INFORMATION SOCIETY TECHNOLOGIES
(IST)
PROGRAMME

Project IST-2001-33562 MoWGLI

Report n. D2.g
Tools for automatic extraction of Metadata

Main Author:
A. Asperti, F.Guidi, C.Sacerdoti Coen

Project Acronym: MoWGLI
Project full title: Mathematics On the Web: Get it by Logic and Interfaces
Proposal/Contract no.: IST-2001-33562 MoWGLI

Contents

1 Overview

This document describes both the class of metadata automatically extracted from mathematical documents in order to enhance their location by means of searching agents, and the software tools and technologies which have been tested and used for their extraction and storage.

2 Automatically Syntetized Metadata

The lait-motif of the MoWGLI project is the exploitation of a content description of mathematics, mostly relying on XML-technology and in the perspective of the Semantic Web.

As already widely discussed in report D1b about the "Structure and Meta-Structure of Mathematical Documents", mathematics is a a domain of knowledge with a particualry reach and articulated structure, spanning from the low-level, microscopic description of formulas, to the macroscopic organization of theories, and of documents. Each of these layers may deserve its own metadata, as it is also reflected in the classification of metadata of report D3b: MoWGLI's Metadata Model. According to our model, we identify three main classes of Metadata:

administrative metadata such as title, author, date of creation, copyright etc., clearly pertaining to the macrostructure;

mathematical metadata mostly covering basic dependencies relations between mathematical items;

application dependent metadata several kinds of semantic dependencies which all the applications agree at. Finally, every application can have specific additional metadata in a separate module.

In general, administrative metadata must be explicitly added either by the author or in any case by an external user. On the other side, both mathematical metadata and some kind of application dependent metadata can be automatically extracted from the source document, provided the source is encoded in a sufficiently structured (i.e. contenutistic) way.

In this report we are merely concerned with this latter class of metadata, and in particular on metadata for mathematical *statements* (and *formulas*). The aim of this Metadata is to provide an "approximation" of the actual content of the mathematical item, in order to aid its description and location.

3 Metadata as Approximations

The forthcoming development of Web services for automated deduction and computation put high pressure for the development of alternative effective methods to retrieve a single result. Luckily, at the same time more and more mathematical knowledge is put on the web in a structured form, thanks to the introduction of standard markup languages [3, 1, 2] for the encoding of mathematical formulae and expressions. Once the statements in a library are marked up, say, in OpenMath, we can use standard pattern-matching or unification techniques to retrieve a theorem given a part of its statement. For example, the pattern $\forall x : ?_1 . \forall y : ?_1 . (?_2\ x\ y) = (?_2\ y\ x)$ will retrieve any theorem stating the commutative property of a binary operation. The question marks are *metavariables*, which are non-linear placeholders for any (well-typed, if enforceable) sub-expression closed in the context.

To resolve a query based on pattern matching, the trivial approach consists in iterating the matching operation over the whole library. This solution does not scale well and it can not be applied in those cases where the library itself is physically distributed and no code can be run on remote servers[1].

To solve this problem, we propose an approach based on four distinct phases:

- *Data-mining*: using a spider, we extract a small set of automatically computed metadata from each theorem or definition in the distributed library. These metadata are devised to capture some of the invariants of the chosen matching operation.

- *Pattern compilation:* when the user submits a pattern, we extract a set of constraints over these metadata from the pattern and we generate a low-level query[2] starting from these constraints.

- *Filtering:* we execute the generated query over the database built by the spider, obtaining our set of candidates.

- *Matching:* we iterate the matching operation only on the set of candidates.

If the filtering operation is both quick and correct (in the sense that no good candidates are dropped), we are able to achieve both accuracy and performance.

Since the metadata must capture the invariants of matching, we need to identify several kinds of metadata. In the following sections 3.1–3.3 we will progressively consider several examples of matching criteria and we will identify the related metadata. We will also consider the issue of correctness and a tradeoff between correctness and performance. In section 4 we will present the generic set of metadata adopted in MoWGLI.

3.1 Use Case 1: Finding the Applicable Theorems

The first use case that we consider is the problem of retrieving all theorems which can be applied to prove a given goal. A goal is a sequent made of several hypotheses and a conclusion. A theorem can be applied if its conclusion, where every bound variable has been replaced with a metavariable, matches the conclusion of the sequent. We restrict ourselves to the case of *first order matching*, i.e. only one expression (the conclusion of the theorem) has metavariables in it; and the expression is rigid. For example:

- The theorem $\forall n.\ \forall m.\ \forall a.\ a \geq 0 \Rightarrow n \leq m \Rightarrow an \leq am$ can be applied to prove $2x \leq 2(x+1)$ since the pattern $?a?n \leq ?a?m$ matches $2x \leq 2(x+1)$.

- The theorem $\forall n.\ \forall m.\ \forall a.n \leq m \Rightarrow an \leq am$ can not be applied to prove $x \leq 2x$ since the pattern $?a?n \leq ?a?m$ does not match $x \leq 2x$.

[1]Indeed, if the library is distributed, we have only two possibilities: either we can perform the matching where the theorem is located or we need to download it. In the second case the download times make the operation unfeasible already for very small libraries. In the first case we need to install software were the library is published, which is something we would rather avoid.

[2]In our prototype, we have also developed an ad-hoc query language [6], that is however independent from the specific set of metadata. Thus, generating a query in other languages is surely possible.

- The induction principle over natural numbers $\forall P.\ (P\ 0) \Rightarrow (\forall n.\ (P\ n) \Rightarrow (P\ (n{+}1))) \Rightarrow \forall n.\ (Pn)$ does not match $(is\_prime\ 5)$ since the pattern $(?P\ ?n)$ is not rigid.

Now we identify a set of invariants of the matching procedure that we want to capture as metadata:

- The constant in *head position* in the conclusion of the theorem. Since the pattern is rigid, any matching term must have the same head constant of the pattern.
- The list of constants in the conclusion. Since we are doing matching and not unification, any constant in the pattern must also occur in the matching term (since instantiating a metavariable can not remove an occurring constant).

Thus we identify the following set of metadata: for each theorem in the library we remember the name of the constant in head position in the conclusion and the names of all the constants occurring in the rest of the conclusion. Now, to find the theorems that are candidates for the query "give me all the theorems that can be applied to this goal", we simply compute these metadata over the conclusion of the goal and we look for every theorem in the library such that:

- the "constant in head position" metadata is the same as the computed one
- the set of "constant in conclusion" metadata is a subset of the computed one

Since our metadata reflects the matching invariants, the filtering operation is correct, i.e. no pattern that matches the goal conclusion is filtered out.

Let's see how it works in practice: given the goal $2x \leq 2(x+1)$

- $?a?n \leq ?a?m$ is a candidate that will match
- $?n \leq ?n^2$ is not a candidate, since the power operator does not occur in the goal
- $?n?a \leq ?m?a$ is a candidate that will not match (a *false match*)

We could also capture a much stronger invariant, that subsumes both our previous invariants: *the rigid skeleton of the term is preserved by metavariable substitutions*. Nevertheless, the stronger invariant has two drawbacks: the first one is that it produces complex metadata (contexts, i.e. terms with holes) that can not be handled efficiently within the relational database model or the RDF model (which is often reduced to the relational model); the second drawback is that the invariant is probably too strong. In particular, it would reject the third candidate of the previous example. Even if that candidate turns out to be a false match, it is pretty obvious (at least to a human user) that the statement found is useful to proceed in the proof. Very often, indeed, the user is also interested in theorems that can "almost" be applied, since they can reveal an error in the proof (when a required assumption is missing) or they can suggest a different way to prove the goal.

Of course, the danger of not adopting the stronger invariant is that of finding too many useless false matches. To evaluate our invariants, we tried several queries over the library of the Coq proof-assistant (about 40.000 theorems). Even for queries involving only frequently used notions (e.g. algebraic operations), the accuracy of the filtering phase was always very high.

Let's now face the issue of performance, considering for instance the goal $2x \leq 2(x+1)$ which generates the constraints:

- $\leq$ `in head position in the conclusion`
- `only` 2 `and` + `in other positions in the conclusion`

The generated query (in a pseudo-language inspired by MathQL level 1 [6]) is:

```
select every t in the library such that
 t.head = '<=' and t.in_conclusion subset of {'2','+'}
```

Since the second test can not be performed in a relational DB, we are forced to extract from the DB all the theorems `t` that satisfy the first constraint together with their `in_conclusion` field and then iterate the second test. Thus, the computational cost is linear over the size of the result set of the first test. If the head position of the goal is a frequently used constant, the result set is large. For example, there is an equality in head position in the 11% of the theorems of the library of Coq, i.e. in 3660 theorems. To reduce the computational cost of the query, we propose a user-controlled tradeoff with the accuracy of the query: the user provides an additional set of constraints consisting in a list of constants that *must* appear in the conclusion of the matched theorems. To avoid confusion, we will call these constraints *must constraints* and we rename the previous set of constraints *only constraints*. In order to retrieve a non-empty subset of results, the set of constants occurring in the *must constraints* must be a subset of the constants occurring in the *only constraints*.

To better understand the *must constraints*, let us consider the goal $2x \leq 2(x+1)$. The generated *only constraints* are $\leq$ in head position and 1,2,$*$ and + in other positions in the conclusion. Let's suppose that the user choice of *must constraints* is $\leq$ in head position and $*$ in other positions. Then

- $?a?n \leq ?a?m$ is a candidate that will match (a true match)
- $?n?a \leq ?m?a$ is a candidate that will not match (an interesting false match)
- $?n \leq ?n+1$ is not a candidate since it satisfies the *only constraints* but not the *must constraints* ('$*$' does not occur). Anyway, it would have been an uninteresting false match.

Note that the *must constraints* are meaningful to the user: in the previous example, the *must constraints* are stating that the user is interested in a property of multiplication. Thus, it is not difficult to provide an user-friendly interface to select these constraints. It is also easy to provide some heuristics to automatically chose the *must constraints*[3].

A non-empty set of *must constraints* reduces the accuracy of the query, since some theorems that could have been applied are not found any longer. For example, the previous query is unable to find the transitivity principle for the $\leq$ relation, since it states nothing about multiplication[4]. The performance gain achieved can be remarkable. To understand why, let us see the query generated for the last example:

```
let S =
 select every t in the library such that
  t.head = '<=' and '*' occurs in t.in_conclusion
```

[3] As an example, we provide a predefined heuristic that chooses only the constants occurring in the first n levels of the syntactic tree of the goal thesis.

[4] In our experience, very often loosing these "very general" theorems can be considered a feature.

```
in
 select every t in S such that
  t.in_conclusion subset of {'1','2','+','*'}
```

The first part of the query (the first `select`) can be performed by a single call to the database. Then, as before, we need to iterate the second test over the set returned by the first select. The more *must constraints* we have, the smallest the set `S` will be and the smallest the time required to perform the query will be.

3.2 Use Case 2: Finding the Elimination Principles

Let us consider queries for the retrieval of all the elimination principles over a given datatype, that is elimination principles that allow to prove a generic property P over an element n of a datatype T by proving that P holds in several cases, some of them under additional inductive hypotheses. For example, there are several elimination principles over lists of natural numbers. Among them, we find the structural induction principle $\forall P.\ (P\ \mathit{empty}) \Rightarrow (\forall l.\ \forall n.\ (P\ l) \Rightarrow (P\ n :: l)) \Rightarrow \forall l.\ (P\ l)$ and the induction principle over ordered lists $\forall P.\ (P\ \mathit{empty}) \Rightarrow (\forall l.\ \forall n.\ (\mathit{ordered}\ l) \Rightarrow n \leq (\mathit{head}\ l) \Rightarrow (P\ n :: l)) \Rightarrow \forall l.\ (\mathit{ordered}\ l) \Rightarrow (P\ l)$.

Note that, since the pattern $(?P\ ?x)$ is not rigid, every elimination principle can always be used to progress in any proof, independently of the goal. Since there exist literally thousands of elimination principles, it is not a good idea to include them as results of the query of the first use case we considered[5]. Nevertheless, we are interested in giving the user the possibility to retrieve all elimination principles over a given datatype. In particular, given a datatype T, we want to retrieve all the theorems whose statement matches[6] the schema:

$$\forall \overline{x} : \overline{S}.\ \forall P : T \rightarrow \mathit{Prop}.\ \forall \overline{y} : \overline{S'}.\ \forall x : T. \forall \overline{z} : \overline{S''}.\ (P\ x)$$

We identify the following set of constraints:

- The sort `Prop` (i.e. the type of every proposition) must occur in the head position of an hypothesis which is a product[7] of length 1.
- The datatype must occur in the head position of an hypothesis which is a product of length 0 and and also in another hypothesis (not in main position).
- The head of the conclusion must be an occurrence of a bound variable.

Every constraint captures a matching invariant, since context metavariables substitution can not change the shape of the conclusion nor the shape of an hypothesis (which are all rigid parts). As in the previous use-case, there exist stricter invariants: the set of constraints identified here has been fine-tuned by hand to be able to retrieve also a few useful false

[5]That is the reason why we constrained ourselves to first order unification in section 3.1.

[6]We do not specify formally here the kind of match we are interested in. Anyway, note that the "vector parts" of the schema play the role of *context metavariables*, in the sense that they are contexts of formulae that must be instantiated with a context made of repeated quantifications or implications, concluded by a single contextual hole.

[7]The terminology comes from type-theory and is derived by the Curry-Howard isomorphism, where universal quantifications are seen as (dependent) products. A product of length n is a term of the form $\forall x_1.\ \ldots\ \forall x_n.\ t$ where t does not start with an universal quantification. An implication is just a special case of universal quantification: it is a non-dependent quantification where the bound variable does not occur in the sub-expression.

matches. Even in this case, though, the final set of invariants was identified quite soon and without any major effort.

Note that, unlike the previous use case, we derive the matching pattern from the user input and not from the statements of the theorems in the library. In section 3.1 we had just one term to match against several different patterns; here we have just one pattern to match against the theorems in the library. This duality is reflected in the way the constraints are used to generate the query: in the previous case the identified constraints were *only constraints* (only the constants occurring in the goal can occur in the statement of the theorem); in this case the constraints are *must constraints* (all the constants in the pattern must occur in the statement of the theorem)[8].

The query generated from the constraints is the following:

```
select every t in the library such that
 Prop occurs in head position at depth 1 in an hypothesis of t and
 T occurs in head position at depth 0 in an hypothesis of t and
 T occurs not in head position in an hypothesis of t and
 a bound variable occurs in head position in the conclusion of t
```

Unlike the case of section 3.1, the generated query is already very efficient: this time we do not need any trade-off to improve the performance.

3.3 Use Case 3: Finding the Proofs of a Statement

The last use case we present is locating a proof (or a definition) whose statement (i.e. its type) is known. The query seems quite uninteresting, but the inability to perform it may have very bad implications in terms of waste of efforts. Indeed, every time we need to prove a lemma, we would like to issue a query that checks if somebody else already proved it; the Coq system provides no way to perform this operation and, as a result, it is not at all unusual to find simple arithmetical properties proved five, eight or even more times by different persons working in remotely located teams[9].

The kind of matching we are interested in is extremely simple: no metavariables are present in both the statement of the theorem and the statements in the library. Close enough false matches, though, play an important role: for example, when looking for a proof of $\forall n.\ n = n{*}1$, any of the following statements would be satisfactory: $\forall n.\ n = 1{*}n$; $\forall n.\ n{*}1 = n$; $\forall n. 1{*}n = n$

We identify the following set of metadata, which constitute a close approximation of the first two levels of the abstract syntax tree of the pattern:

- For each occurrence of a constant, we record its *position*, as an element of the following set: {`MainConclusion, InConclusion, MainHypothesis, InHypothesis`}. The prefix `Main` means that the constant occurs in head position in the `Conclusion/Hypothesis` of the theorem.

[8]Both sentences are instances of the following invariant that is true for any kind of matching: "all the constants in the pattern must occur in the matched term".

[9]E.g. the proof of the fact that 1 is the neutral element of multiplication was given five times. Of course everybody suspects that a proof should already exist. Nevertheless, proving it again is faster than guessing who gave the proof and in what library. As a consequence, the theorem is proved again and the new proof is stored in the library under development by the user, which is likely to be unrelated to arithmetical properties. Thus, without an automatic searching procedure, it is virtually impossible to know that a proof of the theorem was put in that library.

- For each occurrence of a bound variable, we record its *position* in the set {`MainConclusion, MainHypothesis`}. Since bound variables occur almost in every statement, they are not very effective for searching. Thus we are interested only in second order or higher order variables, that are much less frequent. We syntactically recognize a subset of these variables by the fact that they occur in head position: these are the only occurrences we record in the metadata.

- For each occurrence of a sort *Prop* or *Set*[10], we record its *position* in the set {`MainConclusion, MainHypothesis`}

- For each occurrence in {`MainConclusion, MainHypothesis`} we record its *depth*, i.e. the number of universal quantifiers in the statement or hypothesis.

On the previous metadata we automatically impose both *must constraints* and *only constraints*[11] to achieve both performance and accuracy at the same time[12]. As in the previous cases, our experiments over the Coq library show that the chosen set of constraints is a good compromise between the strictness of the query (in terms of false matches) and the need to get results close to the expected ones (interesting false matches).

Expert users of the system can also edit the set of generated constraints, relaxing both the *must constraints* or the *only constraints*. As a result, they are able to answer similar queries such as: *Retrieve every binary relation* or *Retrieve any n-ary relation over natural numbers*.

Instead of describing here these and other use cases with the introduction of new ad-hoc constraints, we will present in the next section a generic set of constraints which is general enough to subsume the constraints identified in the previous use-cases. Moreover it is relatively stable, since no new constraints have been added in the last few months. The current work is focused on the automatic generation of larger and larger classes of queries based on these constraints. For example, we are now able to write by hand queries to retrieve every theorem in the library of Coq stating the associative or commutative property of any binary operator on any datatype. The number of false matches for these kind of queries is surprisingly low[13]. It is surely possible to generate these queries automatically starting from the formal definition of the wanted property.

4 A Generic Class of Metadata and Constraints

There is an obvious common pattern for the metadata identified in the use cases above: they provide an approximate description of the abstract syntax tree of the statements[14] in the library in terms of the constants and bound variables occurring in them, together with the occurrence locations. This observation leads us to describe our metadata model using *objects*

[10]*Prop* is the type of every proposition and *Set* is the type of any data-type. They are used to capture second order and higher-order quantifications over properties and types. They can also be used to capture axiom schemas in first-order theories.

[11]With *only constraints* we mean that every occurrence of a constant, bound variable or sort in the matched term must occur in the *only constraints*. In section 3.1 this test was restricted to the occurrences in the conclusion.

[12]There is no tradeoff here: the *must constraints* can not filter out any good candidate since there are no metavariables in the pattern.

[13]For example, looking for every associative property in the Coq library gives 82 good matches and just 6 false matches.

[14]With statements here we mean the statements of axioms and theorems, and also the types of definitions.

having a list of *references*: the objects represent the statements in the library at the metadata level, i.e. they are the entities the metadata are about, while the references are the metadata describing the occurrences of constants and bound variables in the statements. As we said, an object may have three kinds of references. Namely:

- `refObj` describes the occurrence of a primitive or defined constant in terms of an `occurrence`, a `position` and a `depth` (see below).
- `refSort` describes the occurrence of a sort in terms of a `sort`, a `position` and a `depth` (see below). A sort is the type of propositions or data-types in higher-order logics (i.e. the second order quantification "*for every property P*" can be thought as the typed universal quantification $\forall P : \mathit{Prop}$). Sorts can also be used to capture axiom schemas in first-order theories.
- `refRel` describes the occurrence of a bound variable in terms of a `position` and a `depth` (see below).

The value of a reference is a record made of several fields, whose semantics is:

- `occurrence` specifies the referred defined constant. Its value is an object.
- `sort` specifies the referred sort. Its value belongs to a predefined collection of entities representing the possible sorts at the metadata level.
- `position` specifies the position of the described occurrence in the statement. Its value belongs to a predefined collection of entities representing the possible positions at the metadata level. According to the previous use cases, we identified the following positions:
  - `MainHypothesis`: in head position of a statement premise.
  - `InHypothesis`: in another position of a statement premise.
  - `MainConclusion`: in head position of the statement conclusion.
  - `InConclusion`: in another position of the statement conclusion.
  - `InBody`: not in the statement (for instance in its proof).
- `depth` specifies a depth index associated to the position of an occurrence in the statement. Its value is a natural number. As we saw in the use cases, the `depth` of a reference is defined only when its `position` is set to `MainHypothesis` or to `MainConclusion` and it represents the number of premises of that hypothesis or conclusion.

The above description shows that every reference has some fields for locating the corresponding occurrence in the statement (`position` and `depth`) and may have one field specifying the referred entity (`occurrence` and `sort`)[15].

The queries we generate on these metadata are once again inspired by the use cases we saw in the previous sections. Our general approach concerning query generation is that complex queries should be obtained composing basic queries generated from the imposed atomic constraints on the wanted objects.

We identify the following two classes of basic queries:

[15]`refRel` does not have this field because, up to now, we did not meet any use case in which we need to discriminate between the occurrences of different bound variables.

A. **The wanted objects must have a reference to a given object R (or to a given primitive constant S or to a bound variable) in a given position P with a given depth index D.**

   Here we search for the objects having the assigned values in corresponding fields of at least one of their references. Using the pseudo-language exploited in the previous sections, a query of this class looks like:

```
select every t in the library such that
 the set {
  select every r in t.refObj such that
   r.occurrence = R and r.position = P and r.depth = D
 } is not empty
```

   This class of queries includes what we called the basic *must constraints*.

B. **The wanted objects may have a reference to an object (or to a primitive constant or to a bound variable) only if its position is not included in a given set U of positions, or if it concerns a given object R (or a primitive constant S, or a bound variable) in a given position P with a given depth index D.**

   Here we query the objects such that there are no references whose fields do not have the assigned values. A query of this class may look like:

```
select every t in the library such that
 the set {
  select every r in t.refObj such that
   not (r.occurrence = R and r.position = P and r.depth = D)
   and r.position belongs to U
 } is empty
```

   This formulation exploiting the logical double negation is more convenient, since it can be expressed more easily in SQL. This class of queries includes what we called the basic *only constraints*.

Note that the parameters R, S, P, D, U are always optional.

Coming to the issue of how the above queries should be composed, let us suppose that we are interested in finding the objects satisfying a given set (call it K) of "basic constraints". If the set K contains p constraints in the class A, the wanted objects are meant to satisfy all of them. Since a query of the class A (a *must constraint*) is of the general form

```
M(i) =
 select every t in the library such that
  the set m(i) is not empty
```

the composition of M(1) . . . M(p) is:

```
M = M(1) + ... + M(p) =
 select every t in the library such that
  the set m(1) is not empty and ... and the set m(p) is not empty
```

Dually if the set K contains some constraints in the class B, the wanted objects are meant to satisfy at least one of them. Since a query of the classes B (an *only constraint*) is of the general form:

```
O(j) =
 select every t in the library such that
  the set o(j) is empty
```

the composition of O(1) ... O(q), provided that they concern the same kind of reference, say `refObj`, is:

```
O = O(1) + ... + O(q) =
 select every t in the library such that
  the set o(1) intersected ... intersected o(q) is empty
```

and the same holds for the queries O'(1) ... O'(q') concerning `refSort` and for the queries O"(1) ... O"(q") concerning `refRel`. Now the final query is obtained composing M, O, O' and O" conjunctively:

```
M + O + O' + O'' =
 select every t in the library such that
  the set m(1) is not empty
    and ... and the set m(p) is not empty  and
  the set o(1) intersect ... intersect o(q) is empty  and
  the set o'(1) intersect ... intersect o'(q') is empty  and
  the set o''(1) intersect ... intersect o''(q'') is empty
```

5 Automatic Metadata Generation

The actual extraction of metadata and their actual storage into a data base is possibly the most trivial and less exciting part of the work. In spite of this, the prototype tool for metadata extraction was completely rewritten several times, mostly for performance reason. As a matter of fact, even if the generation of metadata is conceived in MoWGLI as a batch process, and does not strictly require a "real time" speed, we should auspiciously try to keep the execution cost of this operation as low as possible. Considering that:

1. the library of documents we are currently using as a test-bench is reasonably big (40000 documents, taking, compressed, about 0.5 GigaBytes of memory);

2. each of these documents is a higly structured XML file, which is typically composed by thousands of elements, whose nesting nesting depth is often larger than several hundreds (the average size, compressed, is about 13K, but we also have individual documents of a few megabytes);

3. the currently available xml technology for parsing and processing (comprising libxsl that is the more performant processor available so far) is not particularly fast (not to say plainly slow)[16];

[16]The size of the library and of the single documents is not a problem *per se* - memory is cheap, indeed - but for the cost of parsing and processing; any operation comporting even a small, more than linear cost, as is frequently the case with XML technology - becomes practically unfeasible

it is clear that nothing should be assumed for granted.

Our first implementation used sthylesheets: it is extremely simple to express the patterns locating metadata in the source document as XSLT templates; as output of this process, we directly produced an RDF-compliant encoding of part of the metadata (forward references). In a second phase we inverted this relation (backward references, also used for the visualization of dependencies graphs), and finally stored everything in a relational databases (PostgreSQL).

Unfortunately, the whole process was far too slow to be acceptable. Processing a small subset of the library (6000 documents, available at the time the test was done) required about 12 hours.

The second solution was to drop xslt and to perform the extraction of metadata in a more efficient language; since, for different purposes, we already had an XML parser into OCAML (producing a concrete OCAML data structure in output), we wrote the new tool in this latter language (whose performance is comparable with the C Programming Language). With this new implementation we were able to process the whole library (40000 documents) in one night.

Still the whole process was a bit involved, so we decided to drop the intermediate phase of creation of the RDF files, immediately storing metadata in the data base (thus avoiding the explicit creation of backward pointers, which in any case can be easily reconstructed from the DB). At the same time we noticed that the mere parsing of the source document was a quite costly operation, due to the dimension of the input XML-files, while the kind of metadata we were interested in could be easily located even without a complete parsing.

As a last resort (crying shame on XML specifications and technology), we wrote the final tool in . . . flex! A simple lexical analysis of the source, together with a simple use of counters for skipping well-balanced portions of the input, and a symbol table for managing identifiers, is indeed enough to locate and extract the metadata we are interested in.

With this latter tool, we are able to process the whole library in a few hours; moreover, the bottle-neck is not any-more the extraction, as it used to be, but the process of populating the DB, i.e. the actual insertion of the tuples in the tables.

As a by-product of this experience we are currently investigating the possibility to develop a sort of "XML-grep" tool directly working on sax-event.

References

[1] Mathematical Markup Language (MathML) Version 2.0, W3C Reccomendation 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] Asperti, A., Goguadze, G., Melis, E., "Structure and Meta-Structure of Mathematical Documents", Deliverable n.D1b of Project IST-2001-33562 MoWGLI, `http://www.mowgli.cs.unibo.it/`.

[5] Goguadze, G., "Metadata for Mathematical Libraries", Deliverable n.D3a of Project IST-2001-33562 MoWGLI, `http://www.mowgli.cs.unibo.it/`.

[6] Ferruccio 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.

[7] Guidi, F., Sacerdoti Coen, C., "Querying Distributed Digital Libraries of Mathematics" In Calculemus 2003, Aracne Editrice S.R.L., Therese Hardin e Renaud Rioboo editors, ISBN 88-7999-545-6, pp. 43–57.

[8] Guidi, F., Schena, I., "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.