Knowledge Archives in Theorema: A Logic-Internal Approach

Camelia Rosenkranz, Bruno Buchberger and Tudor Jebelean

Abstract. Archives are implemented as an extension of *Theorema* for representing mathematical repositories in a natural way. An archive can be conceived as one large formula in a language consisting of higher-order predicate logic together with a few constructs for structuring knowledge: attaching labels to subhierarchies, disambiguating symbols by the use of namespaces, importing symbols from other namespaces and specifying the domains of categories and functors as namespaces with variable operations.

All these constructs are logic-internal in the sense that they have a natural translation to higher-order logic so that certain aspects of Mathematical Knowledge Management can be realized in the object logic itself. There are a variety of operations on archives, though in this paper we can only sketch a few of them: knowledge retrieval and theory exploration, merging and splitting, insertion and translation to predicate logic.

Mathematics Subject Classification (2000). Primary 03E99; Secondary 68T15.

Keywords. Mathematical knowledge management, knowledge representation, knowledge organization.

1. Introduction

A major goal of Mathematical Knowledge Management (MKM) is the storage of "mathematical knowledge" in a suitable way:

- In our understanding, a rigorous formalization (typically in a first-order logic with set theory or in a higher-order logic) is indispensable.
- The knowledge should be reasonably easy to access for retrieval, proving, simplifing, solving and various other operations.

This work was completed with the support of Project SFB 1302 of the FWF (Österreichischer Fonds zur Förderung der wisssenschaftlichen Forschung = Austrian National Science Foundation) and the PhD Fellowship of the Land Oberösterreich (Upper Austria).

By the first of these constraints, the fuzzy notion of "mathematical knowledge" is made precise as a set of well-formed formulae $\Phi \subseteq L(\Sigma)$ over a signature Σ. But this brings us into conflict with the second constraint: For a large knowledge base it is unwise to gather all its symbols into a "flat" signature Σ and all its formulae into a "flat" assumption set Φ. It is of no help either to use lists instead of sets—the real problem is that the hierarchical layers present in informal mathematical practice are discarded.

One way to cope with this problem is to employ logic-external means for supplying the missing structure in the form of metadata [5], semantic annotations [20, 29], the label tools of [25, 24] or content dictionaries within OMDoc [18]. In this paper we would like to point out an alternative that has a very natural flavor: By a slight extension of the object language (the *Theorema* language, a version of predicate logic), this problem can also be remedied by using special logic-internal constructs. Namely, instead of having flat assumption sets, the language of archives provides *labels* for specifying a hierarchical structure on the knowledge base and a hierarchy of *namespaces* for binding the global symbols in an associated structure.

We refer to knowledge bases involving these constructs as *archives*. In fact, they can be viewed as a single formula in a slight extension of the *Theorema* langugage (see Section 2), which is based on the following two design principles:

- Its constructs should be *logic-internal* in the sense that there is a transparent translation to the underlying *Theorema* language.
- As a part of *Theorema*, it emphasizes a natural style similar to that of the working mathematician [13].

The framework presented here should only be understood as a first step in the general direction outlined by these principles.

The benefits of the logic-internal approach become apparent when one considers the logical nature of some "MKM operations". For example, we might say (logic-externally): "Assuming the preliminaries of lattice theory and various definitions (both contained in suitable theory files), the Jordan-Hölder-Dedekind Theorem as in [21, p. 494] holds." Using the logic-internal labels `BasicDefs`, `LatTheory` • `Prelim` and `LatTheory` • `JHDTheorem`, this statement becomes a formula in the object language:

$$\texttt{LatTheory} \bullet \texttt{Prelim} \wedge \texttt{BasicDefs} \Rightarrow \texttt{LatTheory} \bullet \texttt{JHDTheorem}$$

Now "using" this theorem just amounts to modus ponens, once the hypotheses `LatTheory` • `Prelim` and `BasicDefs` are established. (In the logic-external representation, this step can only be expressed as a metalevel inference.) It turns out that logic itself has the means for realizing such labels; one can view them essentially as propositional constants. Likewise namespaces can be given a natural interpretation within the logic, somewhat similar to the domains of categories and functors in the sense of [11].

Let us briefly review some other systems operating on structured knowledge repositories; see also [26]. The largest organized library of mathematics available is

the MML library of Mizar [29]. Other libraries of mathematics include HELM [15] and the mathematical databases formalized in OMDoc [18]. The ongoing efforts for reworking the theory of special functions into an online database in the DLMF project [23] are another instance of structuring a considerable portion of mathematics. Authoring tools like ActiveMath [22] also contain significant portions of structured mathematics. Most of these systems employ textual-based retrieval. A similar mechanism [4], based on HELM, is incorporated into the interactive theorem prover Matita [3]. On a somewhat different line, MathWebSearch [19] allows to search OMDoc documents up to α-conversion.

Another idea, similar to archives in spirit but using a proof-theoretic approach to MKM, is presented in [1, 2]. The library of proofs, theorems and tactics are integrated in the underlying proof logic, where scoping and tactics are represented in a typed λ-calculus. But the point of emphasis there seems to be more on the migration between object and metalevels in mathematics. Similar ideas have also been studied in *Theorema* under the heading of reflection [17].

The *structure of the paper* is as follows. We start by summarizing various MKM-related features of *Theorema* (Section 2). The structure of archives is described in detail within the next two sections. We introduce labels for hierarchical blocks of formulae and various useful notational conventions for already existing *Theorema* constructs (Section 3). Next we discuss namespaces and their relation to the domains of categories and functors (Section 4). Archives can be manipulated by various operations like loading/saving, combining/splitting/inserting, basic instances of retrieval and theory exploration (Section 5).

2. Building and Structuring Mathematics in Theorema

Since the language of archives is developed within the *Theorema* system, let us now describe some of its features that are particularly interesting from the viewpoint of MKM. For a general survey of *Theorema* we refer to [13, 14].

The overall goal of the *Theorema* system is to provide computer support for the working mathematician in his routine tasks, supporting the entire process of mathematical work. Some characteristic features of *Theorema* are:

- It offers a flexible two-dimensional syntax for presenting formulae naturally.
- Proofs can be generated and presented in a style close to that of textbooks.
- Using a variant of higher-order logic, it allows to specify, verify and use algorithms in a unified language.
- Theory exploration is preferred over isolated theorem proving.
- Every part of mathematics can be equipped with efficient specialized reasoners (like CAD for elementary analysis).
- For specifying and manipulating model classes, *Theorema* encourages the usage of so-called categories and functors (see Subsection 2.2 and 2.3).

In the remainder of this section, we explain the logic-external labelling system in *Theorema* (Subsection 2.1), operation objects and how they can be assembled in

a category (Subsection 2.2), and how they can be created by functors (Subsection 2.3).

2.1. Logic-External Labels

Apart from the newer developments described in this paper, *Theorema* offers in its core language various types of (logic-external) labels:

1. Individual formula labels
2. Names for environments like Definition, Theorem, Lemma, ...
3. Names for nested theories

Here is a fragment of a theory of the natural numbers, consisting of an `Axioms` and `Definition` environment, each of which have labels attached to their formulae:

```
Axioms["Peano 2, 4", any[m, n],
    m⁺ ≠ 0                    "p2"
    m⁺ = n⁺ ⇒ m = n           "p4"]

Definition["Addition", any[m, n],
    m + 0 = m                 "a1"
    m + n⁺ = (m + n)⁺         "a2"]

Theory["NaturalNumbers",
    Axioms["Peano 2, 4"]
    Definition["Addition"]
    ...]
```

A theory may also contain—among plain environments as above—other theories, thus building up a hierarchical knowledge base (see the example in Subsection 5.2).

In addition to the above way of specifying labels, *Theorema* also offers a more convenient interface, in which formulae may be input directly into individual *Mathematica* cells and their accompanying (logic-external) labels in the corresponding cell tags. This interface includes also various tools for managing labels as well as simple search facilities, illustrated by the following examples:

4. *Search by formula label*: Look for all formulae in tuple theory with labels ending in `associative` by `LabelLookup["*associative", "BN:Tuples"]`.
5. *Search by formula constants*: Return all formulae containing `is-empty` and $\asymp$ by `SymbolLookup[{is-empty, ≍}, "BN:Tuples", FilterType → All]`).
6. *Applicative higher-order pattern matching*: Search all formulae in `BN:Tuples` using $\asymp$ left of an equality by `FormulaLookup[_ ≍ _ = _, "BN:Tuples"]`.

For more details we refer to [24, 27].

The language of archives is a continuation of the features (1)–(6), with labels now being logic-internal. Moreover, it provides namespaces as a suitable construct for structuring the constants occurring in various branches of a theory hierarchy.

2.2. Operation Objects and Categories

Theories are the building blocks of mathematics and thus also of archives. A theory is determined by its symbols (having a certain arity) and the axioms characterizing them; we call the corresponding model class a *category*. Note that in this paper

the notions of categories and functors are meant in the sense of [11, 8] and not necessarily in the sense of Eilenberg and MacLane (although there will often be a natural correspondence).

We will show later how categories can be defined succintly in the language of archives (see Subsection 4.3). An example of a category in plain *Theorema* is given by the real vector spaces:

$$\texttt{Definition}[\texttt{"RealVectorSpace"}, \texttt{any}[V],$$
$$\texttt{is-vecspace}[V] \Leftrightarrow \underset{\substack{\epsilon \\ V}}{\forall}{}_{[x,y,z]} \underset{\substack{\epsilon \\ \mathbb{R}}}{\forall}{}_{[\lambda,\mu]} \bigwedge \left\{ \begin{array}{c} \left(x \underset{V}{+} y\right) \underset{V}{+} z = x \underset{V}{+} \left(y \underset{V}{+} z\right) \\ x \underset{V}{+} \underset{V}{0} = x \\ x \underset{V}{+} \left(\underset{V}{-} x\right) = \underset{V}{0} \\ x \underset{V}{+} y = y \underset{V}{+} x \\ \lambda \underset{V}{\cdot} \left(x \underset{V}{+} y\right) = \lambda \underset{V}{\cdot} x \underset{V}{+} \lambda \underset{V}{\cdot} y \\ \left(\lambda \underset{\mathbb{R}}{+} \mu\right) \underset{V}{\cdot} x = \lambda \underset{V}{\cdot} x \underset{V}{+} \mu \underset{V}{\cdot} x \\ \left(\lambda \underset{\mathbb{R}}{*} \mu\right) \underset{V}{\cdot} x = \lambda \underset{V}{\cdot} \left(\mu \underset{V}{\cdot} x\right) \\ \underset{\mathbb{R}}{1} \underset{V}{\cdot} x = x \end{array} \right.]$$

Underscripted operation symbols like $\underset{V}{+}$ are a shorthand notation for the corresponding curried versions like $V[+]$. The unary predicate `is-vecspace` is introduced for deciding whether some V is a vector space or not. Observe that such a vector space V is represented as a single object—called an *operation object*—even though it contains the various constituents ϵ, $+$, $-$, $\cdot$, 0. Note also that in *Theorema* ranges can be described by unary predicates, like $\mathbb{R}[\epsilon]$ in the above example; thus $\mathbb{R}[\epsilon][\lambda, \mu]$ actually means $\mathbb{R}[\epsilon][\lambda] \wedge \mathbb{R}[\epsilon][\mu]$.

We have to distinguish between the "whole vector space" V and its "carrier", which can either be defined by a membership predicate ϵ or by a carrier set. A similar distinction can be made for the other operations: They can either be realized as functions/predicate symbols of higher-order logic or as mappings/relations in the sense of set theory. (For simplicity reasons, we will usually refrain from employing set theory in domains.)

In conjunction with categories and functors, operation objects are also called "domains". We will see later that operation objects generalize naturally to the concept of namespaces (see Subsection 4.2). They allow for an elegant formulation of the preservation statements typically encountered in relation with categories and functors. For example, using a suitable definition of direct product (see Subsection 2.3), we have $\texttt{is-group}[G] \wedge \texttt{is-group}[H] \Rightarrow \texttt{is-group}[G \times H]$. Without using operation objects, this statement would look somewhat as follows:

$$\texttt{form-group}[\epsilon 1, +, 0, -] \wedge \texttt{form-group}[\epsilon 2, \oplus, \odot, \ominus] \Rightarrow \texttt{form-group}[$$
$$\texttt{dir-prod-carrier}[\epsilon 1, \epsilon 2], \texttt{dir-prod-binary}[+, \oplus], \texttt{dir-prod-neutral}[0, \odot],$$
$$\texttt{dir-prod-unary}[-, \ominus]]$$

This problem becomes more pronounced when dealing with rings or vector spaces (not to mention that one runs out of symbols for the operations needed).

2.3. Operation Objects and Functors

In a larger archive, one will generally avoid such a proliferation of symbols by using suitable operation objects in a modular way. Consider for example computation in the matrix ring informally denoted by $\langle \mathbb{Z}[x]^{2\times 2}, +, *, \ldots \rangle$.

$$\begin{pmatrix} x^2-7 & 2x+4 \\ x-3 & x^2+3 \end{pmatrix} + \begin{pmatrix} x^2-3 & x-2 \\ x & x^2+4 \end{pmatrix} * \begin{pmatrix} x^3 & x^2+7 \\ 1-2x & 2x^2-3x \end{pmatrix}$$

Operation objects realize the principle of generic implementation and thus avoid code duplication. Without them we need three additions/multiplications: one for the matrix ring $\mathbb{Z}[x]^{2\times 2}$, another for the polynomial ring $\mathbb{Z}[x]$, and a third for the coefficient ring $\mathbb{Z}$. This becomes even worse if we want to consider other coefficient rings like $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Q}(\sqrt{3})$, $\mathbb{C}$, and other constructions instead of the polynomials and matrices, e.g. power series.

Such constructions are typical examples of *functors*—constructing new domains out of given ones. For example, the polynomial functor, written informally as $\texttt{Pol}\colon R \mapsto R[x]$, sends the coefficient ring R to the ring of polynomials over R. Observe that this is already a preservation statement for the functor, claiming that it maps the category `Rng` of rings to itself, informally expressed by $\texttt{Pol}\colon \texttt{Rng} \to \texttt{Rng}$. Such preservation properties are typical for functors F: If the input domain D satisfies a certain property P, its output domain $F[D]$ satisfies some property Q. Viewing the properties as categories, this can also be expressed by saying that the functor F is a map between the categories P and Q. Similar things can be said about functors operating on several domains.

Functors have a computational as well as a proving aspect: While the former amounts to transporting algorithms (e.g. implementing the polynomial arithmetic in terms of the coefficient operations), the latter can be seen as transporting properties (e.g. the property of being a ring in the previous example). The resulting algorithms can of course be implemented in a programming language (e.g. *Mathematica* or *Theorema*), but for verifying their properties one needs a theorem prover; the *Theorema* system is an integrated environment providing both.

Functors can be expressed without extra language constructs, using only the higher-order predicate language of *Theorema*. For functors, a special notation is provided. As an example (mentioned in Subsection 2.2), consider the functor constructing the direct product of groups(though the input need not be restricted to groups—but it usually is to ensure nice "preserved" properties):

$$\begin{array}{l} \texttt{Definition}[\texttt{"DP"}, \texttt{any}[G,H], \\ \quad G \times H = \texttt{Functor}[D, \texttt{any}[x,y], \\ \qquad \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\ \qquad \epsilon[x] \Leftrightarrow \left(\texttt{is-tuple}[x] \wedge (\texttt{card}[x]=2) \wedge \underset{G}{\epsilon}[x_1] \wedge \underset{H}{\epsilon}[x_2]\right) \\ \qquad x \underset{D}{+} y = \left\langle x_1 \underset{G}{+} y_1, x_2 \underset{H}{+} y_2 \right\rangle \end{array}$$

$$\underset{D}{0} = \left\langle \underset{G}{0}, \underset{H}{0} \right\rangle$$
$$\underset{D}{-}x = \left\langle \underset{G}{-}x_1, \underset{H}{-}x_2 \right\rangle]]$$

The expression $\texttt{Functor}[D, \ldots]$ is just an abbreviation for the description quantifier $\underset{D}{\partial}$ yielding the desired operation object ("such a D that all the subsequent definitions are satisfied"). In fact, *Theorema* expands the above formula (usually before starting a proof or a computation) as follows:

$$\begin{aligned}\underset{G,H}{\forall}\underset{x,y}{\forall}\Big(& \underset{G\times H}{\in}[x] \Leftrightarrow \texttt{is-tuple}[x] \wedge \texttt{card}[x] = 2 \wedge \underset{G}{\in}[x_1] \wedge \underset{H}{\in}[x_2] \wedge \\ & x \underset{G\times H}{+} y = \langle x_1 \underset{G}{+} y_1, x_2 \underset{H}{+} y_2\rangle \wedge \\ & \underset{G\times H}{0} = \langle \underset{G}{0}, \underset{H}{0}\rangle \wedge \\ & \underset{G\times H}{-}x = \langle \underset{G}{-}x_1, \underset{H}{-}x_2\rangle \Big)\end{aligned}$$

We will see later how the language of archives supports both formulations of functor definitions (see Subsection 4.2).

3. Syntactic Labels for Hierarchical Knowledge Organization

An archive is a single formula in an extension of the *Theorema* language. Two new symbols are needed, $\rightleftharpoons$ for attaching labels and : for declaring namespaces. We will explain the usage of these two symbols in this and the next section. We start by discussing various notational conventions for already existing *Theorema* language constructs; they help to make large mathematical knowledge bases more readable and less redundant (Subsection 3.1). After this, we will turn to using labels for creating hierarchies of formulae (Subsection 3.2).

The user interface to an archive is a *structured notebook*, a *Mathematica* notebook written with the predefined `TheoremaFormalization` stylesheet. It will contain comments (represented by the cell styles `Author`, `Formalizer`, `Notes`) and nested formula cells (having the cell style $\texttt{Formal}\langle n\rangle$ with n a natural number from 1 to 9). The title of the notebook (having the cell style `Title`) is also considered a formula cell, providing the root label for the underlying label hierarchy. Of course comment cells do not influence the archive created and are meant only as a help for the reader.

Loading an archive from a structured notebook is accomplished by the command `LoadArchive`, saving correspondingly by `SaveArchive`. Another straightforward operation on archives is their translation to plain predicate logic (in the sense of the *Theorema* language). In this paper, we represent the output of this translation in a slightly modified format for ease of readability: If the output formula is a conjunction, we list only its conjuncts.

We view this translation mainly as a convenient way of specifying a semantics for archives. Our ultimate goal is not the translation. On the contrary, we prefer to work directly with the archives for exploiting their organizational annotations in various tasks of MKM, in particular starting a retrieval on an archive, and

expanding an archive by a theory exploration. For more details on these and other operations, see Section 5.

3.1. Blocks and quantifiers

In the language of archives, we can assert a conjunction of several mathematical formulae by using the 'normal' *Theorema* $\wedge$ or by using blocks: A *block* is an "indentation level" in a hierarchy of nested (and hence indented) cells, denoting the conjunction of its parts. Thus a block consists of one or more formulae, possibly in turn containing other blocks. An equivalence or implication with a conjunction on its right-hand side can be broken after the $\Leftrightarrow$ or $\Rightarrow$ with the right-hand side following as an indented block, as in the following example (assuming here the domain D and the relation R to be constants):

$$\begin{aligned}&\texttt{is-equivalence}[D, R] \Leftrightarrow\\ &\quad \texttt{is-reflexive}[D, R]\\ &\quad \texttt{is-symmetric}[D, R]\\ &\quad \texttt{is-transitive}[D, R]\end{aligned}$$

The internal representation of blocks is realized by the ™Conjunction connective. There is no semantic difference between the *Theorema* $\wedge$ symbol and ™Conjunction, but we decided to distinguish between the two for pragmatic reasons. This distinction is somewhat comparable to the sequent calculus, where the point is to distinguish between $\phi_1, \ldots, \phi_n \vdash \phi$ and $\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \phi$, but note that ™Conjunction represents a nested rather than a flat list of formulae. Just as sequents can be exploited for building more efficient provers, we have the hope that distinguishing blocks and conjunctions could be useful in a similar vein.

If formulae are preceded by common quantifiers, their structure can be made more readable by using blocks:

$$\begin{aligned}&\underset{\circ\, D}{\forall\forall}\\ &\quad \texttt{is-idempotent}[D, \circ] \Leftrightarrow \underset{x\in D}{\forall}(x \circ x = x)\\ &\quad \texttt{is-associative}[D, \circ] \Leftrightarrow \underset{x,y,z\in D}{\forall}((x \circ y) \circ z = x \circ (y \circ z))\\ &\quad \texttt{is-commutative}[D, \circ] \Leftrightarrow \underset{x,y\in D}{\forall}(x \circ y = y \circ x)\end{aligned}$$

Existential quantifiers can be used in archives just as universal ones, and one can also combine them into a quantifier prefix before a block of formulae. The following example is taken from projective geometry:

$$\begin{aligned}&\underset{\substack{\texttt{is-point}[p,q]\\ p\neq q}}{\forall}\ \underset{\texttt{is-line}[l]}{\exists}\\ &\quad \texttt{is-incident}[p, l]\\ &\quad \texttt{is-incident}[q, l]\\ &\quad\quad \underset{\texttt{is-line}[m]}{\forall}(\texttt{is-incident}[p, m] \wedge \texttt{is-incident}[q, m] \Rightarrow m = l)\end{aligned}$$

$$\underset{\substack{\texttt{is-line}[l,m]\\ l\neq m}}{\forall}\ \underset{\texttt{is-point}[p]}{\exists}\ \texttt{is-incident}[p,l]\ \texttt{is-incident}[p,m]\ \underset{\texttt{is-point}[q]}{\forall}(\texttt{is-incident}[q,l]\wedge\texttt{is-incident}[q,m]\Rightarrow q=p)$$

Remember that in *Theorema* ranges can be described by unary predicates, like `is-point` and `is-line` in the above example; thus $\texttt{is-point}[p,q]$ actually means $\texttt{is-point}[p]\wedge\texttt{is-point}[q]$.

As for the existential and universal quantifiers, the language of archives provides a multi-line variant of the substitution quantifier $\leftarrowtail$, typically verbalised as "let" in prefix and "where" in postfix usage. It is easiest to first consider an example (claiming correctness of Cardano's formula):

$$\underset{a,b,c}{\forall}\left(x^3+a*x^2+b*x+c=0\right)\ \leftarrowtail\ x=-\frac{p}{3*u}+u-\frac{a}{3}\ \leftarrowtail\ p=b-\frac{a}{3}\quad q=c+\frac{2*a^3-9*a*b}{27}\quad u=\sqrt[3]{-\frac{q}{2}\pm\sqrt{\frac{q^2}{4}+\frac{p^3}{27}}}$$

One can see from this example that substitution quantifiers can be nested into each other. They are used for avoiding multiple occurences of large terms, and/or for ease of readability (e.g. avoiding a large term in an index position).

In *Theorema*, the substitution quantifier can be used for arbitrary expressions, i.e. both for terms and formulae. The general form is

$$\Lambda\ \leftarrowtail\ x_1=\tau_1\quad\ldots\quad x_n=\tau_n$$

and corresponds to the plain *Theorema* expression $\texttt{where}[x_1=\tau_1,\ldots,x_n=\tau_n,\Lambda]$.

3.2. Labelling Blocks of Formulae

It is often useful to group formulae in a hierarchy similar to the chapters and sections of a book (in particular if the formalization comes from a textbook in the first place). In the language of archives, one can achieve this by using *labels*, special formulae associated with blocks of formulae via the $\rightleftharpoons$ symbol. Cell groups having $\rightleftharpoons$ in their first cell are called packages. The first cell in a package is its *head*, the remaining cells make up its *body*. Obviously, a package body is always a block (in the sense explained above).

Here is an example of some labels and their associated formulae taken from one of our formalized notebooks:

$$
\begin{array}{l}
\texttt{Algebra} \rightleftharpoons \\
\quad \texttt{GroupTheory} \rightleftharpoons \\
\quad\quad \texttt{Magmas} \rightleftharpoons \underset{M}{\forall} \\
\quad\quad\quad \texttt{is-magma}[M] \Leftrightarrow \underset{\underset{M}{\epsilon}[x,y]}{\forall} \underset{M}{\epsilon}[x \underset{M}{\circ} y] \\
\quad\quad \texttt{Semigroups} \rightleftharpoons \underset{S}{\forall} \\
\quad\quad\quad \texttt{is-sgroup}[S] \Leftrightarrow \\
\quad\quad\quad\quad \texttt{is-magma}[S] \\
\quad\quad\quad\quad\quad \underset{\underset{S}{\epsilon}[x,y,z]}{\forall} \left(x \underset{S}{\circ} y\right) \underset{S}{\circ} z = x \underset{S}{\circ} \left(y \underset{S}{\circ} z\right) \\
\quad\quad \ldots \\
\quad \texttt{LatticeTheory} \rightleftharpoons \\
\quad\quad \texttt{Posets} \rightleftharpoons \ldots \\
\quad\quad \texttt{Chains} \rightleftharpoons \ldots \\
\quad\quad \ldots
\end{array}
$$

For the moment, one should think of the labels `Algebra`, `GroupTheory`, ... as propositional constants; see below for a more rigorous description. The above package is then translated into plain *Theorema* as follows:

$$
\begin{array}{l}
\texttt{Algebra} \Leftrightarrow \texttt{GroupTheory} \wedge \texttt{LatticeTheory} \\
\texttt{GroupTheory} \Leftrightarrow \texttt{Magmas} \wedge \texttt{Semigroups} \wedge \ldots \\
\texttt{Magmas} \Leftrightarrow \underset{M}{\forall} \left(\texttt{is-magma}[M] \Leftrightarrow \underset{\underset{M}{\epsilon}[x,y]}{\forall} \underset{M}{\epsilon}[x \underset{M}{\circ} y]\right) \\
\texttt{Semigroups} \Leftrightarrow \underset{S}{\forall} \left(\texttt{is-sgroup}[S] \Leftrightarrow \texttt{is-magma}[S] \wedge \underset{\underset{S}{\epsilon}[x,y,z]}{\forall} (x \underset{S}{\circ} y) \underset{S}{\circ} z = x \underset{S}{\circ} (y \underset{S}{\circ} z)\right) \\
\texttt{LatticeTheory} \Leftrightarrow \texttt{Posets} \wedge \texttt{Chains} \wedge \ldots \\
\texttt{Posets} \Leftrightarrow \ldots \\
\texttt{Chains} \Leftrightarrow \ldots
\end{array}
$$

In the example above, it was sufficient to refer to `Semigroups`, `Posets`, ... because it is clear that there are no other packages with these heads. In other cases, a disambiguation is necessary. For example if the user wants to preserve a similar structure in all her packages: She wants to investigate algebra, distinguishing between the basic and advanced theory. For each important notion, she wants to write definitions and theorems under a head with the ad-hoc name `Defs` and `Thms`, respectively. In this case, it is advisable to use compound labels like `Posets•Basics`, `Posets•Basics•Defs` rather than atomic ones as in the example above. Using an obvious input convention, a fragment of such an archive would look like this:

$$
\begin{array}{l}
\texttt{Posets} \rightleftharpoons \underset{P}{\forall} \\
\quad \bullet\texttt{Basics} \rightleftharpoons \\
\quad\quad \bullet\texttt{Defs} \rightleftharpoons \\
\quad\quad\quad \texttt{is-poset}[P] \Leftrightarrow \\
\quad\quad\quad\quad \underset{\underset{P}{\epsilon}[x,y,z]}{\forall} \\
\quad\quad\quad\quad\quad x \underset{P}{\leq} x
\end{array}
$$

$$\left(x \underset{P}{\leq} y \wedge y \underset{P}{\leq} x\right) \Rightarrow x = y$$
$$\left(x \underset{P}{\leq} y \wedge y \underset{P}{\leq} z\right) \Rightarrow x \underset{P}{\leq} z$$
$$\underset{x,y}{\forall}$$
$$\underset{\texttt{converse}[P]}{\epsilon}[x] \Leftrightarrow \underset{P}{\epsilon}[x]$$
$$x \underset{\texttt{converse}[P]}{\leq} y \Leftrightarrow y \underset{P}{\leq} x$$
$$\ldots$$

•Thms $\rightleftharpoons$
$\texttt{is-poset}[P] \Rightarrow \texttt{is-poset}[\texttt{converse}[P]]$
$\ldots$

•Advanced $\rightleftharpoons$
•Defs $\rightleftharpoons \underset{C}{\forall}$
$$\texttt{is-conn-chain}[C, P] \Leftrightarrow \left(\texttt{is-tuple}[C] \wedge \underset{i=1,\ldots,|C|}{\forall} \underset{P}{\epsilon}[C_i] \wedge \underset{i=1,\ldots,|C|-1}{\forall} \texttt{covers}\left[\underset{P}{\leq}, C_i, C_{i+1}\right]\right)$$
$$\texttt{endpt}[C] = C_{|C|}$$
•Thms $\rightleftharpoons$
$$\texttt{is-lattice}[P] \wedge \texttt{is-finite}[P] \Rightarrow \underset{C,D}{\forall}((\texttt{is-conn-chain}[C, P] \wedge \texttt{is-conn-chain}[D, P] \wedge (\texttt{endpt}[C] = \texttt{endpt}[D])) \Rightarrow (|C| = |D|))$$
$\ldots$

Groups $\rightleftharpoons$
•Basics $\rightleftharpoons$
•Defs $\rightleftharpoons \ldots$
•Thms $\rightleftharpoons \ldots$
•Advanced $\rightleftharpoons$
•Defs $\rightleftharpoons \ldots$
•Thms $\rightleftharpoons \ldots$

Using the archive above, one would refer to the "advanced theorems" of the poset package by the compound label Posets•Advanced•Thms. The package translates to *Theorema* as follows:

Posets $\Leftrightarrow$ Posets • Basics $\wedge$ Posets • Advanced
Posets • Basics $\Leftrightarrow$ Posets • Basics • Defs $\wedge$ Posets • Basics • Thms
Posets • Basics • Defs $\Leftrightarrow$
$$\underset{P}{\forall}((\texttt{is-poset}[P] \Leftrightarrow \underset{\underset{P}{\epsilon}[x,y,z]}{\forall}(x \underset{P}{\leq} x \wedge ((x \underset{P}{\leq} y \wedge y \underset{P}{\leq} x) \Rightarrow (x = y))$$
$$\wedge((x \underset{P}{\leq} y \wedge y \underset{P}{\leq} z) \Rightarrow x \underset{P}{\leq} z))) \wedge \underset{x,y}{\forall}(\underset{\texttt{converse}[P]}{\epsilon}[x] \Leftrightarrow \underset{P}{\epsilon}[x] \wedge$$
$$x \underset{\texttt{converse}[P]}{\leq} y \Leftrightarrow y \underset{P}{\leq} x) \wedge \ldots)$$
Posets • Basics • Thms $\Leftrightarrow \underset{P}{\forall}(\texttt{is-poset}[P] \Rightarrow \texttt{is-poset}[\texttt{converse}[P]] \wedge \ldots)$
Posets • Advanced $\Leftrightarrow$ Posets • Advanced • Defs $\wedge$ Posets • Advanced • Thms

$\texttt{Posets} \bullet \texttt{Advanced} \bullet \texttt{Defs} \Leftrightarrow \underset{PC}{\forall\forall}(\texttt{is-conn-chain}[C,P] \Leftrightarrow (\texttt{is-tuple}[C] \wedge$
$\underset{i=1,\ldots,|C|}{\forall} \underset{P}{\in}[C_i] \wedge \underset{i=1,\ldots,|C|-1}{\forall} \texttt{covers}[\underset{P}{\leq}, C_i, C_{i+1}]) \wedge (\texttt{endpt}[C] = C_{|C|}) \wedge \ldots)$
$\texttt{Posets} \bullet \texttt{Advanced} \bullet \texttt{Thms} \Leftrightarrow$
$\underset{P}{\forall}((\texttt{is-lattice}[P] \wedge \texttt{is-finite}[P] \Rightarrow \underset{C,D}{\forall}((\texttt{is-conn-chain}[C,P] \wedge$
$\texttt{is-conn-chain}[D,P] \wedge (\texttt{endpt}[C] = \texttt{endpt}[D])) \Rightarrow (|C| = |D|))) \wedge \ldots)$
$\texttt{Groups} \Leftrightarrow \texttt{Groups} \bullet \texttt{Basics} \wedge \texttt{Groups} \bullet \texttt{Advanced}$
$\texttt{Groups} \bullet \texttt{Basics} \Leftrightarrow \texttt{Groups} \bullet \texttt{Basics} \bullet \texttt{Defs} \wedge \texttt{Groups} \bullet \texttt{Basics} \bullet \texttt{Thms}$
$\texttt{Groups} \bullet \texttt{Basics} \bullet \texttt{Defs} \Leftrightarrow \ldots$
$\texttt{Groups} \bullet \texttt{Basics} \bullet \texttt{Thms} \Leftrightarrow \ldots$
$\texttt{Groups} \bullet \texttt{Advanced} \Leftrightarrow \texttt{Groups} \bullet \texttt{Advanced} \bullet \texttt{Defs} \wedge$
$\texttt{Groups} \bullet \texttt{Advanced} \bullet \texttt{Thms}$
$\texttt{Groups} \bullet \texttt{Advanced} \bullet \texttt{Defs} \Leftrightarrow \ldots$
$\texttt{Groups} \bullet \texttt{Advanced} \bullet \texttt{Thms} \Leftrightarrow \ldots$

Note that if one uses an atomic label inside a hierarchy of labels, the atomic one will be treated as if the surrounding hierarchy were not present, but an atomic label will be considered for compound labels specified via the input convention exemplified above.

A word of caution about the semantics of compound labels: Since there is only a limited supply of boolean functions of a fixed arity but no bound on the number of labels used in a package, the • in a compound label like `Groups•Basics` should not be interpreted like having the type $\mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool}$ but rather $\mathit{String} \times \mathit{String} \rightarrow \mathit{Bool}$. Atomic labels are then viewed as the degenerate case $\mathit{String} \rightarrow \mathit{Bool}$.

Quantifiers can be combined with labels in a natural way. As explained before, quantifiers may be prefixed to blocks; this remains true for those blocks that form the body of a package. An instance of this usage can be seen in the previous example after the label `Posets•Advanced•Defs` and similar places. Quantifiers appearing in package heads higher up in the hierarchy are distributed to all formulae in the blocks underneath; this is what happened to the quantifier on P in the package labelled `Posets`.

Sometimes existential quantifiers can also be used for introducing local names for subalgorithms that should only be visible within the package containing the definition and properties of the main algorithm. Here is an example, where `qsort` has auxiliary algorithms `left`, `right` and `pick`:

$\texttt{Tuples} \rightleftharpoons$
$\texttt{Sorting} \rightleftharpoons$
$\texttt{BubbleSort} \rightleftharpoons \ldots$
$\texttt{MergeSort} \rightleftharpoons \ldots$
$\texttt{QuickSort} \rightleftharpoons$
$\underset{\texttt{left,right,pick}}{\exists}$
$\texttt{qsort}[\langle\rangle] = \langle\rangle$
$\underset{\texttt{is-non-empty-tuple}[X]}{\forall}$
$\texttt{qsort}[X]{=}\texttt{qsort}[\texttt{left}[X]] \asymp \langle \texttt{pick}[X] \rangle \asymp \texttt{qsort}[\texttt{right}[X]]$

$$\begin{array}{l}
\underset{\texttt{is-tuple}[X]}{\forall} \\
\quad \underset{l,r}{\forall}((l \in \texttt{left}[X] \wedge r \in \texttt{right}[X]) \Rightarrow l \leq \texttt{pick}[X] \leq r) \\
\quad \texttt{left}[X] \asymp \langle \texttt{pick}[X] \rangle \asymp \texttt{right}[X]) \approx X \\
\quad |\texttt{left}[X]| < |X| \wedge |\texttt{right}[X]| < |X| \\
\texttt{PartialCorrectness} \rightleftharpoons \underset{\texttt{is-tuple}[X]}{\forall} \\
\quad \texttt{is-sorted}[\texttt{qsort}[X]] \\
\quad \texttt{qsort}[X] \approx X
\end{array}$$

In this example, the names `left` and `right` used in the package `QuickSort` could also occur in `BubbleSort` and `MergeSort` with different meanings (i.e. having different properties).

4. Namespaces for Structured Symbol Reference

Besides the label hierarchies explained in the previous section, the language of archives provides another central construct needed for organizing large knowledge repositories: *Namespaces* are used as a tool for resolving ambiguity of symbols. Thus there are altogether three types of symbols in the language of archives:

- *Global symbols* are practical for notions that are highly important in the whole of mathematics so that one does not want to refer to them via any prefixed namespace, e.g. the symbols $\in$, $\subseteq$, $\emptyset$ of set theory.
- *Local symbols* are visible only in the subhierarchy of blocks below their point of introduction, hence they avoid name clashes with symbols used in a parallel block (see the example immediately above).
- *Symbols in a namespace* are disambiguated by prefixing symbols with a "path" providing the necessary context information. In analogy to certain programming languages we call such a path a namespace.

In the remainder of this section, we will discuss the third type of symbols only.

4.1. Symbols bound to a Namespace

Namespaces are realized in a fashion analogous to the operation objects mentioned in Subsections 2.2 and 2.3, namely as higher-order functions wrapping the symbols for distinguishing their different meanings. Consider the following package:

$$\begin{array}{l}
\texttt{BinRel} : \langle \texttt{is-transitive} \rangle \rightleftharpoons \\
\quad \underset{\sim}{\forall}\texttt{is-transitive}[\sim] \Leftrightarrow \underset{x,y,z}{\forall}(x \sim y \wedge y \sim z \Rightarrow x \sim z) \\
\quad \ldots
\end{array}$$

Its translation to *Theorema* will be:

$$\texttt{BinRel} \Leftrightarrow \underset{\sim}{\forall}\Big(\underset{\texttt{BinRel}}{\texttt{is-transitive}}[\sim] \Leftrightarrow \underset{x,y,z}{\forall}(x \sim y \wedge y \sim z \Rightarrow x \sim z)\Big) \wedge \ldots$$

Hence the predicate symbol `BinRel`[`is-transitive`] will be distinguished from other occurences of `is-transitive`, e.g. in the context of set theory:

$$\begin{array}{l}\texttt{SetTh} : \langle \texttt{is-transitive} \rangle \rightleftharpoons \\ \quad \underset{z}{\forall} \Big(\texttt{is-transitive}[z] \Leftrightarrow \underset{x,y}{\forall} (x \in y \wedge y \in z \Rightarrow x \in z) \Big) \\ \quad \ldots \end{array}$$

This will of course be translated to:

$$\texttt{SetTh} \Leftrightarrow \underset{z}{\forall} \Big(\underset{\texttt{SetTh}}{\texttt{is-transitive}}[z] \Leftrightarrow \underset{x,y}{\forall} (x \in y \wedge y \in z \Rightarrow x \in z) \Big) \wedge \ldots$$

In the above example, we have employed the namespaces `BinRel` and `SetTh`, in both cases on the symbol `is-transitive`. Note that the identifiers `BinRel` and `SetTh` are interpreted as labels as well as namespaces, but their internal representation is distinct: As explained in Section 3, the labels are actually interpreted as $\bullet$["BinRel"] and $\bullet$["SetTh"], respectively. In contrast, the namespaces occur as function constants in the corresponding internal representation `BinRel`[`is-transitive`] and `SetTh`[`is-transitive`].

In general, we can bind a sequence of symbols $\sigma_1, \ldots, \sigma_n$ to the namespace associated with a label L, its so-called *home namespace*, in a block of formulae:

$$\begin{array}{l} L : \langle \sigma_1, \ldots, \sigma_n \rangle \rightleftharpoons \\ \quad \phi_1 \\ \quad \ldots \\ \quad \phi_m \end{array}$$

This is translated to *Theorema* as $L \Leftrightarrow (\phi_1 \wedge \ldots \wedge \phi_m)_{\sigma_1 \leftarrow L[\sigma_1], \ldots, \sigma_n \leftarrow L[\sigma_n]}$. In other words, the block is interpreted in the usual way as a conjunction, but with the specified symbols being replaced by their wrapped correlates $L[\sigma_1]$,...,$L[\sigma_n]$; we then say that $\sigma_1, \ldots, \sigma_n$ are bound to the namespace L. Such packages will be called *wrapped*, as opposed to the *plain* ones considered in Section 3.

For referring to a symbol from a *foreign namespace* (i.e. a namespace different from the current home namespace), one normally would have to use its full name (symbol with the namespace underneath). For improving readability, this can be abbreviated by "opening" the foreign namespace for "importing" the needed symbols. Let us consider building up the theory of real numbers (with $\mathbb{R}$ being the universe) and stating that certain relations are transitive in the sense of binary relations defined above. In this case we would write:

$$\begin{array}{l} \texttt{Reals} : \langle \ldots, \sin, \cos \rangle \rightleftharpoons \\ \quad \ldots \\ \quad \underset{x}{\forall} \big(\sin[x]^2 + \cos[x]^2 = 1 \big) \\ \quad \ldots \\ \quad \texttt{RealRelations} \rightleftharpoons \underset{x}{\forall} \\ \qquad \texttt{BinRel} : \langle \texttt{is-transitive} \rangle \\ \qquad \texttt{is-transitive}[<] \\ \qquad \underset{a,b}{\forall} (a \sqsubset b \Leftrightarrow |a - x| < |b - x|) \Rightarrow \texttt{is-transitive}[\sqsubset] \end{array}$$

The general situation is as follows: The *namespace declaration* $N : \langle \sigma_1, \ldots \sigma_n \rangle$ translates a formula ϕ within its block to $\phi_{\sigma_1 \leftarrow N[\sigma_1], \ldots, \sigma_n \leftarrow N[\sigma_n]}$. Let us note in

passing that home namespaces as introduced above can actually be seen as a shortcut for the following plain package combined with a namespace declaration:

$$
\begin{array}{l}
L \rightleftharpoons \\
\quad L : \langle \sigma_1, \ldots \sigma_n \rangle \\
\quad \phi_1 \\
\quad \ldots \\
\quad \phi_m
\end{array}
$$

So home namespaces and foreign namespaces could be identified, but we prefer to keep the distinction because it may be useful for formula retrieval and related MKM tasks. We will have to say more about this in Section 5.

4.2. Functors via Namespaces

As mentioned above, namespaces are realized in a similar fashion as the operation objects of categories and functors. In fact, *operation objects* are identical to namespaces from a semantical point of view. The difference is more of a psychological nature: Operation objects are typically conceived as the domains residing in a certain category or constructed by a certain functor. As an example consider the following formula defining the category of semigroups:

$$
\begin{array}{l}
\underset{S}{\forall} \\
\quad \texttt{Semigroup}[S] \Leftrightarrow \\
\quad\quad S : \langle \epsilon, * \rangle \\
\quad\quad \underset{x,y}{\forall} \\
\quad\quad\quad \epsilon[x * y] \\
\quad\quad\quad (x * y) * z = x * (y * z)
\end{array}
$$

Using the same mechanism as explained above, this is translated to:

$$\underset{S}{\forall}\left(\texttt{Semigroup}[S] \Leftrightarrow \underset{x,y}{\forall}\left(\underset{S}{\epsilon}[x \underset{S}{*} y] \wedge \left(x \underset{S}{*} y\right) \underset{S}{*} z = x \underset{S}{*}\left(y \underset{S}{*} z\right)\right)\right).$$

But observe that here we have used a variable rather than a constant for opening a namespace: The symbols ϵ and $*$ are bound to the variable namespace S.

Using compound labels (see Subsection 3.2), it is very natural to build up a parallel *hierarchy of namespaces* for binding the miscellaneous symbols introduced in them. In fact, this is what happens automatically when symbols are bound to the home namespace corresponding to a certain (atomic or compound) label. The above fragment from the theory of relations could occur inside a surrounding hierarchy:

$$
\begin{array}{l}
\texttt{Algebra} \rightleftharpoons \\
\quad \bullet\texttt{Relations} \rightleftharpoons \\
\quad\quad \bullet\texttt{UnRel} : \langle \ldots \rangle \rightleftharpoons \ldots \\
\quad\quad \bullet\texttt{BinRel} : \langle \texttt{is-transitive} \rangle \rightleftharpoons \\
\quad\quad\quad \underset{\sim}{\forall}\left(\texttt{is-transitive}[\sim] \Leftrightarrow \underset{x,y,z}{\forall}(x \sim y \wedge y \sim z \Rightarrow x \sim z)\right) \\
\quad\quad \ldots
\end{array}
$$

Its translation to *Theorema* reads as follows:

$$\texttt{Algebra} \Leftrightarrow (\texttt{Algebra} \bullet \texttt{Relations} \wedge \ldots)$$
$$\texttt{Algebra} \bullet \texttt{Relations} \Leftrightarrow$$
$$\quad \texttt{Algebra} \bullet \texttt{Relations} \bullet \texttt{UnRel} \wedge \texttt{Algebra} \bullet \texttt{Relations} \bullet \texttt{BinRel} \wedge \ldots$$
$$\texttt{Algebra} \bullet \texttt{Relations} \bullet \texttt{UnRel} \Leftrightarrow \ldots$$
$$\texttt{Algebra} \bullet \texttt{Relations} \bullet \texttt{BinRel} \Leftrightarrow$$
$$\quad \underset{\sim}{\forall} \Big(\underset{\texttt{Algebra}\bullet\texttt{Relations}\bullet\texttt{BinRel}}{\texttt{is-transitive}}[\sim] \Leftrightarrow \underset{x,y,z}{\forall} (x \sim y \wedge y \sim z \Rightarrow x \sim z) \Big) \wedge \ldots$$

Namespaces also provide a handy notation for the domains generated by a functor, where a domain is viewed as a special case of a wrapped package. For example, one can realize the (multiplicatively written) semigroup of naturals in terms of the global symbols $\mathbb{N}$,+ and $\in$ as follows (note the difference between $\in$ and ϵ):

$$\texttt{NaturalSemigroup} : \langle \epsilon, * \rangle \rightleftharpoons \underset{x,y}{\forall}$$
$$\quad \epsilon[x] \Rightarrow x \in \mathbb{N}$$
$$\quad x * y = x + y$$

Such definitions are also called *introduction functors* since they introduce a domain without other domains as arguments (in contrast to "normal" functors like the direct product defined below). Note, however, that introduction functors may have *parameters*, i.e. arguments that do not represent domains. An example would be the n-dimensional real vector spaces (where $n \in \mathbb{N}$ is a parameter).

A more degenerate example of an introduction functor is given by the zero group (note the difference between $\mathcal{O}$ and 0):

$$\texttt{Grp} : \langle \mathcal{O} \rangle \rightleftharpoons$$
$$\quad \underset{g,h}{\forall}$$
$$\qquad \underset{\mathcal{O}}{\epsilon}[g] \Leftrightarrow g = \underset{\mathcal{O}}{0}$$
$$\qquad g \underset{\mathcal{O}}{+} h = \underset{\mathcal{O}}{0}$$
$$\qquad \underset{\mathcal{O}}{-} g = \underset{\mathcal{O}}{0}$$

Now we could use $\underset{\texttt{Grp}}{\mathcal{O}}$ for denoting the zero group; similar functors could be defined for the zero objects of other categories like monoids `Mon`.

A very common example of a *bivariate functor* is the direct product introduced in Subsection 2.3, which could be written directly in a package (assuming that `is-tuple`, `card` and the tuple selector are globally defined):

$$\texttt{DP} : \langle \times \rangle \rightleftharpoons \underset{G,H}{\forall}$$
$$\quad G \times H = \underset{D}{\ni} \underset{x,y}{\forall}$$
$$\quad D : \langle \epsilon, +, 0, - \rangle$$
$$\quad \epsilon[x] \Leftrightarrow \Big(\texttt{is-tuple}[x] \wedge (\texttt{card}[x] = 2) \wedge \underset{G}{\epsilon}[x_1] \wedge \underset{H}{\epsilon}[x_2] \Big)$$
$$\quad x + y = \langle x_1 \underset{G}{+} y_1, x_2 \underset{H}{+} y_2 \rangle$$
$$\quad 0 = \langle \underset{G}{0}, \underset{H}{0} \rangle$$

$$-x = \langle \underset{G}{-} x_1, \underset{H}{-} x_2 \rangle$$

Here is an equivalent definition of the same functor, which dispenses with the description quantifier ɜ and is more in the spirit of archives:

$$\begin{array}{l}
\texttt{DP} : \langle \times \rangle \rightleftharpoons \underset{G,H}{\forall} \\
\quad G \times H : \langle \epsilon, +, 0, - \rangle \\
\quad \underset{x,y}{\forall} \\
\qquad \epsilon[x] \Leftrightarrow \texttt{is-tuple}[x] \wedge \texttt{card}[x] = 2 \wedge \underset{G}{\epsilon}[x_1] \wedge \underset{H}{\epsilon}[x_2] \\
\qquad x + y = \langle x_1 \underset{G}{+} y_1, x_2 \underset{H}{+} y_2 \rangle \\
\qquad 0 = \langle \underset{G}{0}, \underset{H}{0} \rangle \\
\qquad -x = \langle \underset{G}{-} x_1, \underset{H}{-} x_2 \rangle
\end{array}$$

Note that here we have for the first time explicitly used a compound term for denoting a namespace. The translation proceeds as usual and results in the following *Theorema* formulae:

$$\begin{array}{l}
\texttt{DP} \Leftrightarrow \underset{G,H}{\forall} \underset{x,y}{\forall} \left(\left(\underset{\underset{\texttt{DP}}{G \times H}}{\epsilon}[x] \Leftrightarrow \texttt{is-tuple}[x] \wedge \texttt{card}[x] = 2 \wedge \underset{G}{\epsilon}[x_1] \wedge \underset{H}{\epsilon}[x_2] \right) \wedge \right. \\
\qquad \left. x \underset{\underset{\texttt{DP}}{G \times H}}{+} y = \langle x_1 \underset{G}{+} y_1, x_2 \underset{H}{+} y_2 \rangle \ \wedge \ \underset{\underset{\texttt{DP}}{G \times H}}{0} = \langle \underset{G}{0}, \underset{H}{0} \rangle \ \wedge \ \underset{\underset{\texttt{DP}}{G \times H}}{-} x = \langle \underset{G}{-} x_1, \underset{H}{-} x_2 \rangle \right)
\end{array}$$

Observe that the formula above (except for the label DP) is also what the *Theorema* function `FlattenKB` would have made out of the earlier definition (see Subsection 2.3). In fact, the general usage of ɜ is nonconstructive, so it is typically restricted to very specific settings where it can be eliminated (like *Theorema* does in explicit definitions). It is therefore advisable to avoid it, but it may still be used if desired.

4.3. Categories via Namespaces

Namespace declarations also facilitate the *specification of categories*, as we have seen in the example of semigroups. In mathematics, categories are generally built up by gradual refinement—monoids, groups, abelian groups, rings, etc; this can be compared to the idea of inheritance in computer science. Consider the following archive version of a fragment of this refinement chain (omitting the outermost quantifiers on R, G, D, M):

$$\begin{array}{l}
\texttt{Ring}[R] \Leftrightarrow \\
\quad R : \langle \epsilon, +, 0, -, *, 1 \rangle \\
\quad \texttt{AbGrp}[R] \\
\quad \texttt{Monoid}[\lceil \epsilon| \to \epsilon, \circ| \to *, 1| \to 1 \rceil] \\
\quad \texttt{Distributive}[R]
\end{array}$$

$$\begin{array}{l}
\texttt{AbGrp}[G] \Leftrightarrow \\
\quad G : \langle \epsilon, +, 0, - \rangle \\
\quad \texttt{Group}\left[\lceil \epsilon \mapsto \epsilon, * \mapsto +, 1| \to 0, \square^{-1} \mapsto - \rceil\right] \\
\quad \underset{\epsilon[x,y]}{\forall} x + y = y + x
\end{array}$$

$$
\begin{aligned}
&\texttt{Group}[G] \Leftrightarrow \\
&\quad G : \langle \epsilon, *, 1, \square^{-1} \rangle \\
&\quad \texttt{Monoid}[\lceil \epsilon \mapsto \epsilon, \circ \mapsto *, 1 \mapsto 1 \rceil] \\
&\quad \underset{\epsilon[x]}{\forall} \\
&\qquad \epsilon \left[x^{-1} \right] \\
&\qquad x * x^{-1} = 1 \\
&\texttt{Distributive}[D] \Leftrightarrow \\
&\quad D : \langle \epsilon, +, * \rangle \\
&\quad \underset{\epsilon[x,y,z]}{\forall} (x + y) * z = x * z + y * z \\
&\texttt{Monoid}[M] \Leftrightarrow \\
&\quad M : \langle \epsilon, \circ, 1 \rangle \\
&\quad \underset{\epsilon[x,y,z]}{\forall} \\
&\qquad \epsilon[x \circ y] \\
&\qquad \epsilon[1] \\
&\qquad (x \circ y) \circ z = x \circ (y \circ z)
\end{aligned}
$$

In the formulae above, it is sometimes necessary to translate between certain symbols (e.g. between additive and multiplicative group notation). This is realized by using *theory interpretations* written as $\lceil \sigma_1 \mapsto \tau_1, \ldots, \sigma_n \mapsto \tau_n \rceil$. The intuitive meaning of this construct is quite clear: It is the finitely supported function that maps the symbols $\sigma_1, \ldots, \sigma_n$ to $\tau_1, \ldots, \tau_n$ and leaves all other inputs unchanged (this last requirement is only made for definiteness and could be omitted). More precisely, it could be defined as the following lambda expression:

$$
\underset{\sigma}{\lambda} \begin{cases} \tau_1 & \Leftarrow \sigma = \sigma_1 \\ \vdots & \quad \vdots \\ \tau_n & \Leftarrow \sigma = \sigma_n \\ \sigma & \Leftarrow \texttt{True} \end{cases}
$$

Its purpose is to build a new operation object with appropriate operations; we will indicate its usage below. For the *Theorema* translation, the lambda expressions for the theory interpretations are retained but the usual replacement for the bound symbols in a namespace are carried out only in the right-hand sides of the lambda expression. Thus the formulae above will become:

$$
\begin{aligned}
&\underset{R}{\forall} \left(\texttt{Ring}[R] \Leftrightarrow \texttt{AbGrp}[R] \wedge \texttt{Monoid}[\lceil \epsilon \mapsto \underset{R}{\epsilon}, \circ \mapsto \underset{R}{*}, 1 \mapsto \underset{R}{1} \rceil] \wedge \texttt{Distributive}[R] \right) \\
&\underset{G}{\forall} \left(\texttt{AbGrp}[G] \Leftrightarrow \texttt{Group}[\lceil \epsilon \mapsto \underset{G}{\epsilon}, * \mapsto \underset{G}{+}, 1 \mapsto \underset{G}{0}, \square^{-1} \mapsto \underset{G}{-} \rceil] \wedge \underset{\underset{G}{\epsilon}[x,y]}{\forall} x \underset{G}{+} y = y \underset{G}{+} x \right) \\
&\underset{G}{\forall} \Big(\texttt{Group}[G] \Leftrightarrow \texttt{Monoid}[\lceil \epsilon \mapsto \underset{G}{\epsilon}, \circ \mapsto \underset{G}{*}, 1 \mapsto \underset{G}{1} \rceil] \wedge \\
&\qquad \underset{\underset{G}{\epsilon}[x]}{\forall} \underset{\underset{G}{\epsilon}[y]}{\forall} \underset{\underset{G}{\epsilon}[z]}{\forall} (\epsilon[x^{-1}] \wedge x \underset{G}{*} x^{-1} = \underset{G}{1}) \Big) \\
&\underset{D}{\forall} \left(\texttt{Distributive}[D] \Leftrightarrow \underset{\underset{D}{\epsilon}[x,y,z]}{\forall} (x \underset{D}{+} y) \underset{D}{*} z = x \underset{D}{*} z \underset{D}{+} y \underset{D}{*} z \right)
\end{aligned}
$$

$$\underset{M}{\forall}\Big(\texttt{Monoid}[M] \Leftrightarrow \underset{\underset{M}{\epsilon}[x,y,z]}{\forall}\Big(\underset{M}{\epsilon}[x \underset{M}{\circ} y] \wedge \underset{M}{\epsilon}[1] \wedge \big(x \underset{M}{\circ} y\big) \underset{M}{\circ} z = x \underset{M}{\circ} \big(y \underset{M}{\circ} z\big)\Big)\Big)$$

The role of theory interpretations becomes clear when we consider their behavior in proofs (or MKM tasks). So assume a proof situation in which $\texttt{Group}[\mathbb{Q}]$ occurs in the assumptions. By instantiation and modus ponens on the Group definition above we obtain (besides the invertibility axiom):

$$\texttt{Monoid}[\lceil \epsilon \mapsto \underset{\mathbb{Q}}{\epsilon}, \circ \mapsto \underset{\mathbb{Q}}{*}, 1 \mapsto \underset{\mathbb{Q}}{1}\rceil]$$

Writing M for $\lceil \epsilon \mapsto \underset{\mathbb{Q}}{\epsilon}, \circ \mapsto \underset{\mathbb{Q}}{*}\rceil$, the definition of Monoid further yields:

$$\underset{\underset{M}{\epsilon}[x,y,z]}{\forall}\Big(\underset{M}{\epsilon}[x \underset{M}{\circ} y] \wedge \underset{M}{\epsilon}[\underset{M}{1}] \wedge \big(x \underset{M}{\circ} y\big) \underset{M}{\circ} z = x \underset{M}{\circ} \big(y \underset{M}{\circ} z\big)\Big).$$

Since symbols like $\underset{M}{\epsilon}$ are internally represented as $M[\epsilon]$ and the like, the rule of β-reduction and case distinction on the lambda expression above finally leads to:

$$\underset{\underset{\mathbb{Q}}{\epsilon}[x,y,z]}{\forall}\Big(\underset{\mathbb{Q}}{\epsilon}[x \underset{\mathbb{Q}}{*} y] \wedge \underset{\mathbb{Q}}{\epsilon}[1] \wedge \big(x \underset{\mathbb{Q}}{*} y\big) \underset{\mathbb{Q}}{*} z = x \underset{\mathbb{Q}}{*} \big(y \underset{\mathbb{Q}}{*} z\big)\Big)$$

Let us remark that in *Theorema* the application of β-reduction and case distinction can be combined into a single computation step that can be intuitively understood as applying finitely supported functions to arguments.

5. Basic Operations on Archives

Archives are not an aim in themselves—we would like to use them in various operations, specifically for knowledge buildup and retrieval, e.g. in the context of theorem proving and algorithm synthesis. Due to the intricate nature of these operations, however, we can only give a small flavor of how they could look like in the frame of archives (see Subsections 5.5 and 5.6).

Starting with basic I/O operations on archives (Subsection 5.1), we consider next their translation to plain predicate logic (Subsection 5.2). Several operations for restructuring archives are available, "mixing" them in various ways (Subsection 5.3) or manipulating their parts (Subsection 5.4).

5.1. Loading and Saving Archives

In order to load an archive saved under the filename `Algebra.nb` in the home directory, one uses the command:

```
archive = LoadArchive["Algebra.nb"]
```

By this call, the box structure of the notebook `"Algebra.nb"` is parsed into a *Mathematica* expression [30], subsequently stored in the variable `archive`. The underlying expression language is that of *Theorema* [14], extended by the language features presented in the preceding sections.

As a second argument, the user can also specify a keyword referring to a label of the structured notebook, extracting the package headed by the label. Consider for example loading a "subsection" entitled `"BasicProperties"`:

```
archive = LoadArchive["Algebra.nb", "BasicProperties"]
```

For viewing and browsing archives (in particular those generated by the operations described in Subsections 5.3 and 5.4), an inverse operation to `LoadArchive` is needed: the command

```
SaveArchive[archive, fname]
```

generates from `archive` a *Mathematica* notebook to be stored in the file `fname`. The result wil be the canonical box structure of the archive (typically not identical but equivalent to the original structured notebook).

There are two other I/O commands of lesser importance, mainly intended for writing and reading intermediate portions of an archive. The corresponding commands are `WriteArchive[archive, fname]` and `ReadArchive[filename]`.

5.2. Projecting Archives to Plain Predicate Logic

As we explained already in Sections 3 and 4, the translation of an archive to plain *Theorema* involves a partial loss of structure (this is why we call this idempotent operation a projection). In a sense, an archive is a logical formula plus organizational annotations: the annotations can be translated to logic but the resulting formula blurs the distinction between "logic" and "organization", e.g. one cannot say in general whether a $\Leftrightarrow$ stems from a $\rightleftharpoons$.

We have implemented two functions for projecting archives: Besides the translation to a flat set of formulae (called an *assumption list* in *Theorema*) considered up to now, there is also a possibility of retaining the hierarchical structure encoded in the labels by rephrasing them as nested `Theory` environments of *Theorema*. (As explained in Subsection 2.1, these constructs are logic-external but otherwise essentially equivalent.) The flat translator is called by `ArchiveToTmaTheory[archive]`, the other one by `ArchiveToAsml[archive]`.

Here is how the package from the beginning of Subsection 3.2 looks like as a `Theory` in Theorema:

```
Theory["Algebra",
    Theory["GroupTheory"]
    Theory["LatticeTheory"] ]

Theory["GroupTheory",
    Theory["Magmas"]
    Theory["Semigroups"]
    ... ]
```

$$\texttt{Theory}\Big[\texttt{"Magmas"}, \underset{M}{\forall}\,(\texttt{is-magma}[M] \Leftrightarrow \underset{\underset{M}{\in}[x,y]}{\forall}\, \underset{M}{\in}[x \underset{M}{\circ} y])\Big]$$

```
Theory["Semigroups",
```

$$\underset{S}{\forall}\Big(\texttt{is-sgroup}[S] \Leftrightarrow \texttt{is-magma}[S] \wedge \underset{\in S^{[x,y,z]}}{\forall}(x \underset{S}{\circ} y) \underset{S}{\circ} z = x \underset{S}{\circ} (y \underset{S}{\circ} z)\Big)\,]$$

```
Theory["LatticeTheory",
    Theory["Posets"]
    Theory["Chains"]
    ...]
```

Let us emphasize again, however, that these translators are mainly provided for illuminating the semantics of logic-internal labels and namespaces and for connecting with other components of the *Theorema* framework.

5.3. Merging and Joining Archives

Two operations are available for "mixing" archives `arch1` and `arch2`, differing in how they treat global symbols: While the command `MergeArchive[arch1,arch2]` identifies them (meaning the symbols remain global so that they refer to the same concepts), its analog `JoinArchive[arch1,arch2]` wraps them into separating namespaces whose identifiers are the top-level labels of the archives (thus creating the corresponding home namespaces). Both operations generate the conjunction of the original archives.

As an example, consider merging archives `GroupTheory` and `LatticeTheory` as a preparation for asserting the theorem that the subgroups of a given group form a lattice. In this case, (typically globally defined) set-theoretic symbols like $\subseteq$ and $\cap$ should of course be identified.

A typical example where joining would be the method of choice is when one wants to bring together archives like `AffineGeometry` and `ProjectiveGeometry`, where e.g. the predicate symbols `is-subspace`, `is-basis` and the function symbol `dimension` must be kept apart (despite their obvious intuitive correspondences).

Intuitively, merging is similar to set-theoretic union, whereas joining acts like a disjoint union. Both operations are subsumed by the following command:

```
CombineArchive[arch1, arch2, {sym, nmsp1, nmsp2}, ...]
```

Again the formulae of the archives are conjoined, with all global symbols remaining global except for those explicitly specified after `arch1` and `arch2`: The symbol `sym` is respectively wrapped into the namespaces `nmsp1` and `nmsp2` in the subarchives corresponding to `arch1` and `arch2`.

A situation where this occurs is the following example: Assume we want to combine two archives `TopologicalFields` and `OrderedFields`. In this case, symbols like `is-field` should be shared, but one must take care not to confuse the topological notion `is-complete` with its order-theoretic namesake. In fact, both of these will enter into the same type of theorem asserting that $\mathbb{R}$ is complete (in the respective senses), both as an ordered field and as a topological field.

5.4. Operations on Subarchives

One can refer to subarchives by their labels. In the following archive example, the label `TupleProperties` • `Concatenation` • `Definition` refers to the formula defining concatenation:

$$\texttt{TupleProperties} : \langle \texttt{is-tuple}, \texttt{is-empty-tuple} \rangle \rightleftharpoons$$
$$\ldots$$
$$\texttt{Concatenation} : \langle \asymp \rangle \rightleftharpoons$$
$$\texttt{Definition} \rightleftharpoons$$
$$\underset{\bar{x},\bar{y}}{\forall} (\langle \bar{x} \rangle \asymp \langle \bar{y} \rangle = \langle \bar{x}, \bar{y} \rangle)$$
$$\texttt{Properties} \rightleftharpoons$$
$$\underset{\texttt{is-tuple}[X]}{\forall} (X \asymp \langle\rangle = X)$$
$$\underset{\texttt{is-tuple}[X]}{\forall} (\langle\rangle \asymp X = X)$$
$$\underset{\texttt{is-tuple}[X]}{\forall} ((X \asymp X = X) \Rightarrow \texttt{is-empty-tuple}[X])$$
$$\underset{\texttt{is-tuple}[X,Y,Z]}{\forall} ((X \asymp Y) \asymp Z = X \asymp (Y \asymp Z))$$
$$\ldots$$

Moreover, one can also access the fine structure within packages by using the customary position markers (as also provided in *Mathematica*). For example, the position `TupleProperties` • `Concatenation` • `Properties` ∘ 3 ∘ 3 ∘ 2 refers to the subformula `is-empty-tuple`$[X]$. Note that in *Theorema* a quantifier takes three arguments: range, condition, and matrix.

The command `ArchivePart`[`arch`, `pos`] extracts from an archive `arch` the subformula at position `pos`, its counterpart `ArchiveContext`[`arch`, `pos`] yields the surrounding context. Similarly, for inserting archive `arch2` into `arch1` at the label position `lab`, one may use the command `InsertArchive`[`arch1`, `arch2`, `lab`].

5.5. Retrieval in Archives

We regard retrieval as an integral part of theorem proving [9], typically employing a drastically reduced set of inference rules ("symbolic computation proving") for obtaining the formula needed. Textual search is usually not a sufficient solution since this will not find equivalent formulae or "simple" consequences. (For efficiency reasons it may occasionally be useful to apply simpler retrieval mechanisms closer to textual search, so we have also provided them.)

As an example for the idea "retrieval = symbolic computation proving", consider the following proof goal:

$$\tfrac{a+b}{2} \neq 0 \wedge c^2 + 1 \neq 0 \Rightarrow \tfrac{a+b}{2} * \left(c^2 + 1\right) \neq 0$$

One can retrieve the information required if the archive contains the formula:

$$\underset{x,y}{\forall} (x * y = 0 \Rightarrow x = 0 \vee y = 0)$$

But note that this retrieval involves the propositional tautology $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$, handled by suitable steps of "symbolic computation proving".

The current implementation should only be understood as a first step in this direction; it allows *Theorema* provers to execute four retrieval operations offering different levels of sophistication:

- The operation `ArchiveLookup` filters out formulae containing given symbols.
- Using `ArchiveMatch` allows applicative higher-order patterns.
- Finding functions with desired properties is achieved by `SearchFunction`.
- If the function is to be synthesized, one may use `BuildFunction`.

The command `ArchiveLookup`[`arch`, {`sym1`, `sym2`, ...}] filters `arch` for the list of (object, function, predicate) constants `sym1`, `sym2`, ..., returning all positions (in the sense introduced in Subsection 5.4) of formulae containing the symbols.

With the help of the option `Check`, the user can reduce the search space. The default setting is `Check` → `AllArchive`, meaning the check is done in the whole archive. A more restricted (and faster) search can be accomplished by setting `Check` → `HomeNamespaces`. Then the constant symbols will be checked only in home namespaces (see Subsection 4.1). This feature is useful if the user has built the archive in such a way that all crucial properties—in particular, the definitions—of a symbol are asserted under the label designating its home namespace. A case in point is the concatenation symbol $\asymp$ introduced in Subsection 5.4.

Another option, named `Filtering`, can be set either to `AndFiltering` or `OrFiltering`. The latter searches formulae containing at least one of the constant symbols specified, the former requires them to contain all of them. For example, the command

`ArchiveLookup`[`arch`, {+, ∗}, `FilterType` → `AndFiltering`]

returns a list of positions for the formulae containing both + and ∗, for example associativity in rings. By subsequently using `ArchivePart` for each of these positions, one will obtain the corresponding subarchives or subformulae.

If one wants to search in an archive `arch` for formulae having a certain "skeleton" `form`, one can use `ArchiveMatch`[`arch`, `form`]. The skeleton may be an arbitrary formula of the archive language, having additional metavariables designated by a superscripted question mark (subsequently transformed to patterns of *Mathematica* to be interpreted by its higher-order applicative matching algorithms). Suppose, for example, that we want to search for an invariant $S^?$ for a fixed equivalence relation $\sim$. Then we would employ the following skeleton:

$$\underset{a^?,b^?}{\forall}(a^? \sim b^? \Rightarrow S^?[a^?] = S^?[b^?])$$

With its default setting `MatchSubformulae` → `False`, matching is restricted to full *Theorema* formulae (i.e. the leaves of the $\rightleftharpoons$ hierarchy). For descending into the internal structure of the *Theorema* formulae, one can set `MatchSubformulae` → `True`. For example, the above skeleton characterizing invariants could occur in the characteristic property of canonical simplifiers:

$$\underset{x}{\forall}(\sigma[x] \sim x) \wedge \underset{x,y}{\forall}(x \sim y \Rightarrow \sigma[x] = \sigma[y])$$

As stated above, we see retrieval in its broader sense as a sequence of simple inferences. This idea is embodied in the two remaining commands `SearchFunction` and `BuildFunction`, which can therefore be seen as issuing a semantic search for functions specified by certain properties. Whereas the former associates the functions only with specific function constants already present in the archive, the latter may construct terms denoting the functions. Both types of search are typically needed in the context of algorithm synthesis [9].

The first type of semantic search is intiated by `SearchFunction`[`arch`, `form`], where the unknown functions in `form` are designated as for `ArchiveMatch`. As an example, consider the problem of synthesizing the merge-sort algorithm as described in [12]. At a certain stage one has to come up with two functions on tuples, $f^?$ and $g^?$, such that the concatenation of $f^?[X]$ and $g^?[X]$ is a permuted version of X. Written in the formalism employed there, this means one takes an arbitrary but fixed tuple X_0 and searches

$$(f^?[X_0] \asymp g^?[X_0]) \approx X_0$$

in an archive on tuple theory, which will typically contain formulae like:

$$\underset{\texttt{is-tuple}[T]}{\forall}((\texttt{left-part}[T] \asymp \texttt{right-part}[T]) = T)$$
$$\underset{\texttt{is-tuple}[T]}{\forall}(T \approx T)$$

Extracting from the archive all unary function symbols (including here `left-part` and `right-part`) and substituting them for $f^?$ and $g^?$, the algorithm will then proceed to prove the resulting candidate instances by using a suitably restricted set of inferences. In order to restrict the search space further, various selection heuristics are studied.

If term construction should be attempted for searching `form` in `arch`, one uses the command `BuildFunction`[`arch`, `form`]. We note that one can also view this as allowing λ-terms in the bindings for the unknown functions, but we prefer to see matters from a slightly different viewpoint: For the moment we assume that `form` is a quantified formula (without loss of generality, we may assume it in prenex normal form) containing an occurence of one or more unknown functions, for example $\forall_{x_1} \cdots \forall_{x_n} \Phi(F^?[x_1, \ldots, x_n])$.

The unknown function is replaced by a new existentially quantified variable v_F, with the existential quantifier inserted exactly after all quantifiers corresponding to $x_1, \ldots, x_k$; we obtain $\forall_{x_1} \cdots \forall_{x_n} \exists_{v_F} \Phi(v_F)$ for the example above. This formula is then transformed by introducing Skolem functions for the universally quantified variables (each Skolem function having as arguments the existential quantified variables occuring before the respective universal variable). Finally, the existential quantifiers are eliminated by introducing metavariables as in [13, §4], on which first-order unification against the archive `arch` is applied (after Skolemizing it in the usual way): In the succesful case, the unifier will contain bindings both for the universal variables in the assumptions of `arch` and for the metavariables in `form`.

Consider the following example occurring in a synthesis of the Gröbner basis algorithm [10]. Omitting restriction predicates like `is-power-product` and `is-polynomial` for the sake of simplicity, the crucial point in the synthesis requires proving

$$\underset{p}{\forall}\ \underset{g,h}{\forall}\ (\mathtt{lp}[g] \mid p \wedge \mathtt{lp}[h] \mid p \Rightarrow F^{?}[g,h] \mid p)$$

with the background archive typically containing a formula like:

$$\underset{p}{\forall}\ \underset{s,t}{\forall}\ (s \mid p \wedge t \mid p \Rightarrow \mathtt{lcm}[s,t] \mid p)$$

Here `lp` and `lcm` refer respectively to the leading power product of a polynomial and to the least common multiple of two power products. Applying the above procedure, the goal formula becomes $\mathtt{lp}[g_0] \mid p_0 \wedge \mathtt{lp}[h_0] \mid p_0 \Rightarrow v_F^* \mid p$, while the archive formula becomes $s^* \mid p^* \wedge t^* \mid p^* \Rightarrow \mathtt{lcm}[s^*, t^*] \mid p^*$. This yields the unifier:

$$\{p^* \leftarrow p_0, s^* \leftarrow \mathtt{lp}[g_0], t^* \leftarrow \mathtt{lp}[h_0], v_F^* \leftarrow \mathtt{lcm}[\mathtt{lc}[g_0], \mathtt{lc}[h_0]]\}.$$

Thus $F^{?}[g,h]$ has been found to be $\mathtt{lcm}[\mathtt{lc}[g], \mathtt{lc}[h]]$.

5.6. Theory Exploration in Archives

Theory exploration is a specific way of automated knowledge buildup, as opposed to the usual manual creation of archives by formalizing existing textbooks, articles, etc. In practice, both methods should be combined for optimal results.

In the current implementation, theory exploration is provided as an experimental feature based on [7] and [13, §2]. We distinguish two types of exploration:

- New propositions are searched by `InventPropositions[arch,lib,sym]`.
- New concepts are created by `InventConcepts[arch,lib,sym,new]`.

Here `arch` is the archive that should be expanded by the new propositions or definitions, `lib` is a library containing suitable invention schemes, `sym` a list of concepts from `arch` to be involved in the new propositions or definitions, and `new` a symbol to be defined. Note that `lib` is also realized as an archive, albeit with a rather special interpretation.

The algorithm for inventing propositions proceeds as follows: It picks from `lib` a proposition scheme, which is then instantiated by the symbols in `sym`. If the subsequent attempt at proving the resulting proposition fails, it is discarded and the algorithm tries another scheme; the succesful propositions are appended to `arch`.

Let us illustrate this by a simple example. A type of proposition occurring frequently in mathematics asserts that a binary operation F respects a binary relation R. In `lib`, this could show up as the following proposition scheme:

$$\underset{F}{\forall}\ \underset{R}{\forall}$$
$$\mathtt{Opn\text{-}Respects\text{-}Rel}[F,R] \Leftrightarrow \underset{a,b,c,d}{\forall}$$
$$R[a,b] \wedge R[c,d] \Rightarrow R[F[a,c], F[b,d]]$$

Assume now that `arch` is an archive describing the theory of natural numbers, with definitions and (very few) propositions about + and <. Then the invention algorithm, with `sym` containing + and <, will come up with the conjecture that `Opn-Respects-Rel`$[+, <]$, which is immediately expanded to the monotonicity of + with respect to <. If a sufficiently strong prover is available, this will be added as a proposition.

The invention of concepts works in a similar fashion, except that the definition schemes require as an additional argument the new symbol to be defined (and of course no proof is necessary since we restrict ourselves to explicit definitions). A typical example of a definition scheme occuring in `lib` is divide-and-conquer:

$$\underset{\texttt{is-spc}}{\forall}\ \underset{\texttt{left,right}}{\forall}\ \underset{\texttt{mrg}}{\forall}\ \underset{F}{\forall}$$

$$\texttt{Divide-Conquer}[\texttt{is-spc,left,right,mrg},F] \Leftrightarrow$$

$$\underset{x}{\forall} F[x] = \begin{cases} x & \Leftarrow \texttt{is-spc}[x] \\ \texttt{mrg}[F[\texttt{left}[x]], F[\texttt{right}[x]]] & \Leftarrow \texttt{True} \end{cases}$$

The scenario "Logical Algorithm Retrieval = Symbolic Computation Proving" of [9] can be described by a background archive `arch` on tuple theory, containing in particular the notions `is-trivial-tuple`, `left-split`, `right-split` and `merge`. Taking these for `sym` and `merge-sort` for `new`, we obtain a definition of the merge-sort algorithm.

In principle, one could combine this approach with the `BuildFunction` command of Subsection 5.5 for synthesizing the merge-sort algorithm as in the scenario "Logical Algorithm Retrieval = Algorithm Invention" of [9]. But a full integration of the synthesis method described in [12] and implemented in [16], is beyond the scope of the current implementation.

Using labels, theory exploration can also be carried out in a single archive `big-archive`. In this case, `arch` and `lib` are labels referring to suitable portions of `big-archive`. The propositions or definitions generated by the exploration will then be appended to `big-archive` under specific labels `NewPropositions` and `NewDefinitions`, respectively.

6. Conclusion

In this paper we have introduced the notions of labels and namespaces, which permit a structured representation of a mathematical knowledge base; we have called such a knowledge bases an archive. Based on the *Theorema* language, archives offer constructs for splitting formulae in multiple cells, with quantifiers ranging over whole cell groups and labels for attaching names to the groups. This makes archives very readable and particularly suited for large bodies of mathematical knowledge.

Symbols can be bound to a namespace, which is typically associated with the label attached to the cell group containing the symbols. Namespaces provide a

unified approach for two important issues: (1) domains as used for categories and functors, and (2) the intuitive usage of "contexts" for resolving ambiguos symbols.

A decisive feature of archives is that all its language constructs, in particular the symbols $\rightleftharpoons$ for attaching labels and : for declaring namespaces, are logic-internal—they extend *Theorema* in such a way that there is a natural translation back to plain *Theorema*.

We have also introduced various crucial operations on archives, notably mathematical knowledge retrieval and theory exploration. A lot of work is still necessary here. In fact, our present treatment of archives should only be seen as a first step in developing a logic-internal paradigm for MKM. We hope that such a treatment—combined with other (more logic-external) approaches—will shed light on some structural issues of MKM.

Appendix

Grammar of structured notebooks

A structured notebook is a list of *Mathematica* `RowBox` structures [30], which we denote in curly braces by $\{r_1, r_2, \ldots\}$ as for *Mathematica* lists. Thus the comma between r_1 and r_2 represents the "newline" separating the corresponding cells, with r_1 being the leader of the cell group containing $r_1, r_2, \ldots$. (The leader is the first cell of a cell group—it remains visible when the cell group is collapsed.)

We use a variant of EBNF grammar with the following extra conventions: $\{\ldots\}^*$ means repeating zero or more times, $\{\ldots\}^+$ one or more times, and $\{\ldots\}^-$ stands for an option (i.e. zero or one times).

We use the following syntactic categories: `SNB` for structured notebooks, `LFM` for labelled formulae, `UFM` for unlabelled formulae, `ETM` for extended terms, `NSP` for namespace declarations, `SUB` for substitutions, and `QNT` for quantifiers. The terminal symbols `Id`, `Tfm` and `Ttm` stand respectively for the legitimate identifiers in *Mathematica*, the formulae and terms in plain *Theorema* (i.e. contained in a single cell).

$$\texttt{SNB} ::= \{\texttt{Title}\,\{,\texttt{LFM}\}^+\}$$

$$\begin{aligned}\texttt{LFM} ::= {} & \{\{\bullet\}^-\ \texttt{Id}\,\{:\langle\{\texttt{Id}\}^+\rangle\}^- \rightleftharpoons \{\texttt{QNT}\}^*\,\{,\texttt{SUB}^+\}\,\{,\texttt{UFM}^+\}\} \mid \\ & \{\{\bullet\}^-\ \texttt{Id}\,\{:\langle\{\texttt{Id}\}^+\rangle\}^- \rightleftharpoons \{\texttt{LFM},\}^+\ \texttt{LFM}\} \mid \\ & \{\texttt{QNT}\}^*\,\{,\texttt{LFM}^+\} \mid \texttt{UFM}\end{aligned}$$

$$\begin{aligned}\texttt{UFM} ::= {} & \{\{\texttt{QNT}\}^*\,\{,\texttt{UFM}\}^+\} \mid \{\texttt{Tfm}\,(\Leftrightarrow \mid \Rightarrow)\,\{,\texttt{SUB}\}^-\,\{,\texttt{UFM}\}^+\} \mid \\ & \texttt{NSP} \mid \texttt{Ttm} =, \texttt{ETM} \mid \{\texttt{Tfm},\ \texttt{SUB}\} \mid \texttt{Tfm}\end{aligned}$$

$$\texttt{ETM} ::= \texttt{Ttm} \mid \{\{\underset{\texttt{Id}}{\ni}\}^+\{\texttt{QNT}\}^*\,\{,\texttt{UFM}\}^+\}$$

$$\texttt{NSP} ::= (\{\bullet\}^-\ \texttt{Id} \mid\ \texttt{Ttm}) : \langle\{\texttt{Id}\}^+\rangle$$

$$\texttt{SUB} ::= \{\nleftarrow, \{\texttt{Tfm}\}^+\}$$

$$\texttt{QNT} ::= (\forall \mid \exists)_{\texttt{Tfm}}^{\{\texttt{Id}\}^+}$$

Grammar of archives

An archive is a formula in the following extended *Theorema* language. We use the following syntactic categories: `ARH` for (sub)archives, `FRM` for unlabelled formulae, `ARV` for argument/value pairs, `LBL` for labels, and `QNT` for quantifiers. The terminal symbols are as for the previous grammar, plus the constructors $^{\texttt{TM}}\texttt{Conjunction}$, $^{\texttt{TM}}\texttt{Binding}$, $^{\texttt{TM}}\texttt{FiniteFunction}$ and $^{\texttt{TM}}\texttt{ArgVal}$.

$$\texttt{ARH} ::= \texttt{LBL} \rightleftharpoons \{\texttt{QNT}\}^* \ \texttt{ARH} \mid \texttt{LBL} \rightleftharpoons \{\texttt{QNT}\}^* \ {}^{\texttt{TM}}\texttt{Conjunction}[\{\texttt{ARH}\}^+] \mid \texttt{FRM}$$

$$\texttt{FRM} ::= \texttt{Tfm} \mid {}^{\texttt{TM}}\texttt{Binding}[\texttt{Ttm}, \{\texttt{Id}\}^+] \mid {}^{\texttt{TM}}\texttt{FiniteFunction}[\{\texttt{ARV}^+\}]$$

$$\texttt{ARV} ::= {}^{\texttt{TM}}\texttt{ArgVal[Id,Id]}$$

$$\texttt{LBL} ::= \texttt{Id} \mid \bullet\texttt{Id} \mid {}^{\texttt{TM}}\texttt{Binding}[\{\bullet\}^- \ \texttt{Id}, \{\texttt{Id}\}^+]$$

$$\texttt{QNT} ::= (\forall \mid \exists)_{\substack{\{\texttt{Id}\}^+ \\ \texttt{Tfm}}}$$

References

[1] K. Aboul-Hosn, *A Proof-Theoretic Approach to Mathematical Knowledge Management*, PhD Thesis, Cornell University, USA, 2007.

[2] K. Aboul-Hosn and T. Damhøj Andersen, *A Proof-Theoretic Approach to Hierarchical Math Library Organization*. In: M. Kohlhase (ed.), Proceedings of the 4th International Mathematical Knowledge Management Conference (MKM'05), pp. 1–16, Bremen, October 2005.

[3] A. Asperti, C. S. Coen, E. Tassi and S. Zacchiroli, *User Interaction with the Matita Proof Assistant*, Journal of Automated Reasoning, Special Issue on User Interfaces for Theorem Proving, vol. 39/2, pp. 109–139, 2007.

[4] A. Asperti, F. Guidi, C. S. Coen, E. Tassi, and S. Zacchiroli, *A Content Based Mathematical Search Engine: Whelp*. In: C. Paulin-Mohring and B. Werner (eds.), Proceedings of Types for Proofs and Programs International Workshop (TYPES'04), LNCS 3839, pp. 17–32, Springer, 2006.

[5] A. Asperti, L. Padovani, C. S. Coen and I. Schena, *HELM and the Semantic Math-Web*. In: R. J. Boulton and P. B. Jackson (eds.), Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'01), pp. 59–74, Edinburgh, Scotland, 2001.

[6] B. Buchberger, *Theory Exploration versus Theorem Proving*. In: A. Armando and T. Jebelean (eds.), Proceedings of the Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus'99), Electronic Notes in Theoretical Computer Science, volume 23/3, Elsevier, 1999.

[7] B. Buchberger, *Theory Exploration with Theorema*. In: T. Jebelean, V. Negru, A. Popovici (eds.), Selected papers of the Second International Workshop on Symbolic and Numeric Algorithms in Scientific Computing (SYNASC'00), Analele Universitatii din Timisoara, Seria Matematica-Informatica XXXVIII(2), pp. 9–32, Timisoara, Romania, 2000.

[8] B. Buchberger, *Groebner Rings and Modules*. In: S. Maruster, B. Buchberger, V. Negru and T.Jebelean (eds.), Proceedings of Third International Workshop on Symbolic and Numeric Algorithms in Scientific Computing (SYNASC'01), pp. 22–25, Timisoara, Romania, 2001.

[9] B. Buchberger, *Algorithm Retrieval: Concept Clarification and Case Study in Theorema*, SFB Report No. 2003-44, Johannes Kepler University Linz, Spezialforschungsbereich F013, October 2003.

[10] B. Buchberger, *Towards the Automated Synthesis of a Groebner Bases Algorithm*, RACSAM—Revista de la Real Academia de Ciencias (Review of the Spanish Royal Academy of Science), Serie A: Mathematicas 98(1), pp. 65–75, 2004.

[11] B. Buchberger, *Groebner Bases in Theorema Using Functors*. In: J.C. Faugere and D. Wang (eds.), Proceedings of the First International Conference on Symbolic Computation and Cryptography (SCC'08), pp. 1–15, Beihang University Press, Beijing, China, 28–30 April 2008.

[12] B. Buchberger and A. Craciun, *Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema*. In: F. Kamareddine (ed.), Proceedings of the Mathematical Knowledge Management Symposium, Electronic Notes in Theoretical Computer Science, vol. 93, pp. 24–59, Edinburgh, February 2004.

[13] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz and W. Windsteiger, *Theorema: Towards Computer-Aided Mathematical Theory Exploration*, Journal of Applied Logic 4(4), pp. 470–504, 2006.

[14] B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru and W. Windsteiger, *The Theorema Project: A Progress Report*, In: M. Kerber and M. Kohlhase (eds.), Symbolic Computation and Automated Reasoning, Proceedings of the Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus'00), pp. 98–113, A.K. Peters, 2000.

[15] C. S. Coen, *Knowledge Management of Formal Mathematics and Interactive Proving*, PhD Thesis, University of Bologna, Italy, 2004.

[16] A. Craciun, *Lazy Thinking Algorithm Synthesis in Groebner Bases Theory*, PhD Thesis, Johannes Kepler University, Linz, Austria, 2008. Available as: Technical Report No. 08-02, Research Institute for Symbolic Computation, Linz, Austria, 2008.

[17] M. Giese and B. Buchberger, *Towards Practical Reflection for Formal Mathematics*, extended abstract. In: T. Kutsia and M. Marin (eds.), Proceedings of the Austria-Japan Workshop on Symbolic Computation and Software Verification, pp. 30–34. Available as: Technical report no. 07-09, Research Institute for Symbolic Computation, Linz, Austria, July 2007.

[18] M. Kohlhase, *An Open Markup Format for Mathematical Documents* (Version 1.2), LNAI 4180, Springer, Heidelberg, 2006.

[19] M. Kohlhase and I. Sucan, *A Search Engine for Mathematical Formulae*. In: T.Ida, J.Calmet and D. Wang (eds.), Proceedings of Artificial Intelligence and Symbolic Computation (AISC'06), pp. 241–253, Springer, 2006.

[20] P. Librecht and E. Melis, *Methods for Access and Retrieval of Mathematical Content in ActiveMath*. In: N. Takayama, A. Iglesias and J. Gutierrez (eds.), Proceedings

of the Second International Congress on Mathematical Software (ICMS'06), LNCS 4151, Springer, 2006.

[21] S. MacLane and G. Birkhoff, *Algebra*, MacMillan, New York, 1968.

[22] E. Melis, J. Büdenbender, G. Goguadze, P. Libbrecht and C. Ulrich, *Knowledge Representation and Management in ActiveMath*, Annals of Mathematics and Artificial Intelligence, vol. 38, pp. 47–64, 2003.

[23] B. Miller and A. Youssef, *Technical Aspects of the Digital Library of Mathematical Functions*, Annals of Mathematics and Artificial Intelligence, vol. 38, pp. 121–136, Kluwer, 2003.

[24] F. Piroi, *Tools for Using Automated Provers in Mathematical Theory Exploration*, PhD Thesis, University of Linz, Austria, August 2004.

[25] F. Piroi and B. Buchberger, *An Environment for Building Mathematical Knowledge Libraries*. In: W. Windsteiger and C. Benzmüller (eds.), Proceedings of the Workshop on Computer-Supported Mathematical Theory Development, Second International Joint Conference on Automated Reasoning (IJCAR'04), pp. 19–29, Cork, Ireland, 4–8 July 2004.

[26] F. Piroi, B. Buchberger and C. Rosenkranz, *Mathematical Journals as Reasoning Agents: Literature Review*, Technical report no. 08-05, Research Institute for Symbolic Computation, Linz, Austria, March 2008.

[27] F. Piroi, B. Buchberger, C. Rosenkranz and T. Jebelean, *Organisational Tools for MKM in Theorema*, Technical report no. 07-11, Research Institute for Symbolic Computation, Linz, Austria, 2007.

[28] C. Rosenkranz, *Retrieval and Structuring of Large Mathematical Knowledge Bases: A Theorema Approach*, PhD Thesis, University of Linz, Austria, forthcoming.

[29] P. Rudnicki and A. Trybulec, *Mathematical Knowledge Management in Mizar*. In: B. Buchberger and O. Caprotti (eds.), Proceedings of the First International Workshop on Mathematical Knowledge Management (MKM'01), Hagenberg, September 2001.

[30] S. Wolfram, *The Mathematica Book*, Wolfram Media Inc., 5th edition, 2003.

Camelia Rosenkranz
e-mail: `Camelia.Rosenkranz@risc.uni-linz.ac.at`
Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria.

Bruno Buchberger
e-mail: `Bruno.Buchberger@risc.uni-linz.ac.at`
Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria.

Tudor Jebelean
e-mail: `Tudor.Jebelean@risc.uni-linz.ac.at`
Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria.