Efficient Retrieval of Mathematical Statements*

Andrea Asperti, Matteo Selmi

Department of Computer Science
Via Mura Anteo Zamboni 7
40127 Bologna, Italy

Abstract. The paper describes an innovative technique for efficient retrieval of mathematical statements from large repositories, developing and substantially improving the metadata-based approach introduced in [13].

1 Introduction

The increasing amount of mathematical literature published on the web in structured, semantically rich XML-formats [3, 1, 2] and the recent combination of these technologies with tools for automated deduction and mechanization of formal reasoning, independently pursued by several European Projects like OpenMath, Calculemus, MKM-NET, MoWGLI and Monet has recently renewed the interest for efficient indexing and retrieving techniques of *single mathematical items* like theorems, definitions, examples, and so on (see e.g. [14, 5, 9, 12, 13]).

Typical techniques to query libraries of structured mathematics are based on some kind of (first order) matching, spanning from simple pattern-matching techniques to unification to more sophisticated queries comprising e.g. type isomorphisms [8, 7]. However, as already pointed out in [13], the straightforward iteration of the matching operation over the whole repository of mathematical knowledge has serious scalability problems with respect to the dimension of the library. For instance, the MBase system [15], thanks to a clever implementation of unification, is able to iterate matching on a data base of 6000 theorems (that is, a pretty small library) in about 1 second: if this performance is possibly acceptable for humans, is definitely too slow for the purposes of automatic deduction, where we need to repeatedly look for applicable theorems exploring huge trees of different possibilities.

The problem of speeding up matching and unification has already been extensively studied in the theorem proving community, but in a sensibly different context: in particular, the knowledge base for retrieval is in that case a local context, subject to frequent insertions and deletions, but stably loaded in RAM. As a consequence, the typical solution is to store the data base of mathematical items in ad-hoc data-structures, particularly optimized and integrated with the matching algorithm, such as discrimination trees [6, 16], substitution trees [11] or context trees [9]. As far as we know, the feasibility of scaling-up these approaches

* Partially supported by the European Project IST-2001-33562 MoWGLI

to huge, stable repositories of available mathematical knowledge (the real context of Mathematical Knowledge Management) has never been addressed.

A natural way for indexing huge libraries of mathematical statements, independently introduced in HELM [13] and Mizar [5], is that to automatically extract a small set of metadata from each mathematical item, providing an *approximation* of its content, and devised to capture some simple invariants of first order matching.

In this paper, we propose a significant improvement of the technique described in [13], still relying on essentially the same Metadata Set (the HELM metadata set, originally conceived by the first author, and first published in [17]).

2 Metadata as Approximations

For efficiency reasons, the metadata we are interested in are supposed to be expressible in a simple relational model. This excludes, for instance, the possibility of indexing an object with its type or with a (complex) subterm occurring in it, since these are not flat data, and we would be back to the problem of efficiently indexing and matching this information. The only flat data we may dispose of are essentially occurrences of constants. The other kind of information we may attach to these occurrences is their actual position inside the term. So, the main relation of our Metadata set is a simple ternary realtion of the kind

$$Ref(source, target, position)$$

stating that *source* contains a reference to *target* at the specified *position*. By suitably describing such a position we may obviously reach the same degree of granularity of a substitution tree (where each variable marks indeed a different position), thus providing a complete description of the term. However, for limiting memory occupation, we opted for a minimal set of "interesting" positions, that, essentially just discriminates the root of the conclusion (we call this position MainConclusion) of the statement from all other occurrences.

Example 1. Suppose we have a statement asserting that for all x

$$x * 0 = 0$$

the metadata are described by the following table:

constant	position
=	MainConclusion
*	InConclusion
0	InConclusion

Note that variables are not indexed (it would be very difficult to take care of alpha conversion).

Example 2. Suppose to have a statement asserting that for all x, y, z

$$x * (y + z) = x * y + x * z$$

Its metadata are described by the following table:

constant	position
$=$	MainConclusion
$*$	InConclusion
$+$	InConclusion

Question: in a mathematical library, how many theorems would you expect to find stating an equality between two different terms containing only $*$ and $+$?

As far as we are concerned with matching in backward reasoning, i.e. with the problem of finding generalizations of a given goal, indexing the conclusion of statements is enough. For this reason, in the rest of the paper we shall essentially work with the few metadata introduced above.

However, for other operations, like e.g. interactive support to forward reasoning, the HELM metadata set also supports indexing of hypotheses. These additional metadata are also useful in all those cases where the conclusion of the statement does not hold enough information to allow a sensible indexing. A typical, pathological case is e.g. the induction principle for natural numbers, or, more generally, the elimination scheme for an inductive type: the conclusion is a generic universal assertion containing no constant at all. On the other side, the hypothesis, listing both the type of the inductive predicate and of the eliminated data, usually provide enough information for an efficient retrieval.

The interested reader is referred to [17, 13] for an exhaustive description of the HELM Metadata Set.

3 Matching conclusions

The problem of matching (for backward reasoning) consists of finding generalizations of a given goal G, i.e. of finding all those theorems in the repository which can be profitably applied to the given goal. If the theorem T has a shape of the kind $\forall x_1, \ldots, x_n.G'$, the problem consists to look for a substitution σ such that $G'\sigma$ can be unified with G (typically, via first order unification). Since substitution can only instantiate terms (and cannot erase any subterm), every constant occurring in G' must already occur in G: we call these constraints *atmost constraints*[1].

Definition 1. *An* atmost constraint *is a set of constants atmost*$\{c_1, \ldots, c_n\}$. *A formula c satisfies the constraint if all constants occurring in the conclusion of c are in atmost*$\{c_1, \ldots, c_n\}$, *that is if*

$$\{x | Ref(c, x, InConclusion) \vee Ref(c, x, MainConclusion)\} \subseteq \{c_1, \ldots, c_n\}$$

[1] We adopted the terminology of Mizar. In particular, the *only constraints* and *must constraints* of [17, 13] are respectively renamed into *atmost constraints* and *atleast constraints*

In this case we shall write $c \trianglelefteq \{c_1, \ldots, c_n\}$.

Checking that a theorem t meets an *atmost constraint* with our metadata is not a very simple operation: it requires a query to the data base and then an explicit comparison of two sets (for each thteorem to test). According to our measures, the cost of checking an *atmost constraint* is in fact comparable to the cost of a direct unification check *provided the theorem t is already in memory.* In fact, the real point of *atmost constraints* (not so well explained in [13]) is not to speed up filtering, but to avoid loading useless objects in the central memory. The latter can be a *very* expensive operation, especially if it require parsing, type-checking or, even worse, downloading via web).

In any case, if we had to iterate the application of an *atmost constraint* to the whole library we would be back to the same scalability problem we started with.

Things are a bit better if we assume to work in a "first order setting", that is quite typical of automatic tactics even in higher order systems. In this case, we may assume that the head operator of the current goal G is already instantiated, and thus it must appear (again in head position) in any generalization G'. So, with no loss of generality, we may restrict our search to the set of theorems sharing the same predicate of the goal (in position MainConclusion).

Example 3. Suppose we are interested in looking for theorems matching the goal $0 \leq S(0)$. We search for theorems having $\leq$ in MainConclusion and with 0 and S as *atmost constraint*. In the Coq library, composed of about 40000 mathematical items (theorems, definitions, inductive types, ...), we have 263 theorems with $\leq$ in MainConclusion. Although this provides a significant reduction of the search space, iterating the evaluation of the *atmost constraint* for 263 times is a really expensive operation (especially in tactics for automatic theorem proving).

The table in Figure 1 gives the ordered list of the (top 20) constants appearing in the Coq library in position MainConclusion, together with their frequency.
The diagram in Figure 2 associates to each theorem of the library (x-line) the number of other theorems sharing with the former the same constant in MainConclusion (y-line); the big pick on the left is of course due to Leibniz-equality. The average value is about 596 that also provides a reasonable estimation of the average number of times we may expect to iterate the computation of *atmost constraint* in a generic query!

4 Atleast Constraints

A way for improving the initial filtering (preliminary to the application of an *atmost constraint*) is by requiring that more symbols of the current goal G occur in the conclusion of the theorem we are looking for; we call these constraint *atleast constraints*.

n	frequency	object
1	3978	Coq/Init/Logic/eq.ind#xpointer(1/1)
2	920	Coq/Init/Logic/not.con
3	753	CoRN/algebra/CSetoids/cs_eq.con
4	557	CoRN/algebra/CSetoids/cs_crr.con
5	438	Coq/Init/Logic/ex.ind#xpointer(1/1)
6	435	Sophia-Antipolis/Functions_in_ZFC/Functions_in_ZFC/Ens.con
7	421	Sophia-Antipolis/Functions_in_ZFC/Functions_in_ZFC/IN.con
8	413	CoRN/algebra/COrdFields/leEq.con
9	366	Nijmegen/Coqoban/Coqoban_engine/Board.ind#xpointer(1/1)
10	354	CoRN/algebra/CSetoids/Ccsr_rel.con
11	353	Coq/Init/Datatypes/nat.ind#xpointer(1/1)
12	350	Coq/Init/Logic/or.ind#xpointer(1/1)
13	333	Coq/Init/Specif/sumbool.ind#xpointer(1/1)
14	328	Sophia-Antipolis/Algebra/Sets/Equal.con
15	303	Coq/Init/Logic/and.ind#xpointer(1/1)
16	280	CoRN/algebra/CSetoidFun/PartFunct.ind#xpointer(1/1)
17	263	Coq/Init/Peano/le.ind#xpointer(1/1)
18	247	Coq/ZArith/zarith_aux/Zle.con
19	245	CoRN/algebra/CLogic/CProp.con
20	245	Coq/Reals/Rdefinitions/Rle.con

Fig. 1. Top 20 operators appearing in position MainConclusion

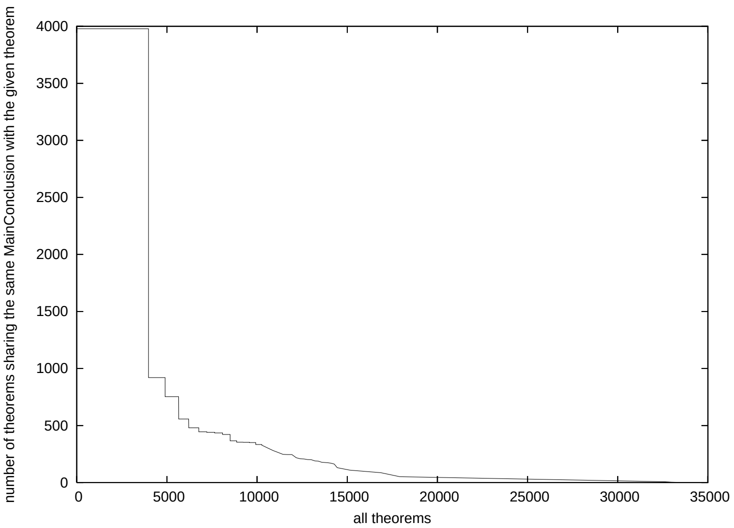

Fig. 2. Diagram associating to each theorem of the COQ-library (x-line) the number of other theorems sharing with the former the same constant in MainConclusion (y-line)

Definition 2. *An* atleast constraint *is a set of constants atleast*$\{c_1, \ldots, c_n\}$*. A formula c satisfies the constraint if all the constants* $\{c_1, \ldots, c_n\}$ *are in the conclusion of c, that is if*

$$\{c_1, \ldots, c_n\} \subseteq \{x | Ref(c, x, InConclusion)\}$$

In this case we shall write $\{c_1, \ldots, c_n\} \trianglelefteq c$ *(and* $\{c_1, \ldots, c_n\} \trianglelefteq_{cmain} c$ *if* c_{main} *is the constant in position MainConclusion of c.*

The interest of *atleast constraints* is that in order to find all theorems satisfying a given constraint $\{c_1, \ldots, c_n\}$ it is enough to compute

$$\bigcap_{i=1\ldots n} \{x | Ref(x, c_i, InConclusion)\}$$

that is a simple n-ary join operation in a relational DB.

On the other side, the big problem of *atleast constraint* is that they can potentially cut-off usefull theorems. Consider for instance, the goal is $0 \leq S(0)$. We could probably expect to find in the library, and to apply, a theorem stating that $\forall x.0 \leq x$ (provided we are talking about natural numbers). So, we may restrict ourselves to all those theorems having $\leq$ in MainConclusion and 0 InConclusion, that is imposing 0 as *atleast constraint* (there are just 36 of them in the whole library). However, the same goal could also have been proved by a direct application of a theorem stating that $\forall x.x \leq S(x)$ (such a theorem can be actually found in the Coq-library: `cic:/Coq/Arith/Le/le_n_Sn.con`); obviously, $\{0\} \not\trianglelefteq$`le_n_Sn.con` an the latter theorem would not be found by the above query.

The theorems containing $\leq$ in MainConclusion and satisfying $\{S\}$ as *atleast constraint* are 98. The union of this set with the previous 36 theorems give a set of 99 theorems (35 theorems contain both 0 and S). This is still a sensible improvement w.r.t the initial 263 candidates, but, alas, we are still neglecting all those theorems just containing $\leq$ in MainConclusion, and *nothing more*, such as, for instance, the transitivity of $\leq$.

5 Exactly constraints

The considerations at the end of the previous section suggests a different approach to the computation of *atmost constraints*. First of all, we need the notion of *exactly constraint*.

Definition 3. *An* exactly constraint *is a set of constants exactly*$\{c_1, \ldots, c_n\}$*. A formula c satisfies the constraint if the constants* $\{c_1, \ldots, c_n\}$ *are exactly the constants appearing in the conclusion of c, that is if*

$$\{c_1, \ldots, c_n\} = \{x | Ref(c, x, InConclusion)\}$$

In this case we shall write $\{c_1, \ldots, c_n\} \bowtie c$ *(and* $\{c_1, \ldots, c_n\} \bowtie_{c_{main}} c$ *if* c_{main} *occurs with position MainCocnlusion in c.*

The first iportant fact is that

$$t \trianglelefteq_c C \Leftrightarrow t \in \bigcup_{P \subseteq C} \{x | x \bowtie_c P\}$$

that is t contains at most a set C of constant if and only if for some subset P of C, t contains exactly the constants in P (in the next section we shall see that not every subset of C has to be taken into account). The second important consideration is that *exactly constraints* may be efficiently conputed by a simple extension to the metadata set. The new relation

$$no_concl(source, no)$$

associates to each statement *source* the number *no* of different symbols in its conclusions (the new table can be computed in few seconds from the already available metadata). Then we have

$$t \bowtie_c \{c_1, \ldots, c_n\} \Leftrightarrow \{c_1, \ldots, c_n\} unlhd_c t \wedge no_concl(t, n)$$

, that is t contains exactly $\{c_1, \ldots, c_n\}$ if and only if it contains at least $\{c_1, \ldots, c_n\}$, and it just contains n constants.

Example 4. Retrieving candidates for matching the theorem $0 \leq S(0)$ amounts to the execution of four SQL queries, corresponding to the four *exactly constraints* specified by the sets $\emptyset$, $\{0\}$, $\{S\}$ and $\{0, S\}$. The cardinality of the respective result set (for the COQ-library) is given in the following table:

exactly constraint	#result
$\emptyset$	60
$\{0\}$	1
$\{S\}$	15
$\{0, S\}$	5
total	81

Example 5. Suppose we have the goal $x + 0 =_{nat} x$. The equality we are dealing with is Leibniz equality, and *nat* is an explicit constant appearing in the goal. So, in this case, matching the goal, amounts to solve the *exactly constraint* $\emptyset$, *nat*, $\{0\}$, $\{+\}$, $\{nat, 0\}$ $\{nat, S\}$, $\{0, S\}$ and $\{nat, 0, S\}$ with results sets as described in the following table:

exactly constraint	#result
$\emptyset$	195
$\{nat\}$	81
$\{0\}$	0
$\{+\}$	0
$\{nat, 0\}$	46
$\{nat, +\}$	9
$\{0, +\}$	0
$\{nat, 0, +\}$	9
total	340

In this case, the total number of 340 should be compared against the number of theorems satisfying an empty *atleast constraint* (with Leibniz equality in MainConclusion), that is 3978.

Remark 1. It never happens, in practice, that constants occur inside the conclusion of an equality statement if the type of the equality is not instantiated as well. This simple strategy allows to spare the three queries with an empty result set in the example above.

For the interested people, the last SQL query it trivially generated from the given constraints, and has the following shape:

```
select refObj.source
  from refObj,refObj as refObj1,
  refObj as refObj2,refObj as refObj3, no_concl
where
  refObj.occurrence
    ="cic:/Coq/Init/Logic/eq.ind#xpointer(1/1)" and
  refObj.position="MainConclusion" and
  refObj1.occurrence
    ="cic:/Coq/Init/Datatypes/nat.ind#xpointer(1/1)" and
  refObj1.position="InConclusion" and
  refObj2.occurrence
    ="cic:/Coq/Init/Datatypes/nat.ind#xpointer(1/1/1)"
  refObj2.position="InConclusion" and
  refObj3.occurrence
    ="cic:/Coq/Init/Peano/plus.con"
  refObj3.position="InConclusion" and
  refObj.source = refObj1.source and
  refObj1.source = refObj2.source and
  refObj2.source = refObj3.source and
  refObj3.source = no_concl.source and
  no_concl.no = 3;
```

The big question is of course how many different sets of *exactly constraints* we may be forced to generate for matching a given goal. If we have n constants in the goal, this is obviously bounded by 2^n, that is the cardinality of all possible subsets of a set with n elements.

Thus it becomes interesting to know what could be an "expected" value for n, and the only sensible way to provide an estimation is again to rely on the library itself.

The table in Figure 3 gives, for several classes of terms, the repartition of the theorems according to the number n of constants in their conclusion (comprising the constant with position MainConclusion). The first column refers to the whole library; the other columns are snapshots relative to the statements with the specified MainConclusion.

The average number of different constants appearing in the conclusion of statements of the COQ library is surprisingly low: 3.3 Things are a bit worse for equality statements, but we are also counting the type, and thus, according to Remark 1 the value in Figure3 should be decremented by one.

Let us finally remark the quite anomalous behaviour of the setoid equality *cs_eq* and the algebraic less or equal relation *leEq* of the CCorn development in Nijmegen. The problem is that in this case the formal statements are cluttered by a long series of coercions taking in charge the hierarchy of structures typical of algebraic developments. It could be possible that coercions should deserve a special treatment for indexing purposes (we are currently investigating the issue).

6 Prefixes

Not every subset of the constants appearing in the conclusion of a goal G should be considered for the generation of *exactly constraints*: we should only take into account subsets appearing in some prefix-tree of G. We say that t' is a prefix-tree (simply, a prefix) of t if t' can be obtained from t by pruning subtrees (replaced by a special "empty" tree $\perp$); this is equivalent to the following definition

Definition 4. *The set $P(t)$ of prefixes of a tree (a first order term) t is defined by the following rules:*

- $\perp \in P(t)$ *for any tree t;*
- *if $t = c(t_1, \ldots, t_n)$, and for $i = 1, \ldots, n$ $t'_i \in P(t_i)$, then $c(t'_1, \ldots, t'_n) \in P(t)$.*

Example 6. Consider the goal $S(0) * x \leq y$. The set of prefixes (neglecting variables) is $\{\perp, \perp \leq \perp, \perp * \perp \leq \perp, S(\perp) * \perp \leq \perp, S(0) * \perp \leq \perp\}$. In particular, we cannot have a prefix containing S but not $*$, nor a prefix containing 0 but not S. Any matching goal G' must satisfy the same condition, thus imposing to have equality in MainConclusion, the set of *exactly constraint* for $S(0) * x \leq y$ is shown in the following table:

exactly constraint	#result
$\emptyset$	60
$\{*\}$	14
$\{*, S\}$	4
$\{*, S, 0\}$	3
total	81

7 Mixing *exactly constraints* and *atleast constraints*

When, even restricting to prefixes, the total number of *exactly constraints* is too big to allow a feasible computation, we may always employ *exactly constraints* up to a given number $k-1$ of occurrences, and then requiring *atleast constraints* for all possible subsets of k constants. We call this constant k the *just-factor*. The important fact is that we have to apply *atmost constraint* only to the union of the result sets of the *atleast constraints*.

n	all	$=$	$=_{nat}$	$=_R$	cs_eq	$\leq$	Rle	$leEq$
1	12635	195			9	60	36	27
2	6001	332	81	31	10	84	58	1
3	4049	616	132	57	8	54	78	94
4	3000	1017	203	122	13	24	23	31
5	2323	763	174	90	3	19	16	15
6	1226	395	78	45	22	10	10	10
7	875	216	48	38	22	6	8	5
8	769	135	20	41	44	1	2	9
9	496	65	15	35	69	1	4	13
10	1167	54	8	25	74	2	0	36
11	469	50	5	10	94	0	5	9
12	444	23	1	4	83	2	1	40
13	312	24	3	9	69	0	1	34
14	228	18	3	2	68	0	1	23
15	155	12	0	3	51	0	1	15
16	89	4	0	0	29	0	1	6
17	90	8	0	1	21	1	0	8
18	66	5	0	1	15	0	0	12
19	56	5	0	0	7	1	0	9
20	38	6	2	0	13	0	0	6
21	51	10	0	1	7	0	0	2
22	28	4	0	0	9	0	0	3
23	27	9	0	0	4	0	0	1
24	15	5	1	0	1	0	0	0
25	14	6	1	0	1	0	0	0
26	15	3	0	0	4	0	0	4
27	8	0	0	0	2	0	0	0
28	2	0	0	0	0	0	0	0
...								
40	3	0	0	0	0	0	0	
$tot.$	34661	3978	775	515	753	263	245	413
$av.$	3.3	4.9	4.7	5.8	10.2	2.7	3.5	8.9

Fig. 3. Repartition of theorems with respect to the number of different constants in their conclusion

Remark 2. The algorithm described in [13] can be considered as a degenerate case of our approach when $k = 0$.

Remark 3. While the result sets of different *exactly constraints* are always disjoint, this in not any longer the case for *atleast constraints* and it is worth to add an additional pass to drop repetitions.

Example 7. Consider again the previous example. If we decide to stop with a just-factor $k = 2$, we should consider $\{*, S\}$ as an *atleast constraint*, getting back 14 candidates with a single query (against the 4+3 theorems obtained with two queries as above). The extra 7 candidates could be filtered out by the application of the *atmost constraint*, but these would require 14 more queries (one for each statement in the result set of the *atleast constraint*).

Example 8. Let us consider the goal $(sinPI/3) =_R (cosPI/6)$. In COQ, the real number n is treated as the sum of n occurrences of 1 ($R1$); so the goal contains the 8 constants $=, R, sin, cos, div, PI, Rplus, R1$. For k=0 we get the usual 3978 candidates with equality in MainConclusion. The case k=1 amounts to restrict the attention to the realm of real numbers, reducing the total number of candidates to 195 (just containing equality) plus 515 statements containing R (and possibly other things). On the latter 515 statements we must compute *atmost constraint.*

For k=2, we have the following set of constraints:

exactly constraint	#result
$\emptyset$	195
$\{R\}$	31
atleast constraint	#result
$\{R, sin\}$	59
$\{R, cos\}$	65
subtotal	124
subtotal (distinct)	93
total (distinct)	319

Note that in this case the intersection of the result sets of the two *atleast constraints* is not empty (it contains 31 elements). At the end, we have just to compute *atmost constraints* for 93 elements: this means that in passing from a just-factor 0 to a just-factor 2 we already reduced the number of queries from $1 + 3978 = 3979$ to $4 + 93 = 97$ (2.4 %).

For higher values of the just-factor things can still improve but in a much less remarkable way. In particular, for k=3, the total number of queries is $7+51 = 58$, and for $k = 4$ is $12 + 35 = 47$.

exactly constraint	#result
$\emptyset$	195
$\{R\}$	31
$\{R, sin\}$	0
$\{R, cos\}$	0
atleast constraint	#result
$\{R, sin, cos\}$	31
$\{R, sin, div\}$	21
$\{R, cos, div\}$	23
subtotal	75
subtotal (distinct)	51
total (distinct)	277

Note also that for k=4 the subtotal of *atleast* is even greater than for k=3, since we have a big number of largely overlapping sets; of course the total number of *distinct* elements is eventually either stable or decreasing.

exactly constraint	#result
$\emptyset$	195
$\{R\}$	31
$\{R, sin\}$	0
$\{R, cos\}$	0
$\{R, sin, cos\}$	0
$\{R, sin, div\}$	0
$\{R, cos, div\}$	0
atleast constraint	#result
$\{R, sin, cos, div\}$	12
$\{R, sin, div, PI\}$	16
$\{R, sin, div, Rplus\}$	19
$\{R, cos, div, PI\}$	16
$\{R, cos, div, Rplus\}$	23
subtotal	86
subtotal (distinct)	35
total	261

A sensible heuristic may be based on the following consideration. Suppose that the Goal G has n constants in position InConclusion. Then for any k, the number of different *exactly constraints* with more than k elements is (roughly) bounded by $p(k) = \sum_{i=k+1}^{n} \begin{pmatrix} n \\ i \end{pmatrix}$. If $p(k)$ is less than the cardinality of the result set of *atleast constraints* for $k-1$ elements, it may look worth to generate and compute *exactly constraints* with k elements; passing then to k+1. As a simpler, further approximation, instead of $p(k)$ we may just use the upper bound 2^n.

For the previous example we would have the following situation (we take $n = 6$, due to the special treatment of the type R; for the same reason, in the case of the equality, $p(k)$ should be compared with the result set of the *atleast constraint* at level k, and not $k-1$).

k	$p(k)$	$\# - must_k$
1	63	515
2	57	93
3	42	51
4	22	35

Our heuristic would suggest to go on (that is indeed the best decision). Using $2^n = 64$ instead of $p(k)$ we would have stopped with just-factor of 3, that is in any case a sensible choice. More generally, taking a just-factor of 3 for equality statements and of 2 for all other statements seems to be a very simple but quite effective politics.

The evaluation and tuning of the above heuristics is currently under way.

8 Conclusion and future work

In this paper we introduced a very simple but extremely effective extension of the approach described in [13] for the retrieval of statements matching a given goal G. Our extension is based on the notion of *just constraint*, whose result is a list of theorems containing *exactly* the constants listed in the constraint. The key observation is that a *just constraint* can be simply implemented as a join of the *must constraint*, with the fact that the statement must contain a number of constants equal to the cardinality of the *must constraint*.

Even with very simple heuristic mehtods, our approach reduces the average matching time of more than ten times.

There are also additional benefits.

First of all, the approach described in [13] becomes practically useless in case the goal to match contains metavariables, since in this case the *onlyc* (responsible for the actual filtering) makes no sense. We have a similar problem, in our case, but much less dramatical since the *onlyc* would be applied (if required) to a much more smaller sets of candidates, and renouncing to this filter does not look problematic.

A second important benefit is that our approach allow to order the resulting candidates according to the dimension of the prefix of G they are matching: the general idea is that a statement matching a bigger prefix is likely to be a better candidate (thus, e.g., in the case of the equality in MainConclusion, leaving symmetry and transitivity as a last resource).

From the point of view of automatic proving, our technique, although far from being resolutive, has substantially improved the automatic tactics under development in HELM, solving part of the problems, and helping to put in evidence the new and crucial weakness of the current approach: the large number of theorems having just one constant in MainConclusion and nothing else. For instance, in the case of equality we have 195 different theorems with a conclusion of the kind $n =_X m$. Typical examples are, for instance[2]:

Singleton_inv stating that x is equal to y provided x is an element of $\{y\}$:

$$\forall x, y : U.x \in \{y\} \rightarrow x =_U y$$

fun_item stating that u is equal to v provided that they are the n-th element of a same list e:

$$\forall u, v : A.\forall e : List.\forall n : nat.(item\ u\ e\ n) =_A (item\ v\ e\ n) \rightarrow u =_A v$$

Lub_is_unique stating that $a =_U b$ provided that (U is a conditionally complete partial order and) a and b are least upper bounds of a same subset C of U.

$$\forall C : (Ensemble\ U).\forall a, b : U.(Lub\ C\ a) =_U (Lub\ C\ b) \rightarrow a =_U b$$

Theorems of the previous kind always match any equality goal; if moreover, as it is often the case, they generate one or more equality subgoals, we may easily expect an exponential explosion of the search-tree with a branching factor close to 100!

Several heuristic may be imagined to solve this problem, but this a completely different story that we plan to tell in a forthcoming paper.

References

1. Mathematical Markup Language (MathML) Version 2.0, W3C Recommendation 21 February 2001, http://www.w3.org/TR/MathML2
2. OMDOC: A Standard for Open Mathematical Documents, http://www.mathweb.org/omdoc/omdoc.ps
3. The OpenMath Standard, http://www.openmath.org/cocoon/openmath/standard/index.html
4. A.Asperti, G.Goguadze, E.Melis. Structure and Meta-Structure of Mathematical Documents. Deliverable n.D1b of Project IST-2001-33562 MoWGLI, http://www.mowgli.cs.unibo.it/.
5. G.Bancerek, P.Rudnicki. Information Retrieval in MML. Proceedings of the second International Conference on Mathematical Knowledge Management, Bertinoro, Italy, February 2003. LNCS 2594, pp. 119-132.
6. J.Christian. Flatterms, Discrimination Nets and Fast Term Rewriting. Journal of Automated Reasoning, 10:95-113, 1993.
7. D.Delahaye, R. Di Cosmo. Information Retrieval in a Coq Proof Library using Type Isomorphisms. In Proceedings of TYPES 99, Lökeberg. Springer-Verlag LNCS, 1999.
8. R. Di Cosmo. *Isomorphisms of Types: from Lambda Calculus to Information Retrieval and Language Design*, Birkhauser, 1995, IBSN-0-8176-3763-X.

[2] It is surprising how some of these statements may look odd and amusing when read in the “wrong” direction

9. H.Ganzinger, R.Nieuwehuis, P.Nivela. Fast Term Indexing with Coded Context Trees. Journal of Automated Reasoning. To appear.
10. G.Goguadze. Metadata for Mathematical Libraries. Deliverable n.D3a of Project IST-2001-33562 MoWGLI, http://www.mowgli.cs.unibo.it/.
11. P.Graf. Substitution Tree Indexing. Proceedings of the 6th RTA Conference, Springer-Verlag LNCS 914, pp. 117-131, Kaiserlautern, Germany, April 4-7, 1995.
12. F.Guidi. Searching and Retrieving in Content-Based Repositories of Formal Mathematical Knowledge. Ph.D. Thesis in Computer Science, Technical Report UBLCS 2003-06, University of Bologna, March 2003.
13. F.Guidi, C.Sacerdoti Coen. Querying Distributed Digital Libraries of Mathematics. In Calculemus 2003, Aracne Editrice S.R.L., T.Hardin and R.Rioboo editors, ISBN 88-7999-545-6, pp. 17–30.
14. F.Guidi, I.Schena. A Query Language for a Metadata Framework about Mathematical Resources. Proceedings of the second International Conference on Mathematical Knowledge Management, Bertinoro, Italy, February 2003. LNCS 2594, pp. 105–118.
15. M.Kohlhase, A.Franke. MBase: Representing Knowledge and Context for the Integration of Mathematical Software Systems, Journal of Symbolic Computation 23:4 (2001), pp. 365–402.
16. W.McCune. Experiments with discrimination tree indexing and path indexing for term retrieval. Journal of Automated Reasoning, 9(2):147-167, October 1992.
17. I.Schena. Towards a Semantic Web for Formal Mathematics. Ph.D. Dissertation, Department of Computer Science, University of Bologna, Italy. January 2002.