v1*t2) + (-w1*u2))
+  (p1*s2 + q1*r2 + (-r1*q2) + s1*p2 + t1*w2 + (-u1*v2) + v1*u2 + (-w1*t2))*
   (p1*s2 + q1*r2 + (-r1*q2) + s1*p2 + t1*w2 + (-u1*v2) + v1*u2 + (-w1*t2))
+  (p1*t2 + (-q1*u2) + (-r1*v2) + (-s1*w2) + t1*p2 + u1*q2 + v1*r2 + w1*s2)*
   (p1*t2 + (-q1*u2) + (-r1*v2) + (-s1*w2) + t1*p2 + u1*q2 + v1*r2 + w1*s2)
+  (p1*u2 + q1*t2 + (-r1*w2) + s1*v2 + (-t1*q2) + u1*p2 + (-v1*s2) + w1*r2)*
   (p1*u2 + q1*t2 + (-r1*w2) + s1*v2 + (-t1*q2) + u1*p2 + (-v1*s2) + w1*r2)
+  (p1*v2 + q1*w2 + r1*t2 + (-s1*u2) + (-t1*r2) + u1*s2 + v1*p2 + (-w1*q2))*
   (p1*v2 + q1*w2 + r1*t2 + (-s1*u2) + (-t1*r2) + u1*s2 + v1*p2 + (-w1*q2))
+  (p1*w2 + (-q1*v2) + r1*u2 + s1*t2 + (-t1*s2) + (-u1*r2) + v1*q2 + w1*p2)*
   (p1*w2 + (-q1*v2) + r1*u2 + s1*t2 + (-t1*s2) + (-u1*r2) + v1*q2 + w1*p2) |].
Ring_decision.
Save
```

The procedure is not yet optimized and for instance we use unary integers to encode the atoms (constructor `atom`). This sheds some light on the limitations of this approach as the same problem is solved in tenth of a second by the computer algebra system Maple on the same machine. However we should not forget that the algorithm used by Maple is certainly largely more efficient than the one we present here as we have the constraint that our algorithm has to be proven correct and complete so that we cannot make too many tricky transformations! Moreover the algorithm is written in a functional style. Indeed we can consider Maple will always be 5000 speeder than Coq to solve problems of this kind. But now, if we use total reflection using the extraction process, the computation is performed in less than 1 second cpu so that we are not so far from Maple performances on this particular example.

7 Related approaches

According to its author Richard Weyrauch, the FOL system [19] was the first theorem prover were reflection was considered as an essential tool for proving theorems. FOL is a proof checker for first order logic so that it was possible for the user to define signatures of first order theories and to give meaning to the constants of the signature by a mechanism of *semantic attachment*. It was possible for instance to define Peano arithmetic and to associate the constant zero to the machine integer 0 and the successor constant to the Lisp function (Lisp was the implementation language of FOL) which translates a machine integer to its successor. However, semantic attatchment allowed to interpret

closed terms only. There were also means of rewriting in a goal directed style like in the LCF system. And finally, an essential means of proving theorems was the use of reflection principles. The idea was that if w is a formula and D a logical derivation of w, then assuming it is possible to handle "w" and "D" in the object language, it is equivalent to prove Prf("w","D") where Prf is also a term of the object language that **computes** the process of checking that "D" is a derivation for "w". This was the great idea: *to change theorem proving in the theory into evaluation in the meta-theory* as Weyrauch says. To perform this program, a particular first order theory of the object language called META was explaining what it means to be a well formed formula of the object language, what it means to be a well formed term, what it means to be a derivation ... And then evaluation in the meta-theory was performed using semantic attachment to objects of META. An important point is that by the process of reflection, variables become constants and lemmas can be proved by computation on closed terms. This is also true in the more general context of reduction systems where computation works essentially on closed terms (you can compute `(mult 0 n)` in Coq even if n is "free", but not `(mult n 0)` as `mult` is defined by primitive recursion on its first argument)

So a lot of essential ideas were discoverd in the FOL experiments but the system itself was suffering a serions lack of **reliability**. The relation between the theory and the metatheory was postulated. The fact that given a problem in the theory, you can translate it into a computation in the meta-theory was not proved nor in the system itself neither anywhere else; the equivalence between D and Prf("w","D") was always an axiom. Moreover, the context of self reflection which arises when you reflect the theory META itself was called a potentially powerful tool according to Weyrauch but looks like a semantic hole for the system.

With respect to this first experiment of computational reflection in theorem provers, the Boyer-Moore approach is sounder. Roughly, rather than reflecting an object of the object language in the object language, they define their abstract syntax (through a function explaining what is a well formed term) in the object language and an evaluation function, EVAL, interpreting abstract syntax in the object language. And then they prove that a certain computation, say `f`, in the abstract syntax preserves meaning w.r.t `EVAL`. Then they can add safely the `f` function to the set of decision procedures of the system. They are able to do all of this because their implementation language and object language are roughly the same and flexible language: Lisp. Moreover they claim that their approach is as efficient as an hand coded extension of the system. However, the Boyer-Moore system is an automatic theorem-prover. This means that when you use this system, you just give a formula to the system and the system suceeds of fails in finding a proof. If it fails, you have to imagine the good succession of intermediate lemmas which will allow the system to prove your initial goal. Automatic provers emphasize on the problem of finding a proof and not on the problem of checking that a given proof is correct. In particular, it doesn't yield a proof object and it relies completely on their implementation of the prover. Furthermore, the operator Prf of the FOL context is now defined by a quotation mecanism and is not defined within the theory of the Boyer-Moore prover.

The work about reflection in Nuprl follow the ideas of Weyrauch but now in a sufficient powerful logical system so that the gap between theory and meta-theory is filled in the Nuprl system itself. In [14], Constable and Knoblock say "*The metatheories are tailored for a particular sort of meta-reasoning: representing enough of the proof theory of the previous language in the hierarchy so that proof tactics, functions that assist in proof development, are representable*". This program is possible in the Nuprl system because the formal logic under the system endows a hierarchy of universes U_i where i is an integer, s.t all the developments of U_j and U_j itself belong to U_{j+1} so that it is possible to perform some kind of boot-strap of the language of U_j in U_{j+1}, or partial bootstrap of U_j into itself, without any contradiction w.r.t Gödel's incompleteness theorem. We have not tried such an approach in the Coq system because of efficiency problems. The performances of the Nuprl approach are not discussed to our knowledge and we have no idea about the efficiency of computation in Nuprl. However, in the Coq system, as this is discussed in the next section, the computation of the system is between 500 and 1000 slower than its implementation language so that it seems hopeless to perform computations in a

boot-strapped version of the system and we think that the simplifications provided by the reflection would not compensate the inefficiency of the bootstrap.
Remark at the end that all these approaches look the same. FOL explained the advantage of using meta-reasoning but did not find a reliable way to perform this. In the Nuprl project, they investigate how to go from the theory to the meta-theory in the object language by some kind of boot-strap, and Boyer-Moore uses the same kind of approach. We propose to do this *metaification* in the meta-language preserving reliability and efficiency.
We inherit of Weyrauch approach the idea that reflection allows to change theorem proving into computation, but we do not perform the *metaification* in the object language as this is proposed in other systems, we perform it in the meta-language of the system. So that our approach seems less ambitious than the approach of FOL or Nuprl but we keep at the same time reliability and efficiency and furthermore complete reflection allows to considerably improve efficiency by only slightly weakening reliability (this can at least be used as an oracle which allows to delay computations at a moment where the user is not present). During the Bra-Aussois workshop, we saw that Barendregt and his co-workers were working on the same kind of project than ours, which they call the two-level approach. Concerning the topic of defining very efficient decision procedures in the context of theorem provers, Harrisson and Thery [9] propose to link HOL [7] and Maple to perform efficient computations but this also links the reliability of both systems! We believe that complete reflection is sounder than such union between a computer algebra system and a proof checker.

Conclusion

In this paper we have presented computational reflection and total reflection in reduction systems. We have experimented this methodology with the example of abelian rings in the Coq system. The idea is that we can solve, using definitional identity, problems posed on propositional identity by applying a meta theorem. At the same time, we translate explicit proof steps into implicit proof steps. In the example of abelian rings, these proof steps are always rewrite, but the methodology can be applied to many kinds of derivations. Simple reflection in Coq allows to define reasonably efficient certified decision procedures and this applies clearly to all reduction systems. Total reflection offers the possibility to gain another 500-1000 factor with a slight loss of reliability obtaining good results, even with respect to computer algebra systems: reliability costs less than an order of magnitude with respect to computer algebra systems for the examples we have tested. However total reflection is not yet implemented in the Coq proof development system and this is our main aim for future works. We are also interested in using the reflection mecanism to build efficient decision procedures for tautologies and even to implement decision by refutation for parts of first order logic following works of Hsiang [12]. For tautologies, an implementation of Stalmark's algorithm seems the best choice [10].

References

[1] http://pauillac.inria.fr/coq/coq-eng.html

[2] B. Barras et al. *The Coq Proof Assistant User's Guide, V6.1*, Inria technical report, to appear, 1997.

[3] R.S. Boyer and JS. Moore, Metafunctions: proving them correct and using them efficiently as new proof procedures. Dans "Then Correctness Problem in Computer Science", R.S Boyer et JS. Moore, pp 103-184, Academic Press, 1981.

[4] R.L Constable et al *Implementing Mathematics with the Nuprl Proof Development System* Prentice-hall 1986.

[5] T.Coquand, G.Huet *The Calculus of Constructions*, Information and Computation, 76, 1988.

[6] The Coq Proof Assistant Reference Manual, C. Cornes *et al*, Inria Research Report 177, 1995.

[7] M.J.C. Gordon and T.Melham, Introduction to HOL, Cambridge University Press, 1993.

[8] J. Harisson, *Metatheory and Reflection in Theorem Proving: A Survey and Critique.*

[9] J. Harisson, L. Théry *Extending the HOL theorem prover with a computer algebra system to reason about the reals.* Proceeding of the 1993 International Workshop on the HOL theorem proving system and its applications, LNCS 780, Springer-verlag, pp 174-184.

[10] J. Harisson, *Stalmarck's Algorithm as a HOL Derived Rule* TPHOL96, pp 221-234.

[11] D. Howe, *Computational Metatheory in Nuprl*, 9th International Conference on Automated Deduction, LNCS 310, Springer-Verlag, 1988, pp 238-257.

[12] J. Hsiang, *Topics in automated theorem proving and program generation*, Thesis, University of Illinois, 1982.

[13] J.M. Hullot, *A catalogue of canonical term rewriting systems* Technical Report CSL-112, April 1980.

[14] T.B Knoblock, R.L Constable *Formalized Metareasoning in Type Theory* Technical Report 86-742, Cornell University, 1986.

[15] X. Leroy, *The Objective Caml system*, Inria Technical Report, to appear 1997. see also http://pauillac.inria.fr/ocaml

[16] C. Paulin-Mohring *Inductive Definitions in the System Coq* TLCA 1993

[17] C. Paulin-Mohring, *Extraction de programmes dans le Calcul des Constructions*, Thesis, Paris 7, 1989.

[18] L. Paulson, *Interactive Theorem Proving with Cambridge LCF*, Technical Report 80, University of Cambridge, 1985.

[19] R. W. Weyhrauch, *FOL: A Proof Checker for First order Logic*, Stanford Artificial Intelligenc Laboratory Memo AIM-235.1, Stanford University.

[20] R. W. Weyhrauch, *Prolegomena to a theory of mechanized formal reasonning*, Artificial intelligence, vol 13 (1980), pp 133-170.