A paramodulation-based automatic tactic for the Matita proof assistant

Alberto Griggio <agriggio@cs.unibo.it>

1 Introduction

The HELM[1] (Hypertextual Electronic Library of Mathematics) project is a long-term effort, taken by prof. Andrea Asperti and his research group at the University of Bologna, with the aim of integrating the existing tools for formal reasoning to create, manage and exploit a large hypertextual library of formal mathematics.

One of the key components of HELM is its interactive theorem prover, called Matita, which allows the definition of new theorems and axioms that can then be added to the library. Such proof assistant is based on a quite expressive higher-order logical framework called CIC [14], an extension of the typed λ-calculus with a rich set of functional and inductive types.

In order to prove a theorem (a *goal*), a user applies to it one or more *tactics*, which replace it with some new subgoals, to which other tactics can be applied, and so on until there are no more open goals left. The role of the proof assistant is then that of ensuring the correctness of the various steps, but the whole procedure requires a continuous interaction with the user, who has to decide at each step which tactic to apply.

It seems clear that, on the one hand, having to follow the system step-by-step even to prove "trivial" subgoals can be quite boring and time-consuming, and on the other hand that there might be some goals for which the right strategy is not immediately clear, and so would require trying a lot of different sequences of tactics before finding the correct one.

To address this issues, Matita provides an `auto` tactic [16] that, given a goal, tries to find a proof for it *automatically*, by a repeated application of the hypotheses of the goal and the theorems and axioms in the HELM library, without the intervention of the user.

[1] `http://helm.cs.unibo.it`

However, `auto` has a problem: in the presence of equality predicates, it becomes rather inefficient. And, since equalities are so common and important, this is a serious limitation to its usefulness. This is due to the way equality is defined as a set of congruence axioms, whose applications generate owerwhelmingly many new subgoals.

The same issue has been faced and solved in the context of resolution-based automated theorem proving for first-order logic, [2], where the solution adopted was to treat equality not as an ordinary predicate, but rather as part of the language, with dedicated inference rules the most important of which is called *paramodulation* [10].

The aim of our work is then to investigate the possibility of applying the same technique in the context of HELM and its higher-order calculus CIC, to actually implement a new `auto` tactic which deals more efficiently with the equality predicate.

2 Preliminaries

This Section contains the basic notions about HELM, CIC and Matita that are needed to describe our work. It is not meant to be complete or even detailed, but only to give the minimum amount of information to understand the rest of the paper without prior knowledge of the topic. For a full treatment, we refer to [14].

2.1 The Matita proof assistant

With the term *proof-assistant* we generally indicate those programs that in some way "guide" a user through the demonstration of a theorem, checking and assuring the correctness of the single proof steps, and as a consequence that of the whole proof. However, this definition says nothing about how the theorems and their proofs are represented and/or manipulated, nor how the verification of correctness is performed, and in fact historically there have been many different approaches. That followed by Matita can be summarized in the following points:

- Both the theorems (goals) to be proved and their proofs are represented by *well typed CIC terms* (λ-*terms* from now on);
- The proofs are constructed applying to the goal a sequence of *tactics*, that intuitively stand for the various steps of the corresponding "traditional" demonstration of the theorem made with pencil and paper;

- The verification of correctness is based on the Curry-Howard isomorphism [4] (Chapter 3), according to which the theorem to prove is seen as a type of a CIC term, and demonstrating it amounts to construct a CIC term whose type is the theorem itself: the correctness is therefore stated by a *proof-checker* that is in fact a *type-checker* for CIC terms.

2.2 Metavariables

Traditionally, logic systems deal with *completely specified terms* and *completed proofs*. The inference rules of a logic system state what *is* derivable, and not what *might* be under some additionaly hyphotheses. Analogously, typing rules of a type system say what *is* well-typed and not what *might* be derivable with additional hyphotheses or instantiations. However, proof-assistants have the purpose of leading the user in the *construction* of a proof, and this obviously implies that they must be able to deal with terms and proofs that are *incomplete*. To this end, the logic system CIC has been extended with the addition of *metavariables*, which stand for conjectures not yet demonstrated. They are a particular kind of CIC term that have a type, but not yet a body. Completing a proof means to find a term of the right type for each metavariable present in the proof term itself.

2.3 Tactics

We have already said that a tactic is a procedure that represents a single proof step, that takes the current goal and replaces it with some new (and "simpler") *subgoals*, to which in turn other tactics can be applied, and so on until the demonstration is completed. More formally, a tactic implements a *backward reasoning*: if the current goal is the conclusion of a deduction rule, the application of a tactic to it replaces it with its premises (i.e. it generates a new subgoal for each premise), according to the following intuition: "if the goal G depends on (has as premises) $G'_1, \ldots, G'_n$, to get a proof for G we first need to have the proofs for $G'_1, \ldots, G'_n$".

Even more formally, a tactic can be defined in the following way, in agreement with [14]:

Definition 2.1 (Tactic) *A* tactic *is a function that, given a goal G (a local context and a type to inhabit) that satisfies a list of assumptions (hyphotheses) P, returns a pair (L, t) where $L = L_1, \ldots, L_n$ is a list (possibly empty) of subgoals and t is a function that, given a list $l = l_1, \ldots, l_n$ of terms such that each l_i inhabits L_i, returns a term $t(l)$ inhabiting G.*

The λ-term that, at the end of the proof, will inhabit the thesis, is built step-by-step by the tactics: this definition underlines the fact that a tactic, in addition to modify the current status of the proof, performs also the task of generating a "piece" of the final λ-term, i.e. the function t that transforms the λ-terms inhabiting the subgoals in the one inhabiting the goal to which the tactic has been applied.

3 Theoretical background

Here we give an overview of paramodulation and related techniques in the context of resolution theorem proving. The presentation is obviously incomplete and very informal. Also, we are assuming that the resolution method for first-order logic is known.

3.1 Basic paramodulation theory

The usual way of defining equality is by a set of axioms that can be summarized as follows:

$$\begin{array}{rcll} & \rightarrow & x = x & \text{(reflexivity)} \\ x = y & \rightarrow & y = x & \text{(simmerty)} \\ x = y \wedge y = z & \rightarrow & x = z & \text{(transitivity)} \\ x_1 = y_1 \wedge \ldots \wedge x_n = y_n & \rightarrow & f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) & \text{(monotonicity)} \end{array}$$

As mentioned in the Introduction, however, the use of these axioms in an automatic tactic leads to an explosion of subgoals generated, and so making the approach not feasible in practice.

Example 3.1 *Suppose we want to prove the goal*

$$(app\ (app\ (app\ S\ K)\ K)\ z) = z$$

with context:

A : SET
app : $A \rightarrow (A \rightarrow A)$
S : A
K : A
H : $\forall x, y, z : A.(app\ (app\ (app\ S\ x)\ y)\ z) = (app\ (app\ x\ z)\ (app\ y\ z))$
$H1$: $\forall x, y : A.(app\ (app\ K\ x)\ y) = x$
$H2$: $\forall x, y : A.(app\ (app\ K\ y)\ (app\ S\ x)) = x.$

We can obtain a proof by transitivity in the following way:

- *Apply transitivity to the goal, obtaining two new subgoals*

$$(app\ (app\ (app\ S\ K)\ K)\ z) = \ ?1 \quad and \quad ?1 = z;$$

- *Applying* H *to* K, K *and* z, *we can demonstrate the first subgoal, and so instantiate* ?1 *with*

$$(app\ (app\ K\ z)\ (app\ K\ z));$$

- *Applying the substitution* $\{?1 \mapsto (app\ (app\ K\ z)\ (app\ K\ z))\}$, *the second subgoal becomes*

$$(app\ (app\ K\ z)\ (app\ K\ z)) = z,$$

which can be demostrated with an application of $H1$ *to* z *and* $(app\ K\ z)$.

However, the search of the proof could have been quite different: for instance, we could have proved the second subgoal applying $H2$ *to* z *and a new metavariable* ?2, *instantiating* ?1 *with*

$$(app\ (app\ K\ ?2)\ (app\ S\ z)).$$

Now we could have proceeded trying to prove the first subgoal, which would have now become

$$(app\ (app\ (app\ S\ K)\ K)\ z) = (app\ (app\ K\ ?2)\ (app\ S\ z)),$$

only to discover later (in fact, maybe after having applied various symmetry, reflexivity, monotonicity or other transitivities) that it can not be demonstrated.

A paramodulation-based approach, instead, treats equality as a built-in construct of the language, thus avoiding the need of specifying it with the above axioms, which are replaced by some new inference rules[2], the main of which is the paramodulation rule:

$$\frac{C \vee s = t \quad D}{(C \vee D[t]_p)\sigma} \qquad \text{if } \sigma = mgu(s, D|_p),$$

where $mgu(a, b)$ is the function that computes the most general unifier of the terms a and b, $D|_p$ is the subterm of D at position p, and $D[t]_p$ is the result of the replacement in D of this subterm with t.

However, basic paramodulation is not enough to solve the original problem of efficiency, because its unrestricted application can lead to the generation of

[2]Beside the usual resolution and factoring rules of the resolution calculus, which is the setting we are considering now.

a large amount of new *clauses*. To overcome this difficulty, application of the paramodulation rule should be restricted in order to avoid redundant inferences. This can be done by imposing the constraint that paramodulation can be used only to replace *big* terms by *smaller* ones, with respect to a special ordering relation $\succ$ among terms, that satisfies certain properties, called *reduction ordering*. This restriction of the paramodulation rule is called *superposition*.

The use of superposition instead of paramodulation is quite important for efficiency because it reduces the search space of the solution by reducing the number of inferences that can be applied. But there is another technique that can be - and in fact is - applied to further reduce the number of clauses, and it is the elimination of redundancy. The criteria used for this purpose are essentially two: *subsumpion* and *demodulation* [19]. The former is used to discard clauses that are instances of more general ones, while the latter uses some equality clause $s = t$ to replace a complex clause containing s, $C[s\sigma]$, with a "simpler" clause $C[t\sigma]$ (where $\sigma = mgu(s, t)$), provided that $s\sigma$ is bigger than $t\sigma$ with respect to the reduction ordering.

In [10], it is shown that resolution combined with superposition and redundancy elimination is still refutation complete.

3.2 Inference system

Here are the inference and simplification rules used by our tactic. In the exposition, we will use the following notation: an equation $t_1 = t_2$ will be denoted with $t_1 = t_2 \rightarrow$ if it is a goal, and with $\rightarrow t_1 = t_2$ if it is a theorem or hyphothesis. Such notation derives from the more general one for clauses $\Gamma \rightarrow \Delta$, in which Γ is the set of premises and Δ that of conclusions: a theorem (or hyphothesis) is then a clause without premises, while a goal is a clause that leads to a contradiction. For simplification rules, we will use this syntax: $S \rightarrow S'$, where S is the current set of clauses, and S' is the result of the simplification.

Inference rules

superposition left

$$\frac{\rightarrow l = r \qquad t_1 = t_2 \rightarrow}{(t_1[r]_p = t_2 \rightarrow)\sigma}$$

if $\sigma = mgu(l, t_1|_p)$, $t_1|_p$ is not a variable, $l\sigma \not\preceq r\sigma$ and $t_1\sigma \not\preceq t_2\sigma$;

superposition right

$$\frac{\rightarrow l = r \qquad \rightarrow t_1 = t_2}{(\rightarrow t_1[r]_p = t_2)\sigma}$$

if $\sigma = mgu(l, t_1|_p)$, $t_1|_p$ is not a variable, $l\sigma \not\preceq r\sigma$ and $t_1\sigma \not\preceq t_2\sigma$;

equality resolution

$$\frac{t_1 = t_2 \rightarrow}{\Box}$$

if there exists $\sigma = mgu(t_1, t_2)$.

Simplification rules

tautology elimination

$$S \cup \{\rightarrow t = t\} \rightarrow S.$$

subsumption

$$S \cup \{C, D\} \rightarrow S \cup \{C\}$$

if C *subsumes* D, i.e. if there exists a substitution σ such that $D\sigma \equiv C$.

demodulation [19]

$$S \cup \{\rightarrow l = r, C\} \rightarrow S \cup \{\rightarrow l = r, C[r\sigma]_p\},$$

where:

(i) $l\sigma \equiv C|_p$;

(ii) $l\sigma \succ r\sigma$.

4 Implementation

Here we describe our implementation of the paramodulation technique described above. So far, we are restricting ourselves to a subset of CIC that is essentially a first-order calculus, so that we can use the algorithms used by the state-of-the-art first-order provers. The code is written in Objective Caml[3], as this is the language in which HELM is implemented.

[3] `http://caml.inria.fr`

4.1 Paramodulation in CIC

More formally, the set of terms accepted by our tactic (`auto paramodulation`) is given by the following definition:

Definition 4.1 (CIC terms accepted by `auto paramodulation`)

Theorms and hyphotheses:

1. *non dependent product, i.e.* $\Pi x : A.t$ *in which* A *has type* SET *and* t *does not depend on* x*. In this case, we will use the notation* $\rightarrow$ *instead of* Π*: for example,* $\Pi dummy : A.A$ *will be written as* $A \rightarrow A$*; or*
2. *application of the inductive type* ***cic:/Coq/Init/Logic/eq.ind****, of type* $\Pi A : \text{TYPE}.A \rightarrow A \rightarrow \text{PROP}$*, with arguments a term* A *of type* SET *and two terms of type* A *that do not contain abstractions: such terms correspond to ground equations. For example,* (`eq` $A\ t_1\ t_2$) *corresponds to a* $t_1 =_A t_2$*, in which* $=_A$ *indicates equality for terms of type* A*; or*
3. *term in the form*

$$\Pi x_1 : A.(\Pi x_2 : A.(\ldots(\Pi x_n : A.(\texttt{eq}\ A\ t_1\ t_2)))),$$

in which A *if of type* SET *and* t_1 *and* t_2 *depend on* $x_1, \ldots, x_n$ *but do not contain abstractions. Such terms correspond to equations* $t_1 =_A t_2$ *in which* t_1 *and* t_2 *contain* variables $x_1, \ldots, x_n$*.*

Goal:

application of ***cic:/Coq/Init/Logic/eq.ind*** *to a termA of type* SET *and two terms of type* A *that do not contain abstractions (i.e. as point 2 above).*

Example 4.1 *For example, some allowed terms are:*

1. A : SET
   $x : A$
   $n : nat$ *(inductive type* `cic:/Coq/Init/Datatypes/nat.ind`*)*
   $f_1 : A \rightarrow A \rightarrow A$ *(function with two arguments of type* A*)*
   $f_2 : nat \rightarrow nat \rightarrow nat$
   $g : nat \rightarrow nat$
2. $(eq\ nat\ (plus\ (S\ 0)\ n)\ (S\ n))$
3. $\lambda x_1 : nat.(\lambda x_2 : nat.(eq\ nat\ (f_2\ x_1\ x_2)\ (g\ n)))$.

Let us now suppose that we want to translate into CIC the problem of demonstrating the goal $f(a,b) = g(a)$, having the hyphothesis $f(x_1,x_2) = g(x_1)$, and in which $\mathcal{F} = \{f,g,a,b\}$ is the set of function and constant symbols and $\mathcal{X} = \{x_1,x_2\}$ is the set of variables. The first step is to assign a type to each object involved: therefore we define a term A of type SET*, that will be the type of constants and variables; then we define the types of the functions f and g according to their arities. In this way we get the following set of terms:*

$$\begin{array}{l} A : \textsc{Set} \\ a : A \\ b : A \\ f : A \to A \to A \\ g : A \to A \end{array}$$

Now, we have to translate the hyphothesis $f(x_1,x_2) = g(x_1)$. According to point 2 of Definition 4.1, we get

$$H : \Pi x_1 : A.(\Pi x_2 : A.(eq\ A\ (f\ x_1\ x_2)\ (g\ x_1))),$$

Finally, the goal has type

$$(eq\ A\ (f\ a\ b)\ (g\ a))$$

Variables and metavariables

In this translation from a first-order calculus to CIC, variables $\mathcal{X}$ are mapped into identifiers introduced by the λ operator. To get a ground instance of a term that contains variables, therefore, it is enough to apply the term to the proper arguments. For example, referring to Example 4.1, applying H to a and b, $(H\ a\ b)$, we get a term of type $(eq\ A\ (f\ a\ b)\ (g\ a))$, that is a proof for the goal.

However, using the inference rules of Section 3.2, the proof is obtained in a different way: first by an application of a superposition left to rewrite $f(a,b) = g(a)$ into $g(a) = g(a)$ using $f(x_1,x_2) \Rightarrow g(x_1)$, with *mgu* $\sigma = \{x_1 \mapsto a, x_2 \mapsto b\}$, and then applying equality resolution to the new goal. To do the same in CIC, we would need to unify the terms $\Pi x_1 : A.(\Pi x_2 : A.(f\ x_1\ x_2))$ and $(f\ a\ b)$ in order to be able to apply the superposition left rule. For this to be possible, however, we first have to *transform variables in metavariables*, in order to make the two terms unifiable:

$$\Pi x_1 : A.(\Pi x_2 : A.(f\ x_1\ x_2) \text{ becomes then } (f\ ?1\ ?2).$$

Now it is possible to unify the two terms, obtaining the substitution

$$\sigma = \{?1 \mapsto x_1 : A, ?2 \mapsto x_1 : A\}.$$

More precisely, every type 3 (with respect to Definition 4.1) equation is transformed into a type 2 one, in which identifiers $x_1, \ldots, x_n$ are replaced by metavariables (§2.2).

4.2 The main algorithm

The high level description of the procedure is the following: we have two sets of clauses, called *active* and *passive*. At the beginning, active is empty and passive contains all the initial clauses, i.e. the hypotheses and the negated goal. At each iteration of the loop that constitutes the heart of the algorithm, a clause (called *current*) is selected from passive and activated (i.e. added to active). After being activated, *current* is submitted to an inference function that builds the set (called *new*) of all the clauses that can be inferred from current and any other active clause. After this step, *new*, *active* and *passive* are subject to redundancy elimination (by the *simplify* function). This phase is made of two interleaving steps: forward simplification, when *active* and *passive* are used to demodulate and check for subsumption the clauses in *new*; and backward simplification, when conversely *new* is used to simplify *active* and *passive*: in this second case, however, if some active or passive clause is demodulated, it is removed from its set and added to *new*. Then phase one is performed again, and so on until the three sets don't change anymore. At this point *new* is examined to see if it contains the empty clause: if so, a refutation of the goal has been found and the procedure terminates. Otherwise, all the clauses in *new* are added to *passive* and the main loop starts again.

That just described is called the *given-clause algorithm* (figure 1), and it is the procedure used (with some variations) by all modern theorem provers.

4.3 Implementation details

The description just given is detailed enough to have an idea of how the system works and what the operations *select*, *infer* and *simplify* do, but it obviously says nothing about how they are performed efficiently. In fact, it turns out that at least three aspects are crucial for the efficiency of the algorithm.

Selection strategies

The first is the way of selecting *current* from *passive*. Our experiments have shown that this is the factor that affects most the performances: a poor selection policy can in fact cause a slowdown of the algorithm by a factor of ten or even

```
var new, passive, active: sets of clauses
var current: clause
active := ∅
passive := set of input clauses
while passive ≠ ∅ do
  current := select(passive)
  passive := passive \ {current}
  active := active ∪ {current}
  new := infer(current, active)
  new, active, passive := simplify(new, active, passive)
  if new contains empty clause
    then return "success"
  passive := passive ∪ new
end
return "failure"
```

Figure 1: Given-clause algorithm

more. Unfortunately however, there is no optimal strategy, but what might be quite effective for a particular goal can be very inefficient for another one. That's why we have implemented three different "basic" strategies, that can be combined in various ways according to some parameters settable by the user. Such basic strategies are:

age selection. It is the simplest way of selecting a clause: we keep the clauses in a queue, in which new clauses are added at the bottom of the queue and the selected one is always the top of the queue. The only interesting property of this strategy is *fairness*, which means that every non-redundant clause will eventually be selected;

weight selection. The idea here is that of associating to each clause a *weight*, that is a positive integer that in some way denotes the "complexity" of the clause, and is usually tightly related to the number of symbols in the clause. Clauses are then kept in a priority queue, and *select* always picks the clause with smallest weight.

goal similarity selection. The third basic strategy is to select the clause that is the "most similar" to the current goal. To do this, we define a similarity function that is related to the ratio of symbols that are in common between the clause and the goal and those that appear only in one of them.

The implemented strategy is a combination of the three basic ones just described: it is governed by two parameters (integers), k_{wa} and k_{sw}, both settable by the user, that indicate respectively the ratio between the number of clauses selected by weight and those selected by age and the ratio between similarity and weight selection.

Example 4.2 *For example, $k_{wa} = 4$ e $k_{sw} = 3$ mean that every five clauses, one is selected by its age and the other four with one of the two other strategies, and precisely three by similarity and one by weight.*

Simplification procedures

The description of the *simplify* procedure given above presents just one of the various redundancy elimination strategies possible. In fact, it is the so-called *full-reduction strategy*: its idea is to keep the search space as small as possible, by eliminating the redundancy in all the sets of clauses, i.e. *new*, *active* and *passive*. However, this operation is quite expensive from a computational point of view (in fact, it is the most expensive step of the algorithm): this is because the set of passive clauses grows quite quickly, and from a certain point on in the execution the time needed to simplify *passive* dominates that spent to infer new clauses and to simplify *active* and *new*.

This observation is at the basis of another simplification procedure, called *lazy-reduction strategy*, which differs from the previous one because the clauses in *passive* are neiher simplified nor used to simplify those in *new* and *active*: this allows to generate many more new clauses in the same amount of time with respect to the *full-reduction* strategy, however the price to pay is a potentially much higher degree of redundancy in the search space. As in the case of the selection strategy, again there is no best choice here: for some goals full-reduction might be better, whereas for others lazy-reduction could be more efficient. That's why `auto paramodulation` implements both, giving the user the choice to select which one to use.

Term indexing

The third aspect to consider is the constant growth of the sets *active* and *passive*. Both *infer* and *simplify* require in fact to search the two sets for clauses that satisfy certain conditions, as unifiability, subsumption or matching with a particular (or a set of) clause(s). If the searches are performed linearly, then, the constant growth of the two sets leads to a rapid and monotonic drop in performance of the system. Therefore, some form of indexing [15] of such sets had to be implemented: both

path indexing and discrimination trees were tried, and experiments conducted have shown that the latter are slightly better.

5 Construction of the proof terms

In the theorem provers for first-order logic based on resolution and paramodulation (like OTTER [8], SPASS [18] or VAMPIRE [13]), the proof of a goal is usually just a list of the generated clauses terminating with the empty clause. For each clause typically some information on how it was obtained is reported, for example if it was an initial clause or if it was generated by an inference rule, and in this case from which clauses, and so on.

In our case, however, this solution is not acceptable: as we said earlier (§2.1), Matita is based on the Curry-Howard isomorphism, which means that a proof for a goal must be a CIC term that inhabits the type that is the goal itself. This Section is therefore devoted to illustrate how such a term is constructed.

5.1 Demonstration of a single rewrite step

The proof of a goal is built incrementally, starting from the hyphotheses and theorems in the HELM library and proving each inference or semplification step. The "building blocks" with which the proofs are built are the proof terms for the single rewrite steps: the terms which prove, in other words, $T(a) = u$ starting from $T(b) = u$ and $a = b$, in which $T(a)$ indicates a generic term T which contains a subterm a.

They are built applying the theorem `cic:/Coq/Init/Logic/eq_ind.con`, of type

$$\forall A : \text{TYPE}.\forall x : A.\forall P : (A \to \text{PROP}).(P\ x) \to \forall y : A.x = y \to (P\ y)$$

that says exactly that we can obtain a proof that the predicate P holds for y from the proofs that P holds for x and that $x = y$.

5.2 Proof of a goal

When `auto paramodulation` is invoked, the demonstration of the goal is a metavariable, which actually represents the proof not yet completed (§2.2). The proof term is built combining the proofs of the single rewrite steps in the following way: let us suppose that we have an equation $a = c$ (of type A), whose proof is P_1, and a goal $T_1(a) = T_1(c)$, whose proof is a metavariable $?n$. If we apply a superposition left or

a demodulation, we can obtain $T_1(c) = T_1(c)$, which is a tautology whose proof can be obtained by reflexivity (i.e. applying `cic:/Coq/Init/Logic/eq.ind#xpointer(1/1/1)`, that is `refl_equal`):

$$(\texttt{refl\_equal}\ T_1(c)).$$

Observing that the tautology $T_1(c) = T_1(c)$ is equivalent to $((\lambda x : A.T_1(x) = T_1(c))\ \ c)$, which therefore has the same proof, we can demonstrate the initial equation $T_1(a) = T_1(c)$ instantiating the metavariable $?n$ with an application of `eq_ind` in the following way:

$$?n \mapsto (\texttt{eq\_ind}\ A\ c\ \lambda x : A.T_1(x) = T_1(c)\ (\texttt{refl\_equal}\ T_1(c))\ a\ P_1),$$

Generalizing this procedure, every time that we rewrite a goal $G_1(l)$ with proof $?n_1$ using an equation $l = r$ with proof P, we create a new metavariable $?n_2$ which represents the proof of the new goal $G_1(r)$, and instantiate $?n_1$ with

$$(\texttt{eq\_ind}\ A\ r\ \lambda x : A.G_1(x)\ ?n_2\ l\ P).$$

When we rewrite the current goal into a tautology $t = t$, the last metavariable created, $?n_m$, will be instantiated with $(\texttt{refl\_equal}\ t)$.

5.3 Proofs of non-goal clauses

The last point to consider for the construction of the proof is how to get the demonstration of a clause $\rightarrow T$ (generated by superposition right or demodulation). The procedure is similar to that for goals: proofs are compositions of single rewrite steps and of applications of theorems and hyphotheses.

The base case is the direct application of a theorem or hyphothesis: the proof term in this case is just the application of the theorem/hyphothesis to the appropriate arguments. A simple example explains what just stated:

Example 5.1 *If we have a hyphothesis $H : \Pi x : A.\Pi y : A.(f\ x\ y)$, the CIC term that demonstrates $(f\ a\ b)$ (where $f : A \rightarrow A \rightarrow \textsc{Prop}$, $a : A$ and $b : A$), is simply $(H\ a\ b\ )$.*

Having the proof terms for the theorems and hyphotheses, we can build the proofs for the generated equations inductively, applying directly `eq_ind`: if $l = r$ is the equation used to rewrite $T(l)$ into $T(r)$, the CIC term that demonstrates $T(r)$ is

$$(\texttt{eq\_ind}\ A\ l\ \lambda x : A.T(x)\ P_{T(l)}\ r\ P_{l=r}),$$

where A is the type of equality, and $P_{T(l)}$ and $P_{l=r}$ are the proof terms for $T(l)$ and $l = r$ respectively.

6 Using the Matita library

One of the aims of Matita (and HELM more in general) is to reuse as much as possible all the already formalized knowledge present in its extensive library of theorems (and axioms, and definitions); moreover, among the very reasons for developing an automatic tactic there is that of preventing the user from having to search the library for theorems useful to prove a given goal. For these reasons, `auto paramodulation` needs to have access to the Matita library, and the access method has to take into account two different (and contrasting) requirements:

1. first, if a goal g can be proved by `auto paramodulation` without using the library, but having the hyphotheses $H_1, \ldots, H_n$, g should be equally provable with the use of the library, when $H_1, \ldots, H_n$ are in it and with no additional hyphotheses;

2. moreover, if in the proof of a goal g the theorem T present in the library is not used, for efficiency concerns it should never be considered by `auto paramodulation` when looking for a proof of g.

These are obviously *ideal* requirements: satisfying both of them would mean in fact knowing in advance *all and only* the theorems necessary to prove the goal, which is equivalent to know a proof for it. It is also obvious that, taken in isolation, these two requirements are trivially satisfiable: for the first it would be sufficient to use the entire library - that contains more than 30.000 elements! - for all the proofs, while for the second we could simply ignore the library completely. But clearly these two solutions are not very interesting: what we were looking for was a compromise between completeness (point 1) and efficiency (point 2).

The solution we have adopted is similar to that used by the original `auto` tactic [16]. It uses the metadata associated to every item in the library to identify those that are "compatible" with a certain goal. More precisely, the only items that are considered by `auto paramodulation` when constructing a proof for a goal g are those which satisfy all the following points:

- are equations, that is they have `cic:/Coq/Init/Logic/eq.ind#xpointer(1/1)` as constant in position MainConclusion;

- satisfy the *exactly constraint* [16] for the set of constants in the goal, in the local context (i.e. the set of hyphotheses) and in the types and constructors of inductive types of the goal and the objects in the context. If we call S such set of constants, an item t in the library satisfies an *exactly constraint* on S if

$$\exists S' \in \wp(S) \text{ t.c. } \mathrm{constants\_of}(t) = S',$$

where $\wp(S)$ is the powerset of S, and constants_of(t) is the set of constants appearing in t.

7 Results obtained

So far, we have only considered a subset of CIC that corresponds to a first-order logic, and we have limited our scope only to equational theories, i.e. we considered only problems in which both the goal and all the hypotheses contain only one literal which is an equality. This might seem quite limited, but this is the heart of the paramodulation technique, and we believe that extensions to this basic technique will require much less effort than that made so far. Moreover, the performance gain w.r.t. `auto` is impressive (two orders of magnitude and more), and this alone should be enough to consider the approach successful, even in the case that the extensions of it should reveal harder to implement.

As regards the performance w.r.t the state-of-the-art provers, we mainly compared with Spass: the results vary from problem to problem, but on average we are about an order of magnitude slower (the results heavily depend on the clause-selection strategy, as already discussed at the end of the previous Section), which is not bad at all if we consider that:

- we are reusing as much of the existing HELM code as possible, and this code wasn't designed with theorem-proving in mind, so many of its data structures and functions are not optimized for efficiency but rather for easy manipulation, maintainability, extensibility, modularity;

- OCaml, although quite fast when compiled to native code, is still slower than C;

- we are comparing with a state-of-the-art implementation, with many years of development behind it, while our code is less than four months old;

- although we are currently using only a subset of it, CIC is still much more complex than an untyped first-order calculus, and so the algorithms that operate on terms are intrinsically more complex in our case.

8 Conclusions

The motivation of this work was the need to improve the efficiency of the `auto` tactic of the Matita proof assistant. The first thing we did to achieve this goal was

to experiment with some optimizations to the code of `auto`, basically to reduce the size of the search space on which the tactic operates. These optimizations were useful, but they were not effective enough: they caused an improvement of the execution time only of a constant factor (in some cases even 3 or 4), but they did not change its order of magnitude. In particular, compared to the modern theorem provers for first-order logic (like OTTER [8], SPASS [18], VAMPIRE [13], WALDMEISTER [6]), `auto` was still at least 1.000 times slower, and so really not usable for non trivial problems.

The main cause of this huge difference of performance is the treatment of the equality predicate, that in the aforementioned tools is handled in a special way, with dedicated inference rules, the main of which is called *paramodulation* [10].

This observation made us decide to implement a new automatic tactic from scratch, following the architecture of a modern theorem prover. The result of this work is the `auto paramodulation` tactic just described. Despite its limited applicability (so far, only to pure equational problems), it turned out to be quite effective, reducing the execution times at least of two orders of magnitude.

8.1 Related work

Despite its limitations, we can say that `auto paramodulation` is an effort to integrate automatic and interactive theorem proving. A possible scenario for its use is, in fact, the solution of complex problems, the search for the proof of which is guided by the user, but in which some equational subgoals can be solved automatically by `auto paramodulation`.

From this point of view, our work has many analogies with other efforts to combine the interactive and automatic approach to theorem proving, as for example [7], [9] and [1]. The difference between such works and `auto paramodulation` is that in the former the integration is obtained with the development of an intermediate layer of communication between a pre-existing and substantially independent proof assistant and theorem prover (HOL [5] and GANDALF [17] in the first case, Isabelle [11] and VAMPIRE in the second, and KIV [12] and ${}_3TAP$ [3] in the third): in these approaches, the problem is translated from the language (usually a higher order one) of the proof assistant to that (first order) of the theorem prover, which is then invoked on this translation of the problem; finally, in case of success, the inverse translation has to be performed, in order to get a valid proof for the proof assistant. On the contrary, `auto paramodulation` has an architecture which is similar to that of a theorem prover, but is in fact a part of Matita, sharing with it data structures and algorithms, and operating directly on CIC terms (although at present it accepts only a subset of them).

8.2 Future developments

We have already stressed the fact that `auto paramodulation` at present can be used only for purely equational problems. A natural direction of development, therefore, is its extension to the whole CIC, or at least to the whole first order logic, with the standard connectives $\neg$, $\wedge$ and $\vee$ (whose definitions in HELM are respectively `cic:/Coq/Init/Logic/not.con`, `cic:/Coq/Init/Logic/and.ind` and `cic:/Coq/Init/Logic/or.ind`).

In particular, an extension to the full first order logic could be realised putting the goal and the theorems/hypotheses in conjunctive normal form, and then proceeding by resolution with paramodulation like the state-of-the-art theorem provers.

An extension to the whole CIC, instead, would require a different approach. A possible solution would be the following: use a hybrid between the `auto` and `auto paramodulation` tactics, in which rewrite and simplification steps, using paramodulation and demodulation, are alternated to theorem application steps, in the manner of `auto`.

In a different direction, another possible improvement could be the use of the paramodulation technique to generate automatically al the "interesting" consequences of an initial set of theorems and axioms, for some definition of "interest". For example, we could consider as interesting all the theorems that are used very frequently - as rewrite rules - in the demonstration of a certain goal, and then suggest the user to add them to the library for a future use.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integration of automated and interactive theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume II, chapter 4, pages 97–116. Kluwer, 1998.

[2] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*, volume I. Elsevier Science, 2001.

[3] Bernhard Beckert, Reiner Hähnle, Peter Oel, and Martin Sulzmann. The tableau-based theorem prover $_3$*tap*, version 4.0. In M. A. McRobbie and J. K.

Slanley, editors, *Proc 13th CADE*, volume 1104 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[4] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[5] M. J. C. Gordon and T. F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.

[6] Thomas Hillenbrand and Bernd Löchner. A phytography of WALDMEISTER. *AI Communications*, 15(2-3):127–133, 2002.

[7] Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321. Springer, September 1999.

[8] William McCune. OTTER 3.3 reference manual. Technical Report ANL/MCS-TM-263, Argonne National Laboratory, August 2003.

[9] Jia Meng. Integration of interactive and automatic provers. In Manuel Carro and Jesus Correas, editors, *Second CologNet Workshop on Implementation Technology for Computational Logic Systems*, September 2003.

[10] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*, volume I. Elsevier Science, 2001.

[11] Lawrence C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994.

[12] W. Reif, G. Schellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In *14th International Conference on Automated Deduction. Proceedings*. Springer-Verlag, 1997.

[13] Alexandre Riazanov. *Implementing an efficient theorem prover*. PhD thesis, University of Manchester, 2003.

[14] Claudio Sacerdoti Coen. *Kowledge management of formal mathematics and interactive theorem proving*. PhD thesis, University of Bologna, 2004.

[15] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*, volume II. Elsevier Science, 2001.

[16] Matteo Selmi. Tattiche di dimostrazione automatica di teoremi su larghe basi di conoscenza. Master's thesis, University of Bologna, 2004.

[17] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[18] Cristoph Weidenbach. *The theory of* SPASS *version 2.0*. MPI für Informatik, Saarbrücken, 2001.

[19] Lawrence Wos, George A. Robinson, Daniel F. Carson, and Leon Shalla. The concept of demodulation in theorem proving. *Journal of the ACM*, 14(4), 1967.