Chapter 3
Towards a Self-reflective, Context-aware Semantic Representation of Mathematical Specifications

Peter Schodl and Arnold Neumaier and Kevin Kofler and Ferenc Domes and Hermann Schichl

Abstract We discuss a framework for the representation and processing of mathematics developed within and for the MOSMATH project. The MOSMATH project aims to create a software system that is able to translate optimization problems from an almost natural language to the algebraic modeling language AMPL. As part of a greater vision (the FMathL project), this framework is designed both to serve the optimization-oriented MOSMATH project, and to provide a basis for the much more general FMathL project.
We introduce the semantic memory, a data structure to represent semantic information, a type system to define and assign types to data, and the semantic virtual machine (SVM), a low level, Turing-complete programming system that processes data represented in the semantic memory.
Two features that set our approach apart from other frameworks are the possibility to reflect every major part of the system within the system itself, and the emphasis on the context-awareness of mathematics.
Arguments are given why this framework appears to be well suited for the representation and processing of arbitrary mathematics. It is discussed which mathematical content the framework is currently able to represent and interface.

Acknowledgments.

Support by the Austrian Science Fund (FWF) under contract number P20631 is gratefully acknowledged.

3.1 The MOSMATH project

The project "a modeling system for mathematics" (MOSMATH), currently carried out at the University of Vienna, aims to create a modeling system for the specification of models for the numerical work in optimization in a form that is natural for the working mathematician. The specified model is represented

Peter Schodl
Fakultät für Mathematik, Universität Wien, Nordbergstr. 15, A-1090 Wien, Austria, e-mail: peter.schodl@univie.ac.at

Arnold Neumaier
Fakultät für Mathematik, Universität Wien, Nordbergstr. 15, A-1090 Wien, Austria, e-mail: arnold.neumaier@univie.ac.at

Kevin Kofler
Fakultät für Mathematik, Universität Wien, Nordbergstr. 15, A-1090 Wien, Austria, e-mail: kevin.kofler@chello.at

Ferenc Domes
Fakultät für Mathematik, Universität Wien, Nordbergstr. 15, A-1090 Wien, Austria, e-mail: ferenc.domes@univie.ac.at

Hermann Schichl
Fakultät für Mathematik, Universität Wien, Nordbergstr. 15, A-1090 Wien, Austria, e-mail: hermann.schichl@univie.ac.at

and processed inside a framework and can then be communicated to numerical solvers or other systems. While the input format is a controlled natural language (just like Naproche [6] and MathNat [9], but with a different target), it is designed to be as expressive and natural as currently feasible. This paper summarizes the work done in our group, with emphasis on the content of the PhD thesis of the first author.

The user benefits from this input format in multiple ways: The most obvious advantage is that a user is not forced to learn an algebraic modeling language and can use the usual natural mathematical language, which is learned and practiced by every mathematician, computer scientist, physicist, and engineer.

In addition, this kind of specification of a model is the least error prone, and the most natural way to communicate a model. Once represented in the framework, multiple outputs in different modeling languages (or even descriptions in different natural languages) would not mean extra work for the user if appropriate transformation modules are available.

The MOSMATH project makes use of or connects to several already existing software systems:

LaTeX: Being the de facto standard in the mathematical community for decades, the syntax of the input will be a subset of LaTeX.

Markup languages: Texts written in markup languages like XML are highly structured and easily machine readable, e.g., XML employs a tree structure represented in text form. We make use of the tool LaTeXML [33] to produce an XML document from a LaTeX input, which can then be translated into records in our data structure.

Algebraic modeling languages: To be able to access a wide variety of solvers, the algebraic modeling language AMPL [7] is used the primary target language. One of the reasons for this choice is existing software that converts AMPL to other modeling languages.

The Grammatical Framework [25]: A programming language for multilingual grammar applications, which allows us to produce grammatically correct sentences in multiple languages.

Naproche [6]: An interface from a controlled natural language to proof checking software, which can be used to interface proof checkers.

TPTP [34]: The library “Thousands of problems for theorem provers” provides facilities to interface interactive theorem provers.

The thesis of Peter SCHODL [26] will comprise the main topics of this paper in greater rigor and detail.

3.2 Vision: The FMathL project

The MOSMATH project is embedded into a far more ambitious long-term vision – the FMathL project, described in the extensive FMathL web site[1] (for a summary, see [23]).

While the MOSMATH project creates an interface for optimization problems formulated in almost natural mathematical language, the vision of the FMathL project is an automatic general purpose mathematical research system that combines the indefatigability, accuracy and speed of a computer with the ability to interact at the level of a good mathematics student. Formal models would be specified in FMathL close to how they would be communicated informally when describing them in a lecture or paper: with functions, sets, operators, measures, quantifiers, tables, cases rather than loops, indices, diagrams, etc.

FMathL aims at providing mathematical content and proof services as easily as Google provides web services, Matlab provides numerical services, and Mathematica or Maple provide symbolic services. A mathematical assistant based on the FMathL framework should be able to solve standard exercises, intelligently search a universal database of mathematical knowledge, check the represented mathematics for correctness, and aid the author in routine mathematical work. The only extra work the user would have to do is during parse time of the written document, when possible ambiguities have to be resolved.

Important planned features of FMathL include the following:

[1] The FMathL web site is available at `http://www.mat.univie.ac.at/~neum/FMathL.html`

- It has both a "workbench" character where people store their work locally, as well as a "wiki" character where work can be shared worldwide.
- It *supports* full formalization, but does not *force* it upon the user.
- It incorporates techniques from artificial intelligence.
- It communicates with the user in a natural way.
- The language is extensible, notions can be defined as usual and will then be understood.
- To deal with ambiguities, the system makes use of contextual information.

By complementing existing approaches to mathematical knowledge management, the FMathL project will contribute towards the development of:

- The QED project [3]: FMathL would come with a database of basic mathematics, preferably completely formalized. In addition, the natural interface would make contributing to the QED project easier.
- A universal mathematical database: envisioned, e.g., in ANDREWS [1] and partially realized in theorem provers with a big library (such as MIZAR [35]), where they only serve a single purpose.
- An assistant in the sense of WALSH [36] that saves the researcher's time and takes routine off their shoulders – in the classroom, in research, and in industry.
- A checker not only for grammar but also for the semantical correctness of mathematical text.
- Automatic translation of mathematical content into various natural languages.

While FMathL reaches far beyond MOSMATH, we expect that the framework of the MOSMATH project will serve as a first step towards the FMathL project. The FMathL project will also benefit from MOSMATH in the sense that once MOSMATH is integrated into FMathL, it will make FMathL usable in the restricted domain of optimization long before the full capabilities of FMathL are reached.

3.3 Data structure: The semantic memory

The semantic memory is a framework designed for the representation of arbitrary mathematical content, based on a computer-oriented setting of formal concept analysis (GANTER & WILLE [8]). In particular, our goal was to be able to represent mathematical expressions, mathematical natural language, and grammars in a natural way in the semantic memory. (We are aware of existing languages and software systems to represent mathematics, but found them inadequate for our goals, see [14], [15].)

We assume an unrestricted set of **objects** . Objects may, but need not have **names**, i.e., alphanumeric strings not beginning with a digit, by which the user refers to an object. `Empty` is the name of an object. Furthermore objects may, but need not have **external values** , i.e., data of arbitrary form, associated with the object, but stored outside the semantic memory.

Variable objects are variables in the usual sense, ranging over the set of objects, but since alphanumeric strings may refer to objects, we refer to variable objects via a string beginning with a hash (#) followed by some alphanumeric string. Usually, we will use suggestive strings for variables, e.g., for an object that is intended to be a handle, we use `#handle` or `#h`.

A **semantic mapping** assigns to two objects `#h` and `#f` a unique object `#e` which is `Empty` if `#h` or `#f` is `Empty`. We call an equation of the form `#h.#f=#e` with `#h`, `#f`, and `#e` not `Empty` a **semantic unit** or **sem** . We call `#h` the **handle**, `#f` the **field**, and `#e` the **entry** of the sem.

The semantic unit is the smallest piece of information in the semantic mapping. It is intended to be both low level and natural to the human. Depending on the context, the intuitive meaning of a sem `#h.#f=#e` often is "the `#f` of `#h` is `#e`", e.g., `formula27.label=CauchySchwarz` would intuitively mean that the `label` of `formula27` is `CauchySchwarz`. A sem can also express a unary predicate: $P(x)$ could be expressed as `P.x=True`. Since sems are "readable" in this intuitive sense, manipulating information in

the SM is easier and more transparent than in other low level systems like a Turing machine or a register machine.

The SM codifies the foundations of formal concept analysis in a way suitable for automatic storage and processing of complex records. A statement of the form *gIm* (interpreted as "the object *g* has the attribute *m*") can be represented as a sem `g.m=Present`. The semantic matrix precisely matches *multi valued contexts* (GANTER & WILLE [8, p.36]) where I is a ternary relation and $I(g,m,w)$ is interpreted as "the attribute *m* of object *g* is *w*", with the property $I(g,m,w_1)$ and $I(g,m,w_2)$ then $w_1 = w_2$. This corresponds to the sem `g.m=w`, since the property `g.m=w1` and `g.m=w2` then `w1=w2` follows from the uniqueness of the entry of a semantic mapping.

The semantic memory is also representable within the framework of the semantic web [17]. In particular, we have implemented it in RDF [18].

For graphical illustration of a semantic mapping, we interpret a sem `#a.#b=#c` as an edge with label `#b` from node `#a` to node `#c` of a directed labeled graph, called a **semantic graph**. In semantic graphs, named objects are represented by their names, and objects that have no name will be represented by automatically generated strings preceded by a dollar sign (`$`). In a semantic graph, objects that have an external value are printed as a box containing that value. For better readability we use dashed edges for edges labeled with `type`, since these sems have importance for typing.

A **record** with handle `#rec` is the set of sems `#h.#f=e` reachable from `#rec` in the sense that there exist objects `#f1`,...,`#f`*n* such that `#rec.#f1.`$\cdots$`.#f`*n*`=#h`. In contrast to records in programming languages such as Pascal, records in a SM may have cycles since back references are allowed; this is necessary to encode grammars in a natural way.

For example, the expression

$$\lambda x.x+1$$

may be represented by the record with handle `$2472` shown in Figure 3.1. Further examples can be found in Appendix 3.9, see also [28].

Semantic mappings are used to store mathematics. In order to work on data represented, the framework needs to be dynamic. The data structure of the semantic mapping that changes over time is called the **semantic memory** (SM).

While not identical, our representation shares features with some existing representation frameworks:

- The need to represent (mathematical) natural language poses the requirement of "structure sharing", i.e., a phrase, an expression etc. only has to be represented once while it may occur multiple times in the text. This suggests a graph structure rather than a tree structure, as facilitated in the knowledge representation system SNePS [32]. SNePS also makes use of a labeled graph, but on the other hand uses "structured variables", storing quantification and constraints together with the variable. This is not desirable when representing mathematics since structured variables make it hard to represent the difference between, e.g.,
$$\forall x \forall y(P(x,y) \Longrightarrow G(x)) \qquad \text{and} \qquad \forall x((\forall y P(x,y)) \Longrightarrow G(x)).$$
- The record structure where a complex record is built up from combining more elemental records is similar to a parse tree, especially to a parse tree for a dependency grammar [5]. However, a parse tree is always a tree and does not allow structure sharing.

Context dependence.

Even on the lowest level of information representation, context dependence is already incorporated in a natural way.

The same object `#o` may be reachable via different paths, and while it is the same object, the access is different and so can be its interpretation. Therefore the path how an object is accessed contains information about the context, and about the usage of the object. The additional information contained in the path can

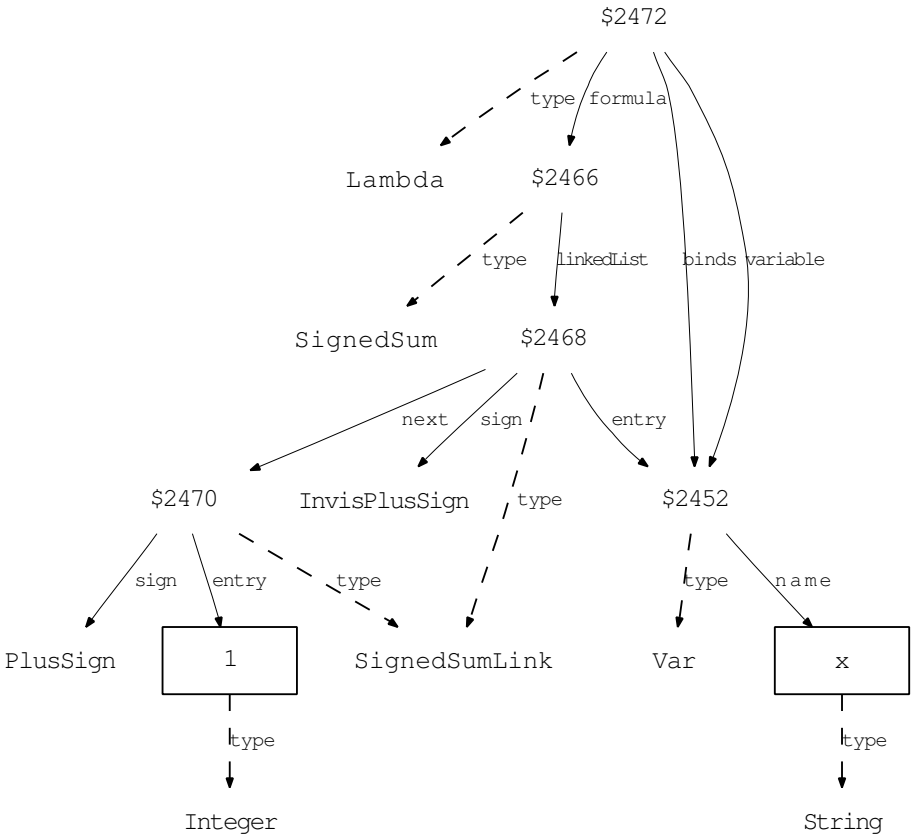


Fig. 3.1 A record representing an expression

be used to account for some of the contextual information in natural languages: while natural languages are to a large extent context free (JURAFSKY et al. [11]), the semantics is also based on the context.

A logic of context compatible with the present framework is presented in [22].

3.4 Processing: The semantic virtual machine

We define a virtual machine that operates on the SM called the **semantic virtual machine** (SVM) to be able to rigorously argue about processes in the SM, and to be able to proof properties. This abstract machine can be implemented in many ways; we currently have implementations in Matlab (using a sparse matrix to represent the SM) and in C++/Soprano (using RDF).

The semantic memory of the SVM contains a **program** to execute, its **context** (i.e., input and output, corresponding to the tape of an ordinary Turing machine), and the information about **flow control** as well. To enable the processing of more than one program in the same memory each program has its own **core**, i.e., a record reserved for temporary data. Since the core is the most important record for a program we use the **caret** ^ to abbreviate the reference the core of the program. Hence `^a` means `#core.a`, where `#core` is the core of the program under consideration.

The most elementary part of the SVM programming language is a command. A **process** is a sequence of commands, beginning with the command `process #proc`. Every process ends with a command that either halts the SVM or calls another process. An **SVM program** is the command `program #prog`

followed by a sequence of processes. The (changing) SVM command currently executed is called the **focus**. Each SVM program is completely represented in the SM. The SVM is given a record containing a program, an object it can use as a core, and can then be started.

The SVM has the ability to access the facilities of the physical device it is implemented on. This may provide the SVM with much better performance for tasks it can export, and allows the use of trusted external processors and programs in different programming languages.

External values can be copied to the memory of the SVM, and conversely. This is done by the commands `in` and `out`. The information about how to represent the external value in the memory of the SVM is called the **protocol**, and is used as an argument for the commands `in` and `out`. Our current implementation includes protocols for representing natural numbers, text, SVM programs and tapes and transition tables for Turing machines. There may be an arbitrary number of protocols, as long as the device on which the SVM is implemented knows how to interpret them.

The commands of the SVM language fall into four groups: Table 3.1 describes the commands that are needed to give the program an appropriate structure. Table 3.2 contains the assignments, i.e., those commands that perform alterations in the SM. If an assignment refers to a nonexisting node, an error is produced. Table 3.3 gives the commands used for flow control, and Table 3.4 the commands that establish communication with the physical device, namely call external processes and access external values. For convenience, some commands also appear in variants that could be simulated by a sequence of other commands.

In the informal description of commands given here, `VALUE(#node)` refers to the external value of the node `#node`. The effect of each command is formally defined by an operational semantics given in [26].

SVM command	comment
`program #1`	first line of the program `#1`
`process #1`	first line of the process `#1`
`start #1`	start with process `#1`

Table 3.1 Structuring commands

SVM command	comment
`^#1=(^#2==^#3)`	sets `^#1` to `True` if `^#2` = `^#3`, else to `False`
`^#1.#2=^#3`	assigns `^#3` to `^#1.#2`
`^#1.^#2=^#3`	assigns `^#3` to `^#1.^#2`
`^#1.#2=const #3`	assigns `#3` to `^#1.#2`
`^#1=^#2.#3`	assigns `^#2.#3` to `^#1`
`^#1=^#2.^#3`	assigns `^#2.^#3` to `^#1`
`^#1=fields of ^#2`	assigns the used fields of the record `^#2` to `^#1.1`, `^#1.2`,...
`^#1=exist(#2.#3)`	sets `^#1` to `True` if `#1.#2` exists, else to `False`
`^#1=exist(^#2.^#3)`	sets `^#1` to `True` if `^#1.^#2` exists, else to `False`

Table 3.2 Assignment commands

SVM command	comment
`goto #1`	sets the focus to the first line of process `#1`
`goto ^#1`	sets the focus to the first line of process `^#1`
`if ^#1 goto #2`	sets the focus to the first line of process `#2` if `^#1=True`, and to the next line if `^#1=False`
`stop`	ends a program

Table 3.3 Commands for flow control

Details of a (now obsolete) preliminary version of the SVM (called there “STM”) are discussed in [24].

As a test case and to show Turing completeness, we programmed a Turing machine in SVM. The SVM program that simulates a Turing machine contains 71 commands and can be found in [24]. Since an ordinary

SVM command	comment
`external #1(^#2)`	starts execution of the external processor `#1` with context `^#2`
`external ^#1(^#2)`	starts execution of the external processor `^#1` with context `^#2`
`in ^#1 as #2`	imports `VALUE(^#1)` into a record with handle `^#1` using protocol `#2`
`in ^#1 as ^#2`	imports `VALUE(^#1)` into a record with handle `^#1` using protocol `^#2`
`out ^#1 as #2`	exports the record with handle `^#1` into `VALUE(^#1)` using protocol `#2`
`out ^#1 as ^#2`	exports the record with handle `^#1` into `VALUE(^#1)` using protocol `^#2`

Table 3.4 Commands for external communication

Turing machine has no external storage and it is not specified how an external processor should behave, it is impossible to give an ordinary Turing machine that simulates an arbitrary SVM program.

3.5 Representation: The type system

An essential step that brings formal structure into the semantic memory is the introduction of **types**. In order to represent mathematics specified in a controlled natural language, a concept of typing is needed that is more general than traditional type systems. It must cover and check for well-formedness of both structured records as commonly used in programming languages and structures built from linguistic grammars. In particular each grammatical category must be representable as a type, in order to provide a good basis for the semantical analysis of mathematical language.

Information in the SM is organized in records. When using a record, or passing it to some algorithm, we need information about the structure of this record. Since we do not want to examine the whole graph every time, we assign types, both to objects and to sems. For a detailed discussion of the concepts and rigorous definition, see [29], illustrations and examples are given in [28].

Types can be defined using plain text documents called **type sheets**. An example of a type sheet can be found in [30]. Tables 3.5 and 3.6 gives an overview of the operators in a type sheet and their usage. For many tasks, giving an (annotated) type sheet defines the syntax of an arbitrary construction in the SM, and in many cases it even suffices to define the semantics.

operator	arguments	usage
`nothing`	none	defines an atomic type
`union`	list of names	defines a union
`atomic`	list of names	defines a union of atomic types
`complete`	none	closes a union

Table 3.5 Keywords in declarations of unions and atomics

Our form of type inheritance adds the specifications from an existing declared type `#T` to a newly defined declared type `#t`. In this case, we call `#T` the **template** of `#t`. All the requirements from `#T` apply to `#t`, and additionally the requirements for `#t`. The difference to the inheritance in typical programming languages is that:

- the declared type and the template may pose requirements on the same constituent,
- inheritance from multiple declared types is possible, but there is always only at most one template of a declared type, and
- it is stored that the template of `#D` is `#T`.

The type is always assigned with respect to a specified type system. A **type system** specifies the **categories** belonging to it, and their properties. The object `Empty` is never a category. In every type system, the set of categories is ordered by an irreflexive and transitive partial order relation `<`. If `#C1<#C2`, we say that `#C1`

operator	arguments	usage
`allOf`	list of equations	restricts entry of certain fields
`oneOf`	list of equations	restricts entry of certain fields
`someOf`	list of equations	restricts entry of certain fields
`optional`	list of equations	restricts entry of certain fields
`fixed`	list of equations	restricts entry of certain fields
`only`	list of equations	restricts entry of certain fields
`someOfType`	list of equations	restricts entry of certain fields
`itself`	list of names	restricts entry of certain fields
`array`	list of equations	restricts entry of certain fields
`index`	list of equations	requires to index each instance
`template`	one name	assigns a template
`nothingElse`	none	forbids further fields

Table 3.6 Keywords in declarations of proper types

contains `#C2` in this type system. We write $\leq$ for the associated reflexive partial order, with `#C1`$\leq$`#C2` iff `#C1<#C2` or `#C1=#C2`.

A category is called a **type** if it is minimal in the ordering `<`, and a **union** otherwise. Types come in three forms: the **default type** `Object`, **atomic types**, and **proper types**. All categories except `Object` are declared in the record whose handle is the type system. This record can be created by means of a type sheet, a piece of text containing the specifications.

If `#a.#f=Empty` for every field `#f`, then `#a` is an **atomic object**. Atomic objects are used as objects with a fixed semantic meaning. Objects of a proper type always have a field `type` whose entry is this type. Proper types are used to pose requirements on the other constituents of the objects of this type.

We define the **type of object** `#obj`:

If `#obj.type` is a declared type, then the type of `#obj` is `#obj.type`.
If `#obj.type=Empty` and `#obj` is an atomic type, then the type of `#obj` is `#obj`.
Otherwise, the type of `#obj` is `Object`.

Hence every object has a type. Determining the type of an object has complexity Ø(1).

An object `#obj` **matches** a category `#C` in the type system `#TS` if either `#t<#C` (in the type system `#TS`) where `#t` is the type of `#obj`, or `#C=Object`.

The definition of well-typedness of a record is more intricate and will only be sketched here.

A pair (`#o`,`#f`) of two objects is called a **position**. A type sheet poses requirements on certain positions, called **declared positions**. Roughly speaking, if for all declared positions (`#o`,`#f`) in a record, the object `#o.#f` matches the required type, then the record is **well-typed**, and **ill-typed** otherwise. Whether or not the record is well-typed can be checked in time linear in the size of the record.

We represent type declarations and unions as records, in order that a type checker, working on the semantic memory, has all the information it needs in the semantic memory and does not need any external type sheet. Thus we store a type declaration as a well-typed record in the SM. A type declaration `#TD` has `#TD.type=Type`.

Typing in FMathL and typing in XML bear significant similarities, most notably with DTD and Relax NG. (For a description of DTD, Relax NG and other XML schemas, see LEE & CHU [16].) Some of the operators in type declarations have a direct correspondence in the language of DTD and the RelaxNG compact syntax. E.g., `?` in DTD corresponds to `optional`, the pipe `|` corresponds to `oneOf` and parentheses `( )` correspond to `allOf`. A valid XML document corresponds to a well-typed record. However, there are also important differences, since cycles are an important feature of our framework that enables an efficient representation of concrete and abstract grammars, while XML documents are always organized in trees.

3.6 Higher level processing: CONCISE

The SVM is merely intended for definition, checkability and low level implementation. After some experience with SVM programs we designed a programming environment that is intended for user-friendly data entry and manipulation, algorithm design and execution, and more general for interaction with the semantic memory. Loosely speaking, it is an integrated development environment (IDE) for mathematics in the semantic memory. This environment is called CONCISE.

A Java implementation of CONCISE is being written by Ferenc DOMES and publication will be announced at the FMathL web site[2]. It consists of a versatile GUI (graphical user interface) that enables the user to view, create and manipulate sems and records in a natural way. An interpreter for a Turing complete subset of CONCISE written in SVM only requires 330 lines of SVM code.

Algorithms can be programmed and executed in CONCISE, but algorithms are represented as records in the SM, making CONCISE a text-free programming environment. Nevertheless, for debugging and alternative coding there are text views on CONCISE programs.

CONCISE has configurable display and text completion, and will support types, function calls, different kinds of variables (global, local, static and persistent), loops over all fields of an object, multiple users and multiple languages.

CONCISE will also incorporate a parser capable of dealing with dynamically changing grammars. This is necessary because in many specifications in ordinary mathematical language, the syntax (and hence the grammar) is partially defined through the context. In particular, definitions give not only the semantics of the term being defined, but implicitly also its grammatical function.

3.7 Mathematical modeling

The semantic memory is designed for representing and processing mathematical content. While generality of the representation was one important goal, another one was to be able to run algorithms on the records in a transparent way.

To test the practicability of the present framework, mathematical content from different sources is represented in the semantic memory:

- A significant fraction of the optimization problems from the OR Library [2] were represented manually in the semantic memory. This is the most important application for the MOSMATH project. We designed a natural representation of these optimization problems as records, in order to be able to run algorithms on these records. The representation is defined in a type sheet [30]. There are algorithms that produce LaTeX from formulas and whole optimization problems. Another algorithm enriches the representation of optimization problems in the semantic memory such that an AMPL document specifying a valid, numerically solvable model can be produced.
- Different types of mathematical formulas were extracted from lecture notes about basic analysis and linear algebra [20]. These were manually fed into the semantic memory to assure generality of the representation of formulas [28]. Partial work on the grammar of the text part of the lecture notes can be found in [27]. Some of the expressions from the lecture notes are presented in Section 3.9.
- An interface was written to automatically import formulas from the TPTP library [34] (“Thousands of Problems for Theorem Provers”, a library of formulas for theorem provers, taken from different branches of mathematics).
- An interface was written to automatically import formalized proofs written in the controlled natural language of Naproche [6] (“Natural Language Proof Checking”).

[2] http://www.mat.univie.ac.at/~neum/FMathL.html

Grammatical issues in the translation from mathematical language into SM documents, and from SM records to natural language, including a dynamic parser for parallel multiple context free grammars (PMCFGs) and an interface to the "Grammatical Framework" (GF) [25] are the subject of the PhD thesis by Kevin KOFLER [13]. This parser will handle updates to the grammar, a feature necessary to handle mathematical definitions that introduce new syntax.

Example: Knapsack problem

The multidimensional knapsack problem is a standard optimization problem, and also contained in the OR-Library [2]. A possible natural formulation is:

Let the integer N be the number of contracts, let the integer M be the number of budgets. Let c_j be the contract volume of project j for $j = 1,\ldots,N$, let $A_{i,j}$ be the estimated cost of budget i for project j for $i = 1,\ldots,M$ and $j = 1,\ldots,N$, and let B_i be the available amount of budget i for $i = 1,\ldots,M$. For $j = 1,\ldots,N$, let $x_j = 1$ if project j is selected, and let $x_j = 0$ otherwise.
Problem : Given integers N and M, vector c, matrix A and vector B, find the binary vector x such that

$$\sum_{j=1}^{N} c_j x_j$$

is maximal under the constraint $\sum_{j=1}^{N} A_{i,j} x_j \leq B_i$ for $i = 1,\ldots,M$.

When represented in the semantic memory, a transformation module produces a LaTeX-document that is identical to the model description above, except for grammatical errors. Another transformation module produces from the same record in the semantic matrix the following AMPL model.

```
param N ;
param M ;
param c{j in 1..N} ;
param A{i in 1..M , j in 1..N} ;
param B{i in 1..M} ;
var x{j in 1..N} binary ;

maximize target : sum{j in 1..N}(c[j] * x[j]);
subject to constraint_3014{i in 1..M} : sum{j in
1..N}(A[i , j] * x[j]) <= B[i];
```

3.8 Reflection

In the most general sense, reflection is the representation of parts of a system within itself.

To test the power of our setting, we tried to achieve a multiple reflection of various features on levels as low as possible:

- For the SVM, reflection is giving an SVM program that can simulate every other SVM program in the same way a universal Turing machine simulates an ordinary Turing machine. We call this program the **universal semantic virtual machine** (USVM). The context of the USVM contains the SVM program P and the context of P. When the USVM has finished, the USVM has produced the same changes in the context as P would have produced when called directly. The USVM is a program short and transparent enough to be checked by hand. It has only 164 commands; compare this, e.g., to the reflective interpreter

by JEFFERSON & FRIEDMAN in [10], which has 273 lines and the implementation by MCCARTHY [19] of a Lisp function evaluation in Lisp which has 58 (but very complex) commands. The USVM serves two purposes: it is a reflection of the actions of the SVM commands, and it gives us a check on many aspects of the SVM for correctness. Once one has convinced oneself of the correctness of the USVM, one can make the implementation of the SVM on some physical device also trustworthy by checking empirically (or, in principle, in a formal way) that any SVM program executed by the implemented SVM produces the same output as in the case when the USVM simulates this program.

- The type definitions are themselves typed, i.e., the creation of a type sheet that allows to type the representation of types in the semantic matrix, which gives the type of types. The type sheet for types [31] has only 40 lines, compared to the RelaxNG schema [4], its analogon in RelaxNG, which has over 300 lines, or Meta-DSD [12], its analogon in DSD, which has over 700 lines.
- The type system of CONCISE can be formulated as a type sheet, and the typechecker itself can be implemented as a CONCISE program.
- An interpreter of the programming language of CONCISE as an SVM program. This makes a large part of CONCISE independent of any specific implementation, since merely the SVM needs to be implemented.
- A definition of the semantics of mathematical concepts, represented within the SM.
- An operational semantics for the SVM, i.e., a rigorous description of the action of the SVM in mathematical terms, to be represented later in a reflective fashion on the concept level of CONCISE.
- For the envisioned FMathL, reflection is a process to deal with the distinction between object level and metalevel which occurs in every foundation of mathematics, hence also in FMathL. Here reflection means the process to express the notions of the metalevel as objects on the object level, and then formulate the relations between these notions as relations between objects. The same process of reflection of the metalevel L_0 in the object level L_1 can be repeated as a reflection of level L_k into the level L_{k+1}. This way, the metalevel is never fully formalized, but if the reflection step L_k in L_{k+1} can be accomplished without running into paradoxes or insufficiencies of the language, we can be confident of the adequacy of our foundation. For more details, and for the significance of reflection for FMathL as a foundation of mathematics, see [21].

3.9 Examples of expressions

Here we give a few examples for the representation of expressions in the SM as records (see [28] for many more).

The following table gives some statistics of the examples.

Example	# visible symbols	# LaTeX-characters	# sems
1	14	49	34
2	22	45	53
3	11	92	48

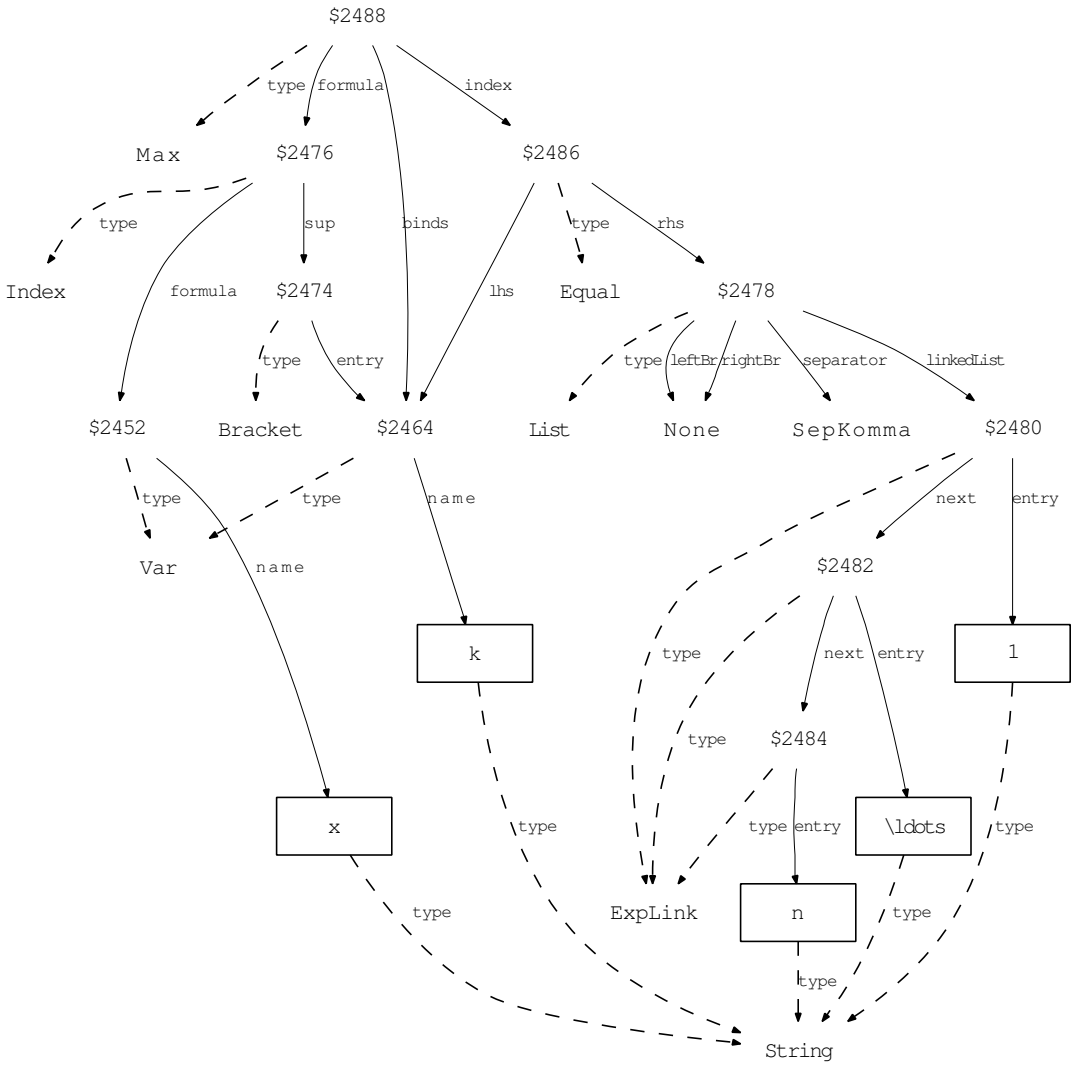
For each example, we give the rendered expression followed by a table containing the sems and external values in the left column, and the MATLAB construction code used to write the sems into our MATLAB implementation of the semantic memory in the right column. This is followed by a semantic graph of the record.

Example 0.

$$\max_{k=1,\ldots,n} x^{(k)}$$

```
\max_{k{=} 1 , \ldots , n }{{}{x}^{\left(k\right)}}
```

record	MATLAB construction code
$2488.type=max	x=createvar('x');
$2488.formula=$2476	k=createvar('k');
$2488.binds=$2464	listk=create('VarList',{k});
$2488.index=$2486	n=createstring('n');
$2464.type=Var	one=createstring('1');
$2464.name=$2466	dots=createstring('\ldots');
$2466.type=String	br=create('Bracket',k);
$2476.type=Index	in=create('Script',x,[],br);
$2476.formula=$2452	komma=createname('SepKomma');
$2476.sup=$2474	none=createname('None');
$2452.type=Var	lst=create('List',{one,dots,n},none,komma,none);
$2452.name=$2454	eq=create('Equal',k,lst);
$2454.type=String	ex=create('Max',in,listk,eq);
$2474.type=Bracket	
$2474.entry=$2464	
$2486.type=Equal	
$2486.lhs=$2464	
$2486.rhs=$2478	
$2478.type=List	
$2478.leftBr=None	
$2478.separator=SepKomma	
$2478.rightBr=None	
$2478.linkedList=$2480	
$2480.type=ExpLink	
$2480.next=$2482	
$2480.entry=$2470	
$2470.type=String	
$2482.type=ExpLink	
$2482.next=$2484	
$2482.entry=$2472	
$2472.type=String	
$2484.type=ExpLink	
$2484.entry=$2468	
$2468.type=String	
VALUE($2454) = x	
VALUE($2466) = k	
VALUE($2468) = n	
VALUE($2470) = 1	
VALUE($2472) = \ldots	

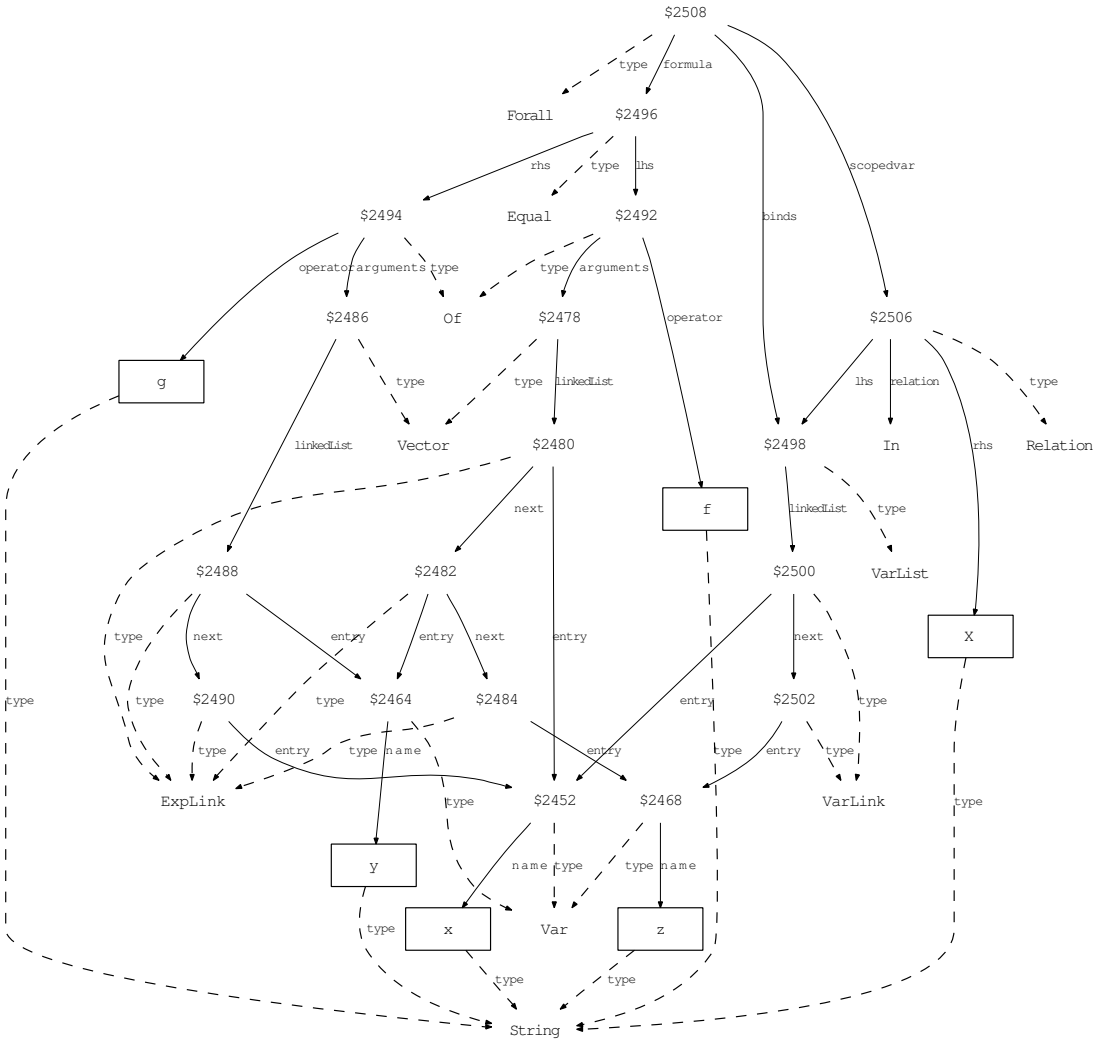

$2488
type
formula
index
Max
$2476
$2486
type
sup
binds
type
rhs
Index
formula
$2474
lhs
Equal
$2478
type
entry
type
leftBr
rightBr
separator
linkedList
$2452
Bracket
$2464
List
None
SepKomma
$2480
type
type
name
next
entry
Var
name
$2482
k
type
next
entry
1
type
$2484
x
type
type
entry
\ldots
type
type
ExpLink
n
type
type
String

Example 1.

$$\forall x,z \in X : f\left(x,y,z\right) = g\left(y,x\right)$$

```
\forall x , z \in X : f \left(x , y , z\right) {=} g \left(y , x\right)
```

record	MATLAB construction code
$2508.type=Forall	x=createvar('x');
$2508.formula=$2496	y=createvar('y');
$2508.binds=$2498	z=createvar('z');
$2508.scopedvar=$2506	f=createstring('f');
$2496.type=Equal	g=createstring('g');
$2496.lhs=$2492	X=createstring('X');
$2496.rhs=$2494	xyz=create('Vector',{x,y,z});
$2492.type=Of	yx=create('Vector',{y,x});
$2492.operator=$2472	lhs=create('Of',f,xyz);
$2492.arguments=$2478	rhs=create('Of',g,yx);
$2472.type=String	eq=create('Equal',lhs,rhs);
$2478.type=Vector	xz = create('VarList',{x,z});
$2478.linkedList=$2480	in=createname('In');
$2480.type=ExpLink	dummy=create('Dummy',x);
$2480.next=$2482	var=create('Relation',xz,in,X);
$2480.entry=$2452	ex=create('Forall',eq,var,xz);
$2452.type=Var	
$2452.name=$2454	
$2454.type=String	
$2482.type=ExpLink	
$2482.next=$2484	
$2482.entry=$2464	
$2464.type=Var	
$2464.name=$2466	
$2466.type=String	
$2484.type=ExpLink	
$2484.entry=$2468	
$2468.type=Var	
$2468.name=$2470	
$2470.type=String	
$2494.type=Of	
$2494.operator=$2474	
$2494.arguments=$2486	
$2474.type=String	
$2486.type=Vector	
$2486.linkedList=$2488	
$2488.type=ExpLink	
$2488.next=$2490	
$2488.entry=$2464	
$2490.type=ExpLink	
$2490.entry=$2452	
$2498.type=VarList	
$2498.linkedList=$2500	
$2500.type=VarLink	
$2500.next=$2502	
$2500.entry=$2452	
$2502.type=VarLink	
$2502.entry=$2468	
$2506.type=Relation	
$2506.lhs=$2498	
$2506.rhs=$2476	
$2506.relation=In	
$2476.type=String	
VALUE($2454) = x	
VALUE($2466) = y	
VALUE($2470) = z	
VALUE($2472) = f	
VALUE($2474) = g	
VALUE($2476) = X	

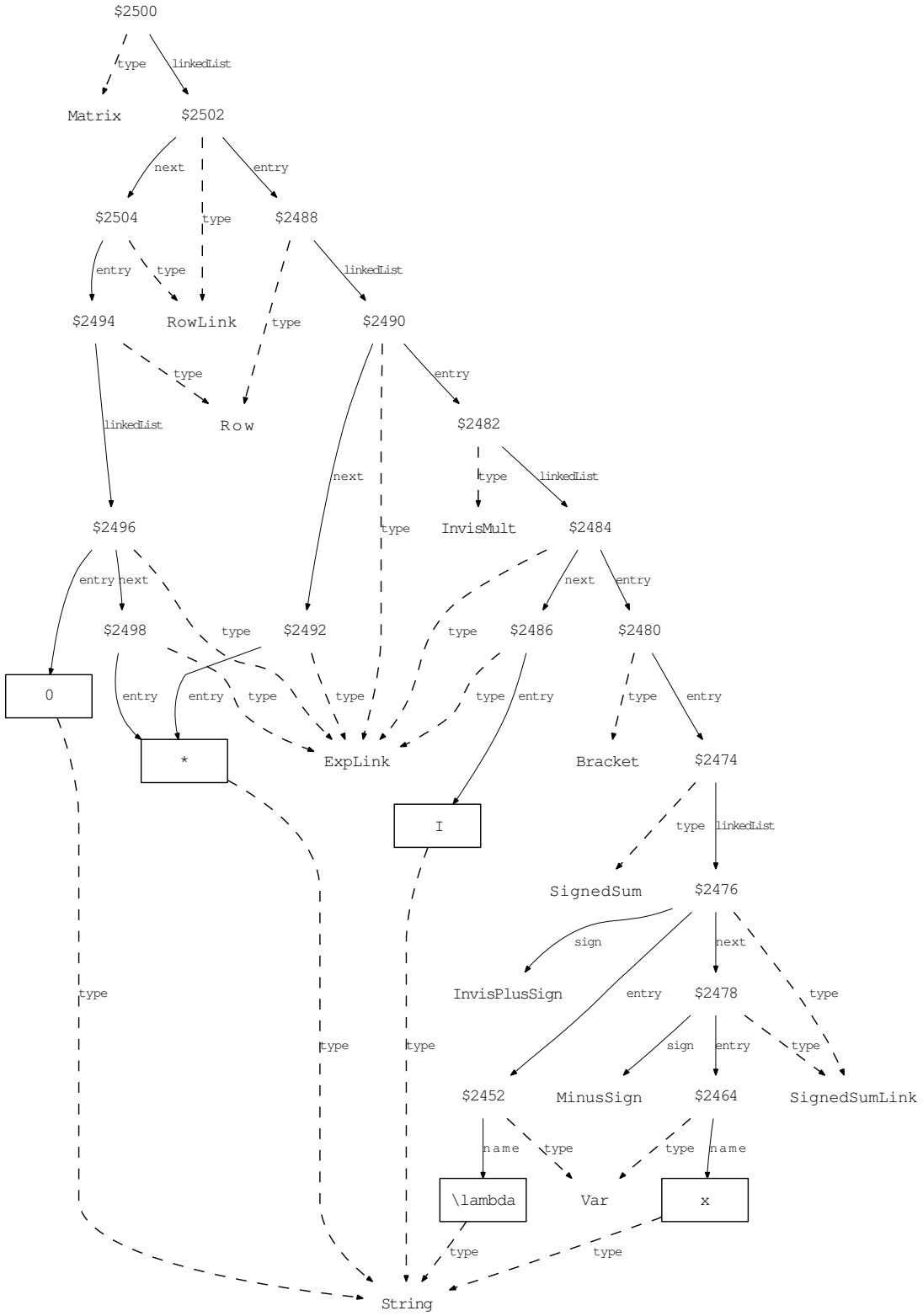

$2508
type
formula
Forall
$2496
rhs
type
lhs
scopedvar
$2494
Equal
$2492
binds
operator
arguments
type
type
arguments
$2486
Of
$2478
operator
$2506
g
type
type
linkedList
lhs
relation
type
linkedList
Vector
$2480
$2498
In
rhs
Relation
next
f
linkedList
type
$2488
$2482
$2500
VarList
type
next
entry
entry
next
entry
next
X
type
type
$2490
type
$2464
$2484
entry
$2502
type
type
entry
type
name
entry
type
entry
type
ExpLink
type
$2452
$2468
VarLink
type
y
name
type
type
name
type
x
Var
z
type
type
String

Example 2.

$$\begin{pmatrix} (\lambda - x) I & * \\ 0 & * \end{pmatrix}$$

```
\left(\begin{array}{cc} \left( \lambda - x \right) I & * \\
 0 & * \\ \end{array} \right)
```

record	MATLAB construction code
$2500.type=Matrix	lambda=createvar('\lambda');
$2500.linkedList=$2502	x=createvar('x');
$2502.type=RowLink	id=createstring('I');
$2502.next=$2504	ast=createstring('*');
$2502.entry=$2488	zero=createstring('0');
$2488.type=Row	iplus=createname('InvisPlusSign');
$2488.linkedList=$2490	minus=createname('MinusSign');
$2490.type=ExpLink	mi=create('SignedSum',{{iplus,lambda},{minus,x}});
$2490.next=$2492	bra=create('Bracket',mi);
$2490.entry=$2482	mult=create('InvisMult',{bra,id});
$2482.type=InvisMult	row1=create('Row',{mult,ast});
$2482.linkedList=$2484	row2=create('Row',{zero,ast});
$2484.type=ExpLink	ex=create('Matrix',{row1,row2});
$2484.next=$2486	
$2484.entry=$2480	
$2480.type=Bracket	
$2480.entry=$2474	
$2474.type=SignedSum	
$2474.linkedList=$2476	
$2476.type=SignedSumLink	
$2476.next=$2478	
$2476.entry=$2452	
$2476.sign=InvisPlusSign	
$2452.type=Var	
$2452.name=$2454	
$2454.type=String	
$2478.type=SignedSumLink	
$2478.entry=$2464	
$2478.sign=MinusSign	
$2464.type=Var	
$2464.name=$2466	
$2466.type=String	
$2486.type=ExpLink	
$2486.entry=$2468	
$2468.type=String	
$2492.type=ExpLink	
$2492.entry=$2470	
$2470.type=String	
$2504.type=RowLink	
$2504.entry=$2494	
$2494.type=Row	
$2494.linkedList=$2496	
$2496.type=ExpLink	
$2496.next=$2498	
$2496.entry=$2472	
$2472.type=String	
$2498.type=ExpLink	
$2498.entry=$2470	
VALUE($2454) = \lambda	
VALUE($2466) = x	
VALUE($2468) = I	
VALUE($2470) = *	
VALUE($2472) = 0	


$2500
type
linkedList
Matrix
$2502
next
entry
$2504
type
$2488
entry
type
linkedList
$2494
RowLink
type
$2490
type
Row
entry
linkedList
$2482
next
type
linkedList
$2496
type
InvisMult
$2484
entry
next
next
entry
$2498
type
$2492
type
$2486
$2480
0
entry
entry
type
type
type
entry
type
entry
*
ExpLink
Bracket
$2474
I
type
linkedList
SignedSum
$2476
sign
next
InvisPlusSign
entry
$2478
type
type
type
type
$2452
MinusSign
$2464
SignedSumLink
name
type
type
name
\lambda
Var
x
type
type
String

References

1. Andrews, P.: A Universal Automated Information System for Science and Technology. In: First Workshop on Challenges and Novel Applications for Automated Reasoning, pp. 13–18 (2003)
2. Beasley, J.: OR-Library: Distributing test problems by electronic mail. Journal of the Operational Research Society **41**(11), 1069–1072 (1990)
3. Boyer, R., et al.: The QED Manifesto. Automated Deduction–CADE **12**, 238–251 (1994)
4. Clark, J., Murata, M., et al.: Relax NG specification – Committee Specification 3 December 2001. Web document (2001). `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`
5. Covington, M.: A fundamental algorithm for dependency parsing. In: Proceedings of the 39th annual ACM southeast conference, pp. 95–102. Citeseer (2001)
6. Cramer, M., Fisseni, B., Koepke, P., Kühlwein, D., Schröder, B., Veldman, J.: The Naproche Project Controlled Natural Language Proof Checking of Mathematical Texts. Controlled Natural Language pp. 170–186 (2010)
7. Fourer, R., Gay, D., Kernighan, B.: A modeling language for mathematical programming. Management Science **36**(5), 519–554 (1990)
8. Ganter, B., Wille, R.: Formale Begriffsanalyse: Mathematische Grundlagen. Springer-Verlag Berlin Heidelberg New York (1996)
9. Humayoun, M., Raffalli, C.: MathNat – Mathematical Text in a Controlled Natural Language. Special issue: Natural Language Processing and its Applications p. 293 (2010)
10. Jefferson, S., Friedman, D.: A simple reflective interpreter. LISP and symbolic computation **9**(2), 181–202 (1996)
11. Jurafsky, D., Martin, J., Kehler, A., van der Linden, K., Ward, N.: Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition, vol. 163. MIT Press (2000)
12. Klarlund, N., Moeller, A., I., S.M.: Meta-DSD. Web document (1999). `http://www.brics.dk/DSD/metadsd.html`
13. Kofler, K.: A Dynamic Generalized Parser for Common Mathematical Language. PhD thesis (2011). In preparation
14. Kofler, K., Schodl, P., Neumaier, A.: Limitations in Content MathML. Technical report (2009). `http://www.mat.univie.ac.at/~neum/FMathL.html#Related`
15. Kofler, K., Schodl, P., Neumaier, A.: Limitations in OpenMath. Technical report (2009). `http://www.mat.univie.ac.at/~neum/FMathL.html#Related`
16. Lee, D., Chu, W.: Comparative analysis of six XML schema languages. ACM SIGMOD Record **29**(3), 76–87 (2000)
17. Lee, T., Hendler, J., Lassila, O., et al.: The semantic web. Scientific American **284**(5), 34–43 (2001)
18. Manola, F., Miller, E., et al.: RDF Primer. Web document (2004). `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`
19. McCarthy, J.: A micro-manual for LISP – not the whole truth. ACM SIGPLAN Notices **13**(8), 215–216 (1978)
20. Neumaier, A.: Analysis und lineare Algebra. Lecture notes (2008). `http://www.mat.univie.ac.at/~neum/FMathL.html#ALA`
21. Neumaier, A.: The FMathL mathematical framework. Draft version (2009). `http://www.mat.univie.ac.at/~neum/FMathL.html#foundations`
22. Neumaier, A., Marginean, F.A.: Models for context logic. Draft version (2010). `http://www.mat.univie.ac.at/~neum/FMathL.html#Contextlogic`
23. Neumaier, A., Schodl, P.: A Framework for Representing and Processing Arbitrary Mathematics. Proceedings of the International Conference on Knowledge Engineering and Ontology Development pp. 476–479 (2010). An ealier version is available at `http://www.mat.univie.ac.at/~schodl/pdfs/IC3K_10.pdf`
24. Neumaier, A., Schodl, P.: A semantic Turing machine. Draft version (2010). `http://www.mat.univie.ac.at/~neum/FMathL.html#SVM`, (to be revised soon)
25. Ranta, A.: Grammatical framework. Journal of Functional Programming **14**(02), 145–189 (2004)
26. Schodl, P.: Foundations for a self-reflective, context-aware semantic representation of mathematical specifications. PhD thesis (2011). In preparation
27. Schodl, P., Neumaier, A.: An experimental grammar for German mathematical text. Manuscript (2009). `http://www.mat.univie.ac.at/~neum/FMathL.html#ALA`
28. Schodl, P., Neumaier, A.: Representing expressions in the semantic memory. Draft version (2010). `http://www.mat.univie.ac.at/~neum/FMathL.html#TypeSystem`
29. Schodl, P., Neumaier, A.: The FMathL type system. Draft version (2010). `http://www.mat.univie.ac.at/~neum/FMathL.html#TypeSystem`
30. Schodl, P., Neumaier, A.: A typesheet for optimization problems in the semantic memory. Web document (2011). Available at `http://www.mat.univie.ac.at/~neum/FMathL.html#TypeSystems`
31. Schodl, P., Neumaier, A.: A typesheet for types in the semantic memory. Web document (2011). Available at `http://www.mat.univie.ac.at/~neum/FMathL.html#TypeSystem`
32. Shapiro, S.: An introduction to SNePS 3. Conceptual Structures: Logical, Linguistic, and Computational Issues pp. 510–524 (2000)
33. Stamerjohanns, H., Kohlhase, M.: Transforming the arχiv to XML. Intelligent Computer Mathematics pp. 574–582 (2010)
34. Sutcliffe, G., Suttner, C.: The TPTP problem library. Journal of Automated Reasoning **21**(2), 177–203 (1998)

35. Trybulec, A., Blair, H.: Computer assisted reasoning with Mizar. In: Proceedings of the 9th International Joint Conference on Artificial Intelligence, pp. 26–28. Citeseer (1985)
36. Walsh, T.: A Grand Challenge for Computing Research: a mathematical assistant. In: First Workshop on Challenges and Novel Applications for Automated Reasoning, pp. 33–34 (2003)