Indexing and Retrieval in a Heterogeneous Formal Library

Indexing and Retrieval in a Heterogeneous Formal Library

Claudio Sacerdoti Coen, Abdelghani Alidra

Indexing and Retrieval in a Heterogeneous Formal Library

Claudio Sacerdoti Coen[1] <claudio.sacerdoticoen@unibo.it> and
Abdelghani Alidra[2] <abdelghani.alidra@inria.fr>

[1] Department of Computer Science and Engineering - University of Bologna
[2] INRIA laboratory, Deducteam team, Laboratoire Methodes Formelles, ENS Paris-Saclay

Abstract. Dedukti and MMT are both examples of tools that can collect mathematical libraries coming from different systems in a unified but heterogeneous body. Indeed, both tools implement a logical framework where the logics or type systems of the various mathematical assistants can be represented, so that their libraries can be encoded in a single format preserving well-typedness. Still, the images of the various libraries remain disjoint: logical consistency of their union is not granted, the encoding of the various statements differs from logic to logic, and the same mathematical entity remains defined repeatedly and independently. To benefit from the common representation, then, additional tools need to be developed, for example, to translate results encoded in the representation of a logic into the representation of a stronger logic, taking care of aligning the duplicated mathematical entities.
In this paper we address the challenges posed by indexing and searching the large, heterogeneous library. For example, users may expect to find results in the library up to the encoding used and up to alignments, so to be able to mix in the search result statements originally coming from different systems and logics. In particular, we describe new indexing and retrieval capabilities that we integrated directly into the LambdaPi proof assistant, that can work on Dedukti files.

1 Logical Frameworks, Interoperability and Searching

Interoperability between different mathematical systems remains a challenging goal despite the amount of research invested into it in the past years. One old approach to it, that has been recently revived by at least the Deducteam[3] and the MMT teams[4], consists in adopting a single logical framework, express every logic/type theory into it and export the libraries of mathematical results, developed in the various systems, into the common logical framework, obtaining a huge, heterogeneous mathematical library.

The encoding of logics and type theories is designed so that well-typed terms that are in the image of the encodings guarantee that the original expression

[3] https://deducteam.gitlabpages.inria.fr/
[4] https://uniformal.github.io/

or proof in the source logic was correct/well-typed. This result does not yield interoperability yet: in general it is not possible to combine terms coming from the encoding of two different logics/systems preserving logic consistency, unless one logic is a strict superset of the other and the encodings also agree. This is the main motivation to name the library as *heterogeneous*. Moreover, even if the library would be consistent as a whole, the same mathematical object occurs multiple times in the library, once for every system it originates from, and nothing identifies the multiple occurrences. Therefore, in order to exploit the library and turn the logical framework into a bridge for interoperability between two systems, one still needs to translate results in the encoding of one logic into results in the encoding of another logic, also taking care of aligning mathematical concepts that can be identified. Multiple approaches to this translation exist: sometimes one can translate a whole logic/type-system into a stronger one in a uniform and complete way (e.g. to import into Lean theorems from HOL, whose logic is weaker); other times one can just do a best-effort, partial translation, where some proofs in a stronger logic may be recognized to hold also in a weaker logic (see, e.g. the work of F. Thiré [1] that detected in the library of Matita those proofs that could be rephrased in the weaker Simple Type Theory with Prenex Polymorphism); yet another possibility is to use the intermediate steps of the proof in one logic just as a proof sketch to informally guide proof search in the other system, possibly as an auto-formalization task.

Whatever approach to interoperability is chosen, passing through the common logical framework helps in simplifying some tasks, factoring out some logical intricacies and simplifying the engineering of the translation, for example by disentangling the translation code from the code that needs to extract the mathematical content from the different mathematical systems to make it system independent. At the same time, we have to say that currently the process ignores all the extra-logical, domain specific information that is associated to mathematical objects in the mathematical systems in order to make them usable, like mathematical notation, implicit arguments, all the type-classes/unification hints/canonical structures machinery to automatically capture the user provided automation that allows to make sense of mathematical statements, and the ad-hoc information to allow some tactics to reason on the proof objects. Future research will clarify whether it will be possible to also encode uniformly all this extra-logical information and if actual interoperability between systems thanks to the logical framework approach will succeed or not.

In case of success, one ingredient that is obviously necessary is the ability to index the heterogeneous library and retrieve mathematical statements from it. Indeed, suppose that a user or an automated tool (like a tactic) wants to know if some result has already been proved in another system — and therefore it occurs in the heterogeneous library — or if some result in the library can be applied now to progress in the proof. How can it get this information? Of course almost all the source mathematical systems already provide search facilities, but we do not want the user to have to interact with all of them to find the results when those are already in the heterogeneous library. At the same time,

we should not assume that the user has already translated all the results from the heterogeneous library to his own system — so that it can just search in his libraries — because the translation can be so time consuming that it has to be done only on-demand. This is in particular the case when the translation is best effort and requires human intervention. The conclusion is that it is worth implementing search engines able to index directly the heterogeneous library. In this paper we describe such an implementation.

Previous works on indexing and retrieval for mathematical libraries and in particular for the libraries of interactive theorem provers abund [2,3]. Moreover, there is a lot of very recent works on the application of machine learning techniques to searching. Nevertheless, heterogeneous libraries pose new problems and exacerbate old ones. Identifying and addressing them is the main contribution of the current paper.

The already cited Dedukti and MMT are both logical frameworks and tools. As logical frameworks they are both extensions of LF [4]. While in principle LF is sufficiently expressive to encode every other system, in practice encodings in LF require huge terms and are unpractical. In this paper we focus on Dedukti, that extends LF by means of user provided rewriting rules in the spirit of deduction modulo [5]. The choice is not essential: we picked Dedukti because no search engine exists for it (while for MMT related previous work exists [6]) and because of support from the Cost Action CA2011 EuroProofNet where Dedukti has a prominent role as an interchange system.

The plan for the rest of the paper is as follows: in Section 2 we recall the main techniques in use for indexing and retrieval in libraries of formal mathematics; in Section 3 we address the new problems that occur when indexing Dedukti files and how to solve them; in Section 4 we discuss our implementation as an extension of LambdaPi; finally Section 5 is devoted to the conclusions and future works.

2 Searching in Formal Mathematical Libraries

In our previous work [2], we presented an exhaustive study of the systems and techniques available at the time for indexing and retrieval in mathematical libraries, comprising both libraries of formal math developed in interactive provers and collections of documents. Even if a more recent exhaustive survey exists [3], not much has changed in the meantime for formal libraries and our own survey has the benefit of providing clear taxonomies of scenarios and modular implementation techniques to briefly describe how a system works. Thus, we will briefly recall here the taxonomies from our study on mathematical libraries indexing [2] that will be helpful in the next sections. The user can consult the original paper for details and references. It is worth mentioning that we do not consider in this paper more recent techniques from machine learning, leaving those as future work.

2.1 Purpose Driven Taxonomy

Search systems are targeted either to document synthesis, to document retrieval or to formula retrieval. We are interested in the last two.

In *document retrieval* the human looks for excerpts of mathematical documents combining formulae, keywords and free text. Formulae may be relevant just by similarity, retrieval may be slow (a few secs). A good balance between precision and recall is important and the results need to be ranked in terms of both similarity and usefulness to the user. For example, the user may have a precise theorem in mind that she wants to prove; however, even retrieving related lemmas or similar theorems proved for other mathematical structures is useful.

Example 1. A query could combine the keywords "ring","homomorphism" and the pattern "Img $f \equiv R/\ker f$" to locate a page explaining the fundamental theorem of ring homomorphisms. Example of useful results that could also be listed are pages on the fundamental theorem of group homomorphisms or introductions to abstract algebra. □

In *formula retrieval*, a program (e.g. a tactic) looks for statements in the library that match a given query. The match can be either by generalization or by instantiation and both can be expressed using unification metavariables either in the query or in place of parts of statement in the library (e.g. in place of universally quantified variables, the quantifier is dropped). Recall and speed need to be maximized and the result set may be unordered: the program will anyway test the returned results to rule out the false positives and it will decide in which order to try them, for example to try to close a proof goal automatically.

Example 2. If the goal to be proved contains the subexpression $\lim_{x\to\infty}(\sin 1/x + \cos 1/x)$, in order to simplify the goal, a tactic could search for all theorems whose conclusions match $E_1 = E_2$ where either E_1 or E_2 are generalizations of the subexpression. The query could retrieve the lemma $\forall f, g, l. \lim_{x\to l}(f(x) + g(x)) = \lim_{x\to l} f(x) + \lim_{x\to l} g(x)$ by indexing it's conclusion as $\lim_{\to \_}(\_ + \_) = \lim_{\to \_}\_ + \lim_{\to \_}\_$, building the query pattern as $\lim_{x\to\infty}(\sin 1/x + \cos 1/x) = \_$ and retrieving all lemmas whose indexed conclusions unify with the query pattern, where underscores are allowed to unify with any expression. □

2.2 Encoding Based Taxonomy

The indexed documents may encode the *presentation* of formulae (i.e. how they look like), their *content* (that captures imprecisely the informal mathematical meaning), or their *semantics* (i.e. the way they are expressed in a specific logic or type theory). The latter is usually associated to formula retrieval inside an interactive theorem prover, while the others, to document retrieval by humans that may ignore or not be interested in the technicalities of the representation. There exist forgetful mappings from semantics to content and from content to presentation. These mappings can be used during indexing to allow users to use presentational or content queries while still retrieving semantics formulae.

Representations in a logical framework fit in the semantics level, but carve into it yet another deeper level: as we will see later, the same statement in Dedukti may be encoded differently if it is expressed in the encoding of HOL [7,8] or in the encoding of the Calculus of Constructions [9], even when it perfectly fits the intersection of the two. Therefore only the content of the two statements match and not their semantics, even if they are both in the same heterogeneous library. In other words, from the point of view of indexing it is still like if the two statements belonged to two disjoint systems using two different representations.

Example 3. The presentation level formula $\forall m, m + 0 = m$ could be

$$\forall m, \text{eq } (\text{add } \ m \ \text{ zero}) \ m$$

at the content level,

$$!m^{\text{nums.num}}, \\ =[\text{nums.num}](\text{arith.}+ \ m^{\text{nums.num}} \ (\text{nums.NUMERAL } \ \text{nums.\_O})) \ m^{\text{nums.num}}$$

at the semantics level for HOL, in the library of HOL-Light, and

$$\text{theory\_hol.Prf}(\text{theory\_hol.}\forall(\lambda m : \text{theory\_hol.El } \ \text{terms.num}, \\ (\text{theory\_hol.}= \ \text{terms.num } (\text{terms.}+ \ m \ (\text{terms.NUMERAL } \ \text{terms.\_0})) \ m)))$$

at the semantics level for Dedukti where the formula shows the encoding of the HOL-Light formula inside Dedukti. The constants Prf, El, = and $\forall$ are used to encode the logic of HOL-Light into Dedukti. To keep the formula short, HOLLight. theory_hol has been shortened to theory_hol and similarly for terms. □

2.3 Techniques Driven Taxonomy

Search engines are implemented combining a *main technique* from a small pool with an arbitrary number of *modular techniques* mostly meant to trade recall with precision. Remember that in the document retrieval scenario a certain degree of imprecision is a feature and in controlled ways (e.g. match up to logical consequence) the same happens for formula retrieval too.

Main techniques: among the main techniques of interest for us are *structure-based indexing*, e.g. via substitution trees, and reduction to SQL-like queries via *extraction of features* from formulae. The first technique is restricted to formula retrieval up to instantiation/generalization and therefore it rigidly captures the shape of the formula. Therefore precision is maximized, but recall is not and modular techniques have to be employed to augment it. The second technique extracts from the formulae certain structural features that are preserved by instantiation/generalization, typically the fact that some constant occurs in a formula in a certain region (hypothesis or conclusion, under how many quantifiers and if it is the head constant or not). It achieves a good balance between precision and recall. The remaining, not discussed, main techniques are either abandoned or they do not perform well for instantiation/generalization.

Modular techniques: the ones relevant to search in formal libraries and that we have used in this work are the following. *Normalization* rewrites formulae to equivalent normal forms preserving their semantics, improving recall without harming precision. For example $\geq$ can be rewritten to $\leq$ or in classical logic formulae can be put in prenex normal form. *Approximation* rewrites formulae losing information and thus trading precision for recall. Mapping formulae encoded at the semantics level to the content level is an example. *Query reduction* also trades recall for precision by dropping or weakening some constraints in the query. For example, one could enlarge the query to include frequently co-occurring terms or one can only preserve the top-level structure of the query.

All the techniques listed above have in common the fact of being expressed as rewritings of the query (for query reduction), of the terms to be indexed (for approximation) or of both (for normalization).

Example 4. When indexing $\forall n, m, d : \mathbb{N}. n + d > m + d \Rightarrow n > m$ the statement can be first normalized to $\forall n, m, d : \mathbb{N}. m + d < n + d \Rightarrow m < n$ and then approximated to $\forall \_, \_, \_ : \mathbb{N}. V\# + V\# < V\# + V\# \Rightarrow V\# < V\#$ where all variables have been replaced by a placeholder $V\#$ that stands for any variable, effectively forgetting the variables names and the binding structure. Then consider the query that looks for the statement $\forall x, y, z : \mathbb{N}. x + y < x + z \Rightarrow y < z$. By applying the same rewriting rules to the query before matching, the statement is found in the library, even if it is logically equivalent to it only up to commutativity of addition and duality between $<$ and $>$. An aggressive form of query reduction applied to this example would be to drop the constraints on the exact structure of the formula and rewrite the query to become "find a statement where $\mathbb{N}$ occurs anywhere, $+$ and $<$ occur in one hypothesis, $<$ occurs in the conclusion and no other constants occur". □

In the next section we identify the peculiarities of indexing and retrieval in the heterogeneous library of Dedukti.

3 Indexing and Retrieval in Heterogeneous Libraries

3.1 An Introduction to Dedukti

LF, also called $\lambda\Pi$, is a type theory (more precisely a Pure Type System [10]) having just two sorts (`Type` to type types and `Kind` to type `Type`), variables, constants, function application, dependently typed λ-abstractions to declare functions and product types to type them. Functions can only take in input terms (i.e. expression whose type is typed by `Type`) and not types. Therefore, to achieve polymorphism, one is obliged to introduce terms (called *codes*) that represent types, together with a decoding function that maps codes to types. According to Curry-Howard isomorphism [11], one can see propositions as types and encode them accordingly.

The following is a self contained example in LF where we encode the axiom `divides` that states that $\forall n, m, q : \mathbb{N}. n * m = q \Rightarrow n|q$ after having encoded a

fragment of first order logic. We use the Dedukti syntax. In particular `s -> t` and `x:s -> t` are the non-dependent and dependent function space and `x:s => b` is the syntax for λ-abstraction. The keyword `def` allows to add some rewriting rules to the running example.

```
s : Type.               (; the type of codes for types ;)
o : Type.               (; the type of codes for propositions ;)
def El : s -> Type.     (; the decoding function for codes for types ;)
def Prf : o -> Type.    (; the decoding function for codes for propositions ;)
nat : s.                (; the code for natural numbers ;)
func : s -> s -> s.     (; the code for functions given the codes for
                           the domain and codomain ;)
def apply : t1:s -> t2:s -> El (func t1 t2) -> El t1 -> El t2.
implies : o -> o -> o.              (; code for logical implication ;)
forall : t:s -> (El t -> o) -> o.   (; code for universal quantification ;)

mul : El (func nat (func nat nat)).   (; product function over naturals ;)
div : El nat -> El nat -> o.          (; code for divides predicate ;)
eq : t:s -> El t -> El t -> o.        (; code for polymorphic equality ;)
divides :
 Prf
  (forall nat (n: El nat => forall nat (m: El nat =>
    forall nat (q: El nat =>
     implies
      (eq nat (apply nat nat (apply nat (func nat nat) mul n) m) q)
      (div n q))))).
```

It should be immediately evident how the statement of `divides` occurs heavily encoded in LF. In particular, LF forces the so called typed-syntax approach, where every subterm explicitly carries its own type derivation, introducing a lot of syntactic overhead. Moreover, the encoding is not at all unique: the names `s`,`o`,`El`,`Prf`,... are arbitrarily chosen and their types are expressed in the encoding of first order logic. To encode other logics or type systems one needs to type the constants differently and add new ones.

In order to greatly reduce the complexity of the encodings, Dedukti [12] introduces user provided, type preserving rewriting rules. The user must take care of keeping confluence of the rewriting system. For example, the next excerpt shows how to simplify the encoding above rewriting logical implications, functions and universal abstraction to identify all of them with the meta-level LF dependent product. The types of `divides` and `divides2` are identified in Dedukti thanks to rewriting. The `-->` symbol splits the left hand side from the right hand side of a rewriting rule, that must start with the list of variables occurring in the left hand side.

```
[t1, t2] El (func t1 t2) --> El t1 -> El t2.
[p1, p2] Prf (implies p1 p2) --> Prf p1 -> Prf p2.
[t, p] Prf (forall t p) --> x: El t -> Prf (p x).
[t1, t2, f, x] apply t1 t2 f x --> f x.
divides2 :
 n: El nat -> m: El nat -> q: El nat ->
  Prf (eq nat (mul n m) q) -> Prf (div n q).
```

Note that the rewriting rule for `apply` is well typed thanks to the rewriting rule for `func`. Note also that, while the statement of `divides2` now looks simpler, details of the encoding are still present as occurrences of `El` and `Prf`. If we had encoded a more complex type theory, like the ones of Rocq [13] or Lean [14], the details still

visible would have been more prominent. For example, `Elem` would have taken an additional parameter to specify via codes in which universe the type lives.

3.2 Issues when Searching in the Library of Dedukti

From Section 3.1 the main difficulties of searching the heterogeneous library of Dedukti should be immediately clear.

Problem 1: search up to encoding. Imagine that the user is looking for a proof of $\forall n, m, q : \mathbb{N}. n * m = q \Rightarrow n|q$. Of course the lemma could occur in the library multiple times, each one using a different encoding according to the logic/type-system of the mathematical system it comes from. We want to spare the user from the need to know every encoding and to issue the query multiple times, one for each encoding.
Solution: we solve the problem via approximation, seeing the query above as a query at the content level and the encoding in Dedukti as the semantics level. Therefore we assume, for each encoding, the existence of a system of rewriting rules to "throw away" the details of the encoding. For our example, in order to recall `divides2` it is sufficient to add the rules:

```
[t] El t ---> t.
[p] Prf p ---> p.
```

Note, that the two rules above, even if written in Dedukti syntax, are not type preserving in the Dedukti logical framework. In Section 4 we will discuss how to implement them in a tool. For the time being, just think of these rules to be in some Dedukti-resembling syntax of a new content level, untyped language.

Problem 2: search up to rewriting rules. Thanks to the rewriting rules introduced in the encoding, the statement of the lemma we are looking for could be present in the library as the one of `divides`, as the one of `divides2` or as another shape obtained reducing the one of `divides`, but stopping before the normal form (the statement of `divides2`) is reached. For example, Krajono, a tool to encode the library of Matita into Dedukti, strangely reduces the occurrences of the equivalent of `implies`, but not when they occur in the source of another `implies`, yielding strange looking statements of the sort `Proof (implies p q) -> Proof p -> Proof q.`
Solution: to augment recall, we can apply normalization, rewriting both the query and the formulae to be indexed also according to all the rewriting rules used in the encoding. Of course, since we are mixing these rules with the ones for approximation, we need to complete the rewriting system in case of critical pairs, in order to obtain a confluent system.

Example 5. The rules to undo the encoding combined with the ones given in Section 3.1 rewrite the Dedukti code of Example 3 to the formula

```
m : terms.num ->
  theory_hol.= terms.num (terms.+ m (terms.NUMERAL terms._0)) m))
```

that closely resembles the semantics level formula for HOL-Light. The formula is still far away from the content level: for example it exposes the base-2 encoding of naturals in HOL-Light using finite sequences of bits prefixed by NUMERAL. □

Problem 3: search up to alignments. Imagine a user of Lean (or a Lean tactic) that is looking for a theorem in the heterogeneous library to progress in a proof in Lean. The mathematical concepts involved in the Lean statement could exist in the library, but they are likely to at least be named differently, and possibly have more differences (e.g. in the order of the parameters, in how general is the mathematical structure they are defined for, in their arity, etc.). This issue is well known and all the proposals for interoperability boils down to detecting *alignments* [15]. We would like the query to match up to alignments.
Solution: by expressing alignments as rewriting rules towards a canonical form, we can again solve the issue by normalization or, when the alignments do not preserve types, by approximation. However, a problem remains of obtaining large sets of alignments, a problem that could benefit from recent advances in the field of machine learning [16]. To supplement alignments, a first useful technique is also query reduction: since it is likely that two constants that have the same name (but not the same qualified identifier) in two different parts of the heterogeneous library could be alignable, the query shall be issued multiple times resolving every time the combination of the names in a different way. For example, the identifier `nat` in the user-provided query could be replaced once by `matita.arithmetics.nat.nat` (the qualified identifier for natural numbers in the library of Matita) and another time by the corresponding qualified identifier for natural numbers in the library of another system. To implement this form of query reduction, we assume to have an index that, given a name, returns all the fully qualified identifiers in the heterogeneous library that end in the very same name (possibly up-to case insensitivity). As a minor note, observe that the combination of query reduction with the rewriting of queries imposed by normalization may yield to perfect duplicates of the query after normalization. For example, a name that occurs in a part of the encoding to be dropped can be resolved in two different ways. It is therefore important to get rid of duplicates before actually running the final shape of the queries.

Example 6. One could recover the content level formula of Example 3 from the one of Example 5 by aligning the semantics level constants (e.g. terms.+) to content level ones (plain + to continue the example) and by mapping the base-2 encoding of HOL-Light to Peano numbers. In this approximation process a lot of details of the representation in the HOL logic are lost. Therefore this makes sense primarily when the HOL-Light library is searched by a user of a different system. □

While the new problems identified in dealing with the heterogeneous library are significant, the good news is that they can all be solved according to well

known techniques and that, with the only exception of query reduction, what is needed is "just" the possibility to let the user specify sets of rewriting rules to be employed. On the other hand, the complexity of the rewriting engine must at least allow the rules already used in the encodings. This strongly suggests that the most efficient and compact way to implement the indexer and search engine is to reuse (parts of) an implementation of the Dedukti logical framework. For this reason we decided to integrate a prototype directly into the code of LambdaPi [17], an interactive prover for Dedukti which is currently the most supported implementation of the Dedukti logical framework.

4 Implementing Search Functionalities in LambdaPi

4.1 Changes to LambdaPi

Dedukti, the logical framework, comes with a set of tools to typecheck, generate or manipulate Dedukti libraries. In particular Dedukti is also the name of the first type-checker for the logical framework. While the tool can still be used and alternative implementations of it exists (e.g. Kontroli, written in Rust [18]), the main development group is now focused on the LambdaPi interactive prover, that is partially compatible with Dedukti sources. LambdaPi is written in the OCaml language and the code includes an optimized rewriting engine that is invoked during type checking. Moreover, the system also checks that every rewriting rule preserves types, which implies that rewriting rules must refer only to constants previously defined in the environment. LambdaPi does not check confluence of the rewriting rules, even if it has options to feed the rules to external tools for confluence checking. Being an interactive prover, LambdaPi also implements metavariables that stand for missing or implicit terms and that can be instantiated via unification. We have exploited metavariables for representing holes in the queries or in the statements to be indexed to obtain indexing via instantiation/generalization, an added benefit of implementing the search engine reusing the code of a prover.

In order to implement normalization and approximation, we have first modified the code of LambdaPi in several ways:

1. It is now possible to disable the check for preservation of types when reading special LambdaPi files made only of rewriting rules; nevertheless, the rules still need to be *scoped*, meaning that the identifiers written in the rules need to be resolved. Note that in normal mode scoping and type-checking are also responsible for inserting implicit parameters and for filling them according to typing constraints. This does not happen in our special mode, creating issues with non-linear rewriting rules. Indeed a non-linear rewriting rule fires only if some arguments occur to be identical, which may not be the case when some implicit argument should have been instantiated by typing. For the time being, the user is responsible to turn non-linear rules into linear rules taking care that confluence is not broken.

2. The special files of the point above list all the rewriting rules that will be applied by normalization to the statement of the library when indexing them. In order to have a single place where to list all such rules and in order to be able to index library files while processing them, the rules in the special files need to refer to constants that have not been declared yet. This required several changes to the code of LambdaPi. First of all the code for scoping rules in special model had to be changed to only allow fully qualified identifiers and to avoid checking if those where already defined. The second change was about where the rules themselves where stored in compiled form. Indeed, to quickly apply rewriting rules, LambdaPi compiles them to decision diagrams, one for each head symbol in the left-hand-side of a set of rules, and stores the decision diagram in the data structure for the symbol itself. To process rewriting rules before the symbols are introduced, the decision diagrams are now detached from the data structure for symbols and the association occurs lazily at run-time.

4.2 Main Implementation Technique and the Query Language

Having already discussed what modular techniques are required to support an heterogeneous library, we are left with the choice of the main technique for indexing and retrieval. While we are fond of reduction to SQL via feature extraction [19], structure-based indexing works perfectly well in some scenarios where extreme precision is preferred to high recall, in particular in formula retrieval employed inside automatic tactics. We have therefore decided to implement both at once, also introducing a *query language* to combine atomic queries to obtain complex ones, which is mandatory when working with features, since an atomic query dealing with just one feature is typically never sufficient.

We start the discussion by presenting the syntax and semantics of the query language:

```
Q ::= B | Q,Q | Q;Q | Q|PATH
B ::= WHERE HOW GENERALIZE? PATTERN
PATH ::= << string >>
WHERE ::= name | anywhere | rule | lhs | rhs | type | concl | hyp | spine
HOW ::= > | = | >=
GENERALIZE ::= generalize
PATTERN ::= << term possibly containing placeholders _ (for terms)
               and V# (for variable occurrences) >>
```

A query `Q` returns a set of results. Each result is a tuple made of the qualified identifier of the constant or rewrite rule found in the library, the coordinates where its declaration is to be found on the hard disk (in the form filename, initial position and final position) and a list of motivations that explain what are the reasons that determined the inclusion of the result in the set.

The syntax for queries is reminiscent of Prolog: `Q,Q` is logical conjunction and semantically it selects the intersection of the results of the two queries, appending the lists of motivations; `Q;Q` is logical disjunction and semantically it

selects the union of the results, also appending the list of motivations when a result satisfies both queries at once; `B` is a base query that comes with results having their own motivation; finally `Q|PATH` restricts the set of results to those whose qualified identifier is a suffix of the given `PATH`. The latter construct can be used for example to restrict the results to those in the part of the library of Rocq that deals with real numbers.

A base query `B` selects those constants that match a given `PATTERN`. The interpretation of the pattern and the kind of match is specified by the `WHERE`, `HOW` and `GENERALIZE` parts of the query. The simplest case is when `WHERE` is `name`. In that case the `PATTERN` can only be a non qualified identifier, `HOW` must be equality and the constant matches if its non qualified name matches exactly the pattern. In all the other cases the `PATTERN` is a LambdaPi term, possibly containing placeholders to be instantiated or the special symbol `V#` that matches any variable occurrence. When placeholders are used, the query is thus resolved by instantiation. `WHERE` specifies where the pattern must be present in the term: `rule`, `lhs` and `rhs` are used to retrieve rewriting rules and the pattern must occur respectively wherever, in the left hand side or in the right hand side of the rule; `type`, `concl`, `hyp`, `spine` are used to retrieve constants and the pattern must match respectively wherever in the constant type, in its conclusion or in one hypothesis or as a suffix of its spine. To make sense of the previous terminology, consider a typical statement of the form $\forall x : T, H_1 \Rightarrow \ldots \Rightarrow H_n \Rightarrow C$ where C is not a product type — remember that universal quantification and logical implication are both product types in dependent type theory and thus in LF and Dedukti. Then each T and H_i are hypotheses, C is the conclusion and $H_j \Rightarrow \ldots H_n \Rightarrow C$ is a suffix of the spine (as well as the whole statement when zero or more initial universal quantification have been dropped). The `HOW` part of the query completes the description of the matching constraints: when `HOW` is equality the pattern must be found exactly in a position specified by `WHERE`; when it is `>` it must match a strict subterm of the position specified by `WHERE`; finally `>=` allows both modes at once.

The results of running a base query `B` have as motivations triples made of the `WHERE`, `HOW` and `GENERALIZE` parts of the query. Ignoring `GENERALIZE` for now, a singleton result could then tell something like: "`divides` is the result because `(eq _ _)` is exactly an hypothesis, `mul` occurs as a strict subterm of an hypothesis, `nat` is also exactly an hypothesis and the conclusion matches exactly `V#|V#`, i.e. a variable divides another one". The link with reduction to SQL is now evident: the features used in Asperti et al. [19] where of the form "the constant X occurs exactly/inside in position Y under N binders" that corresponds to the degenerate case where all patterns are identifiers. Since only identifiers were used as patterns, there was no need to employ substitution trees to match subterms and all that was necessary was to join together queries over tables of records `IDENTIFIER-WHERE-BINDERS` — a perfect job for a relational database. Note that for the time being we are not recording the number of binders information that was used only in some kind of queries and that is not very stable under arbitrary rewritings.

Finally, when `GENERALIZE` is added to a base query, the pattern is matched against the conclusions of the statements in the library up to generalization, i.e. interpreting the variables bound in the spine of the constant type as metavariables. For example, the query `div 2 4` can match the constant `divides` since its conclusion `div m n` is turned into `div _ _`. This makes queries by generalization useful to look for lemmas to progress in the proof (e.g. by back-chaining as in the case of `divides`).

4.3 Implementation Details

To index the heterogeneous library of LambdaPi we proceed as follows, filling in two distinct indexes, a hash-table over strings (to resolve the queries having `WHERE=name`) and a discrimination tree (for all the other queries):

- when processing a file in indexing mode, each constant and rewrite rules is indexed
- the association name/identifier (and location) is added to the hash-table (for constants only)
- the type of the constant or the rewriting rule is normalized according to the user provided rewriting rules that implement normalization and approximation
- the set of subterms of the constant type or of the rewriting rule is determined, remembering for every subterm its location (e.g. as a strict subterm of an hypothesis)
- the association subterm/identifier, location and position of the subterm is added to the discrimination tree with `GENERALIZE=false`
- the previous step is repeated after replacing in the subterm the variables bound by the spine with metavariables, setting `GENERALIZE=true`

Roughly speaking, the discrimination tree can be thought as a trie over strings that describe pre-visits of the abstract syntax trees of formulae; therefore running a basic query where `WHERE≠name` just consists in applying the user provided rewriting rules for normalization to the query, pre-visit it to obtain a sort of string and perform a lookup in the discrimination tree in time linear in the length of the previsit of the string. Resolving a basic query is thus pretty fast. To combine basic queries we just apply unions and intersections of results, that can be done in time $O(m \log(n/m + 1))$ where m and n are the cardinalities of the two sets. Path restriction is an $O(n)$ operation where n is the cardinality, assuming paths to have constant bounded size and thus suffix check to be $O(1)$. Finally, the queries are augmented before normalization to resolve the unqualified names occurring in the queries. To perform the augmentation, additional simple queries to resolve names are automatically issued.

4.4 User Interface

To index the library, we have augmented the command line interface of LambdaPi with a new `lambdapi index` command that takes as input the path of the

files that hold the rewriting rules used in normalization and approximation and the list of files to be indexed, together with the optional location of the index file to create. A variant of the command allows one to incrementally add more files to a previous index. We currently lack the ability to selectively remove files from the index.

To run a query, we have augmented LambdaPi in three different ways. The command line interface now accepts the `lambdapi search` sub-command to specify a query to run against a given index. The list of files that contain the rewriting rules used to normalize the query must also be provided. The second way to run a query is through a new command accepted by LambdaPi during interactive use. Finally, we added the new sub-command `lambdapi websearch` that turns LambdaPi into a web server that accepts queries and shows the results in a Google-style fashion. The command requires the usual set of files holding the rewriting rules to be applied to the query. All three modes present the set of results to the user by recovering from the hard drive each constant or rewriting rule declaration in LambdaPi syntax.

Fig. 1 shows a screenshot of the search engine in Web server mode.

4.5 Examples of queries

We show here some examples of queries to better illustrate the query language.

Example 7. The user is interested into theorems about rearranging elements of a series of real numbers. Since there are many, under different hypotheses, and since they can be formalized in many ways, a first idea is to look for all constants that have as a premise, functions from naturals to naturals (to permute the indexes) and where the type of real numbers also occurs in an hypothesis: `hyp = (num → num), hyp. >= R`. Fig. 1 shows 22 results, and most of those are relevant. □

Example 8. Continuing the previous example, one result found is

```
symbol thm_SUM_PERMUTE_NUMSEG : Prf (∀ (λ f : El (fun num Real), (∀ (λ p : El (fun num num),
      (∀ (λ m : El num, (∀ (λ n : El num, (⇒ ((@permutes num) p (... m n)) (= ((@sum num) (
     ... m n) f) ((@sum num) (... m n) ((@o num num Real) f p))))))))))))
```

To understand the statement, the user needs first to understand the predicates `permutes` and `...`. In order to find the definition of `permutes` in the library, the best query is `concl = (= (permutes _) _)` that looks for a constant whose statement concludes that the predicate `permutes`, instantiated on a type (the `_` argument), is equal (i.e. logically equivalent) to something (the second `_`). Indeed, the query returns a single result that shows that in HOL-Light, the binary predicate `permutes A s p` requires `p` to permute the elements of the set `s` (a characteristic function over the type `A`), leaving all the other elements of `A` fixed.

```
symbol thm_permutes [A] : Prf (∀ (λ s : El (fun A bool), (∀ (λ p : El (fun A A), (= ((
     @permutes A) p s) (∧ (∀ (λ x : El A, (⇒ (¬ ((@IN A) x s)) (= (p x) x)))) (∀ (λ y :
     El A, ((@∃₁ A) (λ x : El A, (= (p x) y))))))))))
```

For `...` the query `concl = (= ... _)` returns no result. We then try `concl = (= (... _ _) _)` since `...` is a binary predicate, getting six results. We can refine the query to

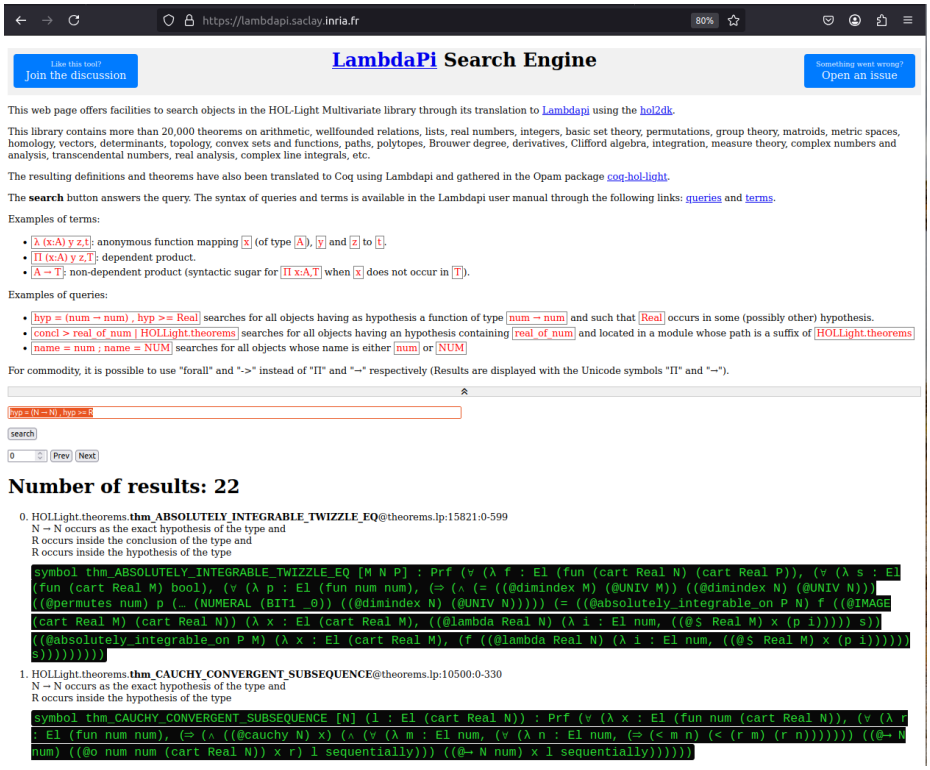

Fig. 1. The search engine in action. The query is displayed just before the results. It looks for all statements having an hypothesis that is the type of functions from natural numbers to natural numbers and another hypothesis containing an occurrence of the real numbers. The query has been written by a Rocq user since `N` and `R` are the names for $\mathbb{N}$ and $\mathbb{R}$ in Rocq. Note that, thanks to the rewriting rules for alignments, the query returns results from the encoding of the HOL-Light [20] library where `N` → `N` occurs as `(fun num num)` and `R` as `Real`. Only the first two results out of 22 are visible in the screenshot.

`concl = (= (... V# V#) _)` to force the arguments to be variables obtaining just three results, the latter one showing that `... m n` is the characteristic function of the interval $[m, n]$ of natural numbers. □

Example 9. Suppose that the user is interested in locating the mean value theorem in the library. Since there exists at least two versions of the theorem with different hypothesis, she can use a disjuntive query:

```
hyp >= absolutely_continuous_on ; (hyp >= continuous_on, hyp >= differentiable)
```
□

Example 10. Disjunctive queries can also be used when normalization is not strong enough to achieve the expected recall level. For example, the following query can be used to retrieve theorems that allow to simplify sums over sets: the sum can occur either on the left hand side or the right hand side of the equality.

```
hyp >= absolutely_continuous_on ; (hyp >= continuous_on, hyp >= differentiable)
```
□

5 Conclusions and Future Works

Thanks to the Cost Action CA2011 EuroProofNet there is a renewed interest in the use of logical frameworks to foster interoperability of mathematical systems. A logical framework allows to collect the encoding of all the results developed in multiple systems in a huge, heterogeneous library. Indexing and retrieval of formulae in such a library raises new challenges, in particular, when user defined rewriting is incorporated in the logical framework to simplify the encodings. Luckily enough, we can reuse ideas from the literature to solve these challenges. Moreover, the solutions are often based on forms of rewriting and therefore the code of the tools that implement the logical framework can be reused for the implementation of the search engine.

In this paper we framed the various problems, proposed solutions and briefly described an implementation for the LambdaPi tool that implements the Dedukti logical framework. Scalability seems to be a minor concern so far: we deployed the search engine on the library exported from HOL-Light (more than 20,000 theorems) with the following result: by using 355 rewrite rules to "undo" the encoding and apply alignments towards the standard library of Rocq, indexing the whole library on a machine with 32 cores i9-13950HX processor and 62GB of RAM, takes 18.19 seconds. It generates the two indexes that, once saved on disk, take only 108MB. The rewriting rules for alignments were automatically generated from hand-made files used to translate HOL-Light proofs into Rocq via LambdaPi [21]. The search engine is available online at the address `https://lambdapi.saclay.inria.fr/` and this particular deployment is an experiment targeted to Rocq users that want to explore the HOL-Light library before importing theorems.

We distinguish between short and long term future works. In the short term, we plan to improve the implementation by letting the user provide also the sources of the libraries translated to other systems/logic. When answering a query, the tool will render each result in the original and in its translated forms. Thus, for example, a Rocq user that search for HOL-Light theorems will also see

the Rocq code of the result. A second short term improvement will be to augment the query language with constructs suggested by the users. At the moment we already received the request for a new filter that allows to use regular expressions over the fully qualified name of each search result, since the name of technical lemmas typically follow a schema that can be exploited by the user.

In the long term, a couple of future works stand out. The first is to collect feedback from users or testers to determine how usable is the current query language. In our previous work [19] on searching the library of Rocq, we hided the query language providing only a limited set of five high-level queries that were deemed useful. The useful queries in an heterogeneous setting, e.g. when a Rocq user is looking for results in the HOL library, are yet to be determined. The second future work is to integrate a way to rank the results. Ranking by similarity between the query and the results, a metric that is often used in formula retrieval, is challenging because normalization, approximation and query reduction interfere: the actual pattern that is used may be quite different in shape from the one entered by the user. The motivations in the query results are already meant to help ranking.

Last but not least, especially in the Lean world, we are observing several new search services that exploit machine learning. The potential of such technology in the context of heterogeneous libraries and proof interoperability deserves some investigation. For example, potentially the phase of turning alignments into rewriting rules could be completely skipped if search happens in a latent space used to automatically translate from one system to another.

Acknowledgements Sacerdoti Coen has been supported by the INdAM-GNCS project "Modelli Composizionali per l'Analisi di Sistemi Reversibili Distribuiti" and both authors by the Cost Action CA2011 EuroProofNet.

References

1. F. Thiré, *Interoperability between proof systems using the logical framework Dedukti. (Interopérabilité entre systèmes de preuve en utilisant le cadre logique Dedukti)*. PhD thesis, École normale supérieure Paris-Saclay, Cachan, France, 2020.
2. F. Guidi and C. S. Coen, "A survey on retrieval of mathematical knowledge," *Math. Comput. Sci.*, vol. 10, no. 4, pp. 409–427, 2016.
3. P. Dadure, P. Pakray, and S. Bandyopadhyay, "Mathematical information retrieval: A review," *ACM Comput. Surv.*, vol. 57, no. 3, pp. 61:1–61:34, 2025.
4. R. Harper, F. Honsell, and G. D. Plotkin, "A framework for defining logics," in *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*, pp. 194–204, IEEE Computer Society, 1987.
5. G. Dowek, "Deduction modulo theory," *CoRR*, vol. abs/1501.06523, 2015.
6. C. Prodescu and M. Kohlhase, "Mathwebsearch 0.5 an open formula search engine," in *Report of the symposium "Lernen, Wissen, Adaptivität 2011" of the GI special interest groups KDML, IR and WM, LWA 2011, Magdeburg, 28.-30.September 2011* (M. Spiliopoulou, A. Nürnberger, and R. Schult, eds.), pp. 257–264, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2011.

7. A. Church, "A formulation of the simple theory of types," *The journal of symbolic logic*, vol. 5, no. 2, pp. 56–68, 1940.
8. J. Hurd, "The opentheory standard theory library," in *NASA Formal Methods Symposium*, pp. 177–191, Springer, 2011.
9. T. Coquand and G. Huet, *The calculus of constructions*. PhD thesis, INRIA, 1986.
10. H. P. Barendregt, *Lambda calculi with types*, p. 117–309. USA: Oxford University Press, Inc., 1993.
11. H. B. Curry, J. R. Hindley, and J. P. Seldin, *To HB Curry: essays on combinatory logic, lambda calculus, and formalism*. Academic Press, 1980.
12. A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard, "Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory," *CoRR*, vol. abs/2311.07185, 2023.
13. ROCQ Prover, "Rocq prover," 2023. Accessed: 2025-07-06.
14. L. de Moura, "Soonho kong, jeremy avigad, floris van doorn, and jakob von raumer," *The Lean Theorem Prover (system description)*, pp. 378–388, 2014.
15. D. Müller, T. Gauthier, C. Kaliszyk, M. Kohlhase, and F. Rabe, "Classification of alignments between concepts of formal mathematical systems," in *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, vol. 10383 of *Lecture Notes in Computer Science*, pp. 83–98, Springer, 2017.
16. L. Horowitz and V. de Paiva, "Mathgloss: Building mathematical glossaries from text," *CoRR*, vol. abs/2311.12649, 2023.
17. G. Hondet and F. Blanqui, "The new rewriting engine of dedukti (system description)," in *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)* (Z. M. Ariola, ed.), vol. 167 of *LIPIcs*, pp. 35:1–35:16, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
18. M. Färber, "Safe, fast, concurrent proof checking for the lambda-pi calculus modulo rewriting," in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, (New York, NY, USA), p. 225–238, 2022.
19. A. Asperti, F. Guidi, C. S. Coen, E. Tassi, and S. Zacchiroli, "A content based mathematical search engine: Whelp," in *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, vol. 3839 of *Lecture Notes in Computer Science*, pp. 17–32, Springer, 2004.
20. J. Harrison, "Hol light: A tutorial introduction," in *International Conference on Formal Methods in Computer-Aided Design*, pp. 265–269, Springer, 1996.
21. F. Blanqui, "Translating HOL-Light proofs to Coq," in *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius, May 26-31, 2024*, vol. 100 of *EPiC Series in Computing*, pp. 1–18, EasyChair, 2024.