A graph-based type system for mathematical content

Peter Schodl[1] and Arnold Neumaier[1]

Fakultät für Mathematik, Universität Wien, Nordbergstr. 15, A-1090 Wien, Austria
peter.schodl@univie.ac.at, arnold.neumaier@univie.ac.at
WWW home page: http://www.mat.univie.ac.at/~neum/FMathL.html

Abstract. We discuss a language for the specification of the representation of objects in the FMathL semantic network to represent arbitrary mathematics. The language also serves as an ontology language. The FMathL type system is a common generalization of context-free grammars and algebraic data types in programming languages. Type declarations themselves are represented via typed objects, making the whole typing self-consistent in the sense of a meta schema. We provide a type system (and hence an ontology) for optimization problems.
The semantic network is implemented in Java, but could also be implemented in RDF.

Keywords: Representation of mathematical knowledge, Type system, Ontology authoring language, Algebraic data structure, Context-free grammar.

1 Introduction

FMathL (**F**ormal **Math**ematical **L**anguage) is a project that aims at transparently representing mathematical knowledge in a directed labeled graph, and therewith providing a framework for algorithms performing tasks on this representation. When used for the representation of knowledge, such graphs are usually called "semantic networks", introduced by Richens [20] in 1956. FMathL makes use of a special semantic network we call the **semantic memory** that is formally equivalent to a restriction of RDF-tuples [14, 17], and hence representable in the semantic web [16].

Since algorithms are to perform tasks on the semantic memory, we need to be able to specify the structure of the represented knowledge. This is traditionally done by data types.

We introduce a type system that enables us to easily express algebraic data types and context-free grammars as FMathL types. Furthermore, by the use of type inheritance, our type system also serves as an ontology language. To serve as a foundation in the sense of the FMathL framework [18], everything is set up in a way that it can reflect itself without the need to augment the basic structure.

Acknowledgments. Thanks to Hermann Schichl, Ferenc Domes and Kevin Kofler for their comments in various discussions of preliminary versions. Furthermore we would like to thank Michael Kohlhase for valuable input. Support by the Austrian Science Foundation (FWF) under contract number P20631 is gratefully acknowledged.

2 The semantic memory

The semantic memory of FMathL implements semantic networks such that the nodes may have an **external value** associated to them, i.e., arbitrary data stored outside the graph. When we print a semantic memory here, nodes with 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 constituents have importance for the typing. The same node of a semantic memory may be printed multiple times if it enhances readability.
Due to the similarity with algebraic data types, we call the labels of the out-edges of some node the **fields** of this node.
The form of representation that is generally used is similar to expression graphs for mathematical formulae, and parse trees of a dependency grammar for natural language. That is, each non-leaf is an application of an operation (represented in the child in the field `type`), while the children in all the other fields can be regarded as arguments of that operation.
A similar style of representation was chosen in the SNePS framework [24].
For example, the expression $\frac{12}{4} = 3$ may be represented the semantic memory in Figure 1.

3 The type system

Types that pose requirements on children of a node are called **proper types**. Requirements can be inherited between proper types, in the sense of code reuse. We are also able to express a sub- and supertype relation, not to be confused with inheritance: a **supertype** does not directly pose requirements on the children of a node, but is rather a collection of types. An instance of a supertype is an instance of one of its subtypes. When using the type system in the sense of an ontology, a supertype is a concept that consists of its subtypes, while inheritance expresses a concept that is more special than the concept it inherited from.
Atomics are the third kind of types. These are nodes that are not allowed to have any children, and are used in the ontology as distinguished objects with meaning in themselves. Examples are the atomics `true`, `false`, `present`.

3.1 Type declarations

A **type declaration** is a text defining a type. A document containing several type declarations is called a **type sheet**.
A type declaration of a proper type has the following structure:

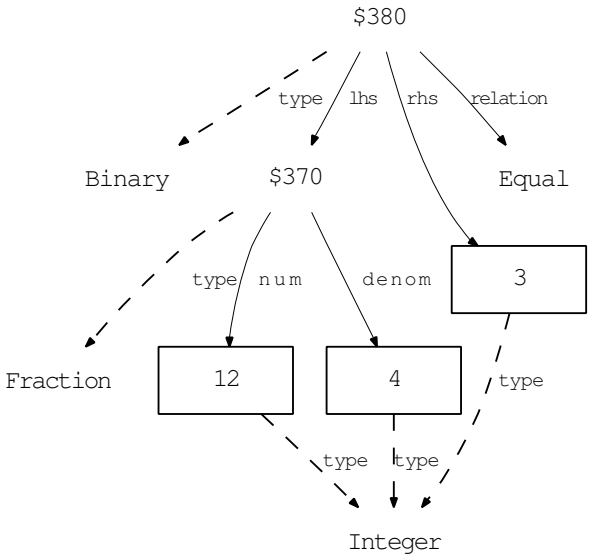


Fig. 1. A semantic graph

1. the name of the proper type (followed by a colon)
2. then optional other proper types (each followed by a `+`) to inherit requirements from
3. then lines of requirements starting with certain keywords listed in Table 1 followed by a greater sign (`>`) and together with some arguments.

In (2), the final `+` is missing if no lines of the form (3) follow.
Type declarations for unions and atomics has the following structure:

1. the name of the union type (followed by a colon)
2. followed by lines starting with certain keywords listed in Table 2 possibly followed by further specifications.

3.2 Examples

We give some examples of type declarations to give and intuition how the keywords are employed in applications. A detailed discussion of the precise constraints the keywords express, and more examples are given in [22] and [21].

Example 1: An integral must have either a field `over` or a field `fromTo`, but not both. The following semantic graph gives the representation of

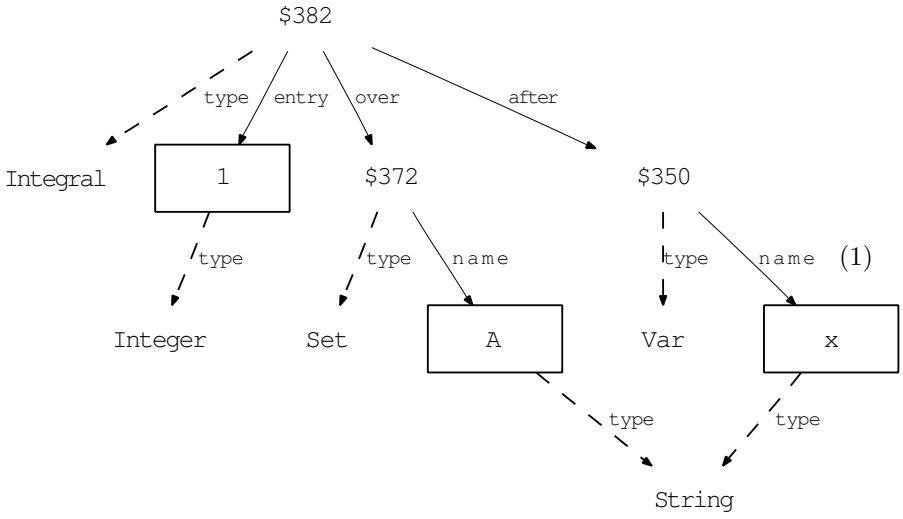
$$\int_A 1 \, \mathrm{d}x.$$

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	requires specific 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
`index`	list of equations	requires to index each instance
`nothingElse`	none	forbids further fields

Table 1. Keywords in declarations of proper types

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
`index`	list of equations	requires to index each instance

Table 2. Keywords in declarations of unions and atomics



The restrictions we want to express are that the child in field `over` must be a set, and the child in field `fromTo` must be an expression. For this, we use the

quantifier `oneOf` in the type declaration of `Integral`. We assume that `Integer` is a subtype of `Expression`.

```
Integral:
oneOf> fromTo=Expression
       over=Set
allOf> entry=Expression
       after=Var
```

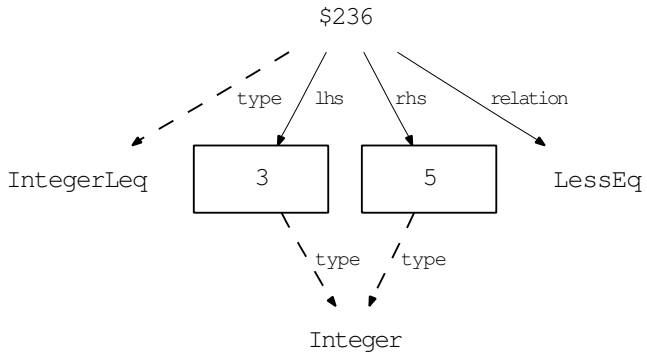
Example 2: We define a special binary relation `IntegerLessEq`:

```
IntegerLessEq:
allof> lhs=Integer, rhs=Integer
fixed> relation=LessEq
```

Then the relation

$$3 \leq 5$$

would be represented by:

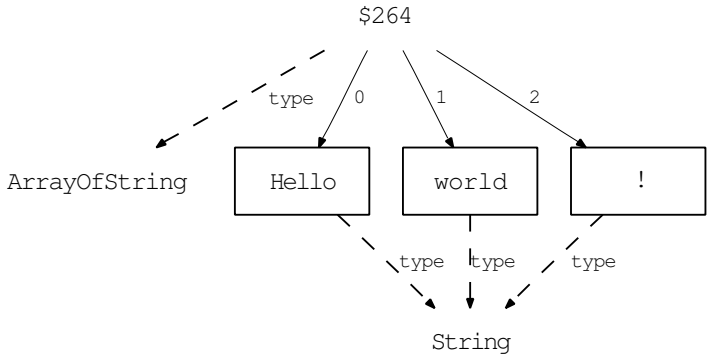


(2)

Example 3: We define a sparse random-access array of strings, where each string is accessible by an integer. So every constituent of this record has to have an integer as a field, and a String as an entry.

```
ArrayOfString:
someOfType> Integer=String
```

The following is the representation of the array of strings with entry 0 is “Hello”, entry 1 is “world”, and entry 2 is “!”.



(3)

Example 4: Inheritance adds the specifications from an existing type declaration to a newly defined type declaration. For example, if we want a proper type that uses all specifications of the type `Norm` as defined above, but adds an optional comment, the type declaration

```
NormWithComment:
allOf> entry=Expression
optional> index=Expression
          comment=String
```

is equivalent to the shorter version

```
NormWithComment: Norm +
optional> comment=String
```

that uses inheritance. Similarly, we can also define the **intersection** of two types. Given the type declaration

```
Comment:
optional> comment=String
```

we can equivalently define

```
NormWithComment: Norm + Comment
```

The same is possible for unions: if we assume a union `Document` that was defined by the type declaration

```
Document:
union> LatexDocument, PlainText
```

and now we want to add `SpreadSheet` as a further subtype, we write

```
Document: Document +
union> SpreadSheet
```

Example 5: We want to define the union `Rational` containing `Integer`, `Float` and `Double`, and the union `Number` containing `Integer`, `Float`, `Double`, `Rational` and `Real`. We specify this in the type sheet by

```
Rational:
union> Integer, Float, Double

Number:
union> Real, Rational
```

Note that `Float`, `Double`, `Real` and `Integer` have to be categories.

Example 6: The type declaration `Documents` should only contain the categories `Article`, `Report` and `Book`, but not anything else.

```
Documents:
union> Article, Report, Book
complete>
```

It is still possible to later define a new category `Shortbook` as a subtype of `Book`, e.g., with the declaration:

```
Book:
union> Shortbook
```

It is however *not* possible to add `Shortbook` as a subtype of `Documents`, due to the keyword `complete>`.

4 Algebraic data types as FMathL types

Assume a data base that contains the staff of an organization. Each person in the data base is represented as a record with a `name`, which consists of a `first` and a `second` name, a `telephone` which is an integer, and a `superior`, which is again a person.
In a Pascal-like notation, the type of `person` would be expressed as:

```
type
person = record
  name: namerecord
  telephone: Integer
  superior: person

namerecord = record
  first: String
  second: String
```

The corresponding type in FMathL is:

```
person:
allOf> name = namerecord
       telephone = Integer
       superior = person

namerecord:
allOf> first = String
       second = String
```

When represented in the semantic memory, the fact that one of the fields of every instance of this type requires an instance of the same type simply reflects in a cycle in the graph representing the type structure (which is itself represented in the semantic memory, see Section 7) and does not cause any problem or need for special treatment.

5 Context-free grammars as FMathL types

Context-free grammars can also be represented naturally. Assume the simple context-free grammar for the non-regular language $a^n b^n$:

```
S -> X
X -> aXb | ϵ
```

The corresponding type in FMathL is:

```
S:
allOf> alternative1 = X

X:
oneOf> alternative1 = Xa1
       alternative2 = Xa2

Xa1:
fixed> first = a
allOf> second = X
fixed> third = b

Xa2:
fixed> first = EmptyString
```

Again, the recursive structure of the grammar leads to a cycle in the representation of this type system in the semantic memory.

The representations of text in the semantic memory of this type are exactly the parse trees of the grammar given above.

6 A type sheet for optimization problems

We have written a type sheet which specifies the representation of optimization problems. We chose this particular set of mathematical texts because this is where our expertise lies.
We also wrote typesheets for mathematical formulas (240 lines) which is also used for the optimization problems.
We represented a number of optimization problems as specified in these type sheets, an wrote an algorithm that can produce both latex and AMPL output [3].
A optimization problem typically has the form:

> Given [a list of objects], find [a list of objects] that minimize/maximize the value of [a formula] under the constraints [a list of formulas].

A simple example is the well-known traveling salesman problem:

> Given a set of nodes N, a set of arcs $A \subseteq N \times N$ and weights $\{w_{ij} \geq 0 \mid (i,j) \in A\}$, find a sequence $s_1, \ldots, s_k$ in N that minimizes the value of $\sum w_{n_i n_{i+1}}$ under the constraints that for all $n \in N$, $n = s_i$ for some i and $s_1 = s_k$.

We give an essential part of the type sheet for optimization problems. Due to space limitations this typesheet of originally 184 lines it is not completely given here.

```
Problem:
  allOf> find = Objects
         target = Target
  optional> given = ObjConc
            constraint = Constraints
```

The fields `given` and `constraints` are optional, which reflects the existence of problems without explicitly given objects (e.g.: "Find $a \in \mathbb{N}$ such that a is maximal under the constraint that a is prime"), and without constraints, which are called "unconstrained problems".

```
Objects:
  union> ObjList, Obj, Expression

Obj:
  optional> formula = Expression
            entry = Expression
            qualification = Qualification
            specification = TextUnit
            typeofobj = ConceptGeneral
            indexrange = Expression
```

```
            with = Expression
            interpretation = TextUnit

ObjList:
  allOf> linkedList = ObjListLink

ObjListLink:
  allOf> entry = Objects
  optional> next = ObjListLink
```

These types define an `Object` is, i.e, one or a linked list of nodes in the semantic memory. The mathematical structures that would be represented as an instance of `Object` are things like "a positive real number c_i for every $i \in I$ (the price of item i)" where "positive real number" would be represented in field `specification`, "c_i" in filed `formula`, "for every $i \in I$" in field `indexrange`, and "the price of item i" in field `interpretation`.

```
Target:
  allOf> formula = Expression
         mode = MinMax
  optional> restriction = Expression

MinMax:
  atomic> Minimize, Maximize
```

Instances of the type `Target` are formulas together with the information whether this expression is to me minimized or maximized, which is represented by the two atomics `Minimize` and `Maximize`.

```
Constraints:
  union> ConstraintList, Constraint

Constraint:
  allOf> formula = Expression
  optional> restriction = Expression
            name = Integer

ConstraintList:
  allOf> linkedList = ConstraintListLink

ConstraintListLink:
  allOf> entry = Constraint
  optional> next = ConstraintListLink
```

Instances of the type `Constraints` are one or a linked list of (possibly restricted) formulas as in $a_k (k = 1, \ldots, n-1)$. They also can be assigned an integer, which may be used as a name of this constraint.

7 Type declarations and unions as types

In this section, we give a type system that defines the type of a type system, both as a type sheet and represented in the semantic matrix. As a type sheet, the type of a type system has 66 lines. Here we give the type declaration of the most basic types, the complete type sheet can be found in [22].

```
Category:
  union> Union, Type, Atomic

Type:
  index> Index=Types
  someOf> template=Type
          allOf=Assembly
          someOf=AssemblyLink
          optional=Assembly
          oneOf=AssemblyLink
          someOfType=CatAssembly
          index=Assembly
          itself=Categories
          nothingElse=Present
          fixed=ObjAssembly
          array=Assembly
          only=Assembly
  optional> index=Assembly
            parents=Unions
            extends=Type

Union:
  index> Index=Unions
  allOf> subtypes=SubTypes
         parents=Unions
  someOf> atomic=Atomics
          union=Categories
  optional> complete=Present
            extends=Union
            index=Assembly

Atomic:
  atomic> Present
  index> Index = Atomics
```

We represented this type sheet in the semantic memory using 126 edges.
In particular, since the types are represented, this typesheet ensures that the information necessary for typing is itself typed, and an algorithm that performs the type checking gets data of the appropriate structure.

8 Annotation with input- and output information

We have the possibility to specify for each type information about input (used during a parse-stage to automatically generate a typed representation) and output (used by algorithms that produce output from the representation). These annotations correspond to "notation declarations" in OMDoc [12].
We have an annotated type sheet that specifies the grammar of annotated type sheets, having 325 lines.
For example, we give the type `Comment`:

```
Comment:
allOf> text=Line
#t> ##blanks&!##blanks#text&n
```

An instance of this type has an object of type `Line` in field `text`. The annotation follows the hash (`#`) in the last line, and means that both for input and output (otherwise there would be `#tr` for input and `#tw` for output) the text representation is a non-escaped exclamation mark (!) followed by the text represented in the field `text`, followed by a newline.
We used this annotated type sheet to parse type sheets and automatically create a typed representation of annotated type sheets in the semantic memory. The dynamic generalized parser (DynGenPar) by Kevin Kofler [9] is an important step towards this goal. DynGenPar has the ability to parse dynamically, i.e., changes in the grammar induced by changing type specifications or annotations are easily accomodated in the parser. This is important for parsing mathematical documents, as their content changes the gramma via definitions of concepts and notation.

9 Related work

A semantic network that is equipped with a form of typing has been introduced as the "graph-oriented database model" (GOOD) [4] in 1994.
Data types for graphs in the sense we employ them, have been discussed in [6]. However, these types are mainly concerned with the global structure of the graph, and have no means to express local structure such as alternative choices of fields.
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 [15].) These cannot be compared directly, since XML documents are always organized in a tree structure. But 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`.

9.1 Comparison of FMathL with OMDoc

A widely known system to represent mathematical knowledge is OMDoc [10, 11], an XML-based markup language. OMDoc is an extension of (the likewise XML-based) OpenMath, but is able to represent mathematics on higher levels, up to whole theories. For the formula level, OMDoc uses the syntax of OpenMath [2] and is therefore able to profit from the existing infrastructure of OpenMath for rendering, symbolic calculations, etc. OMDoc also provides an input language in the form of sTeX [13] and a database of formalized mathematics called MBase [1].
Both FMathL and OMDoc want to represent mathematics on every level (symbols up to whole theories), and aim to be both human- and machine readable. Nevertheless, there are also big differences:

- FMathL is not a markup language, the most basic layer is not a text document but rather a graph. Nevertheless, from well-structured information represented in FMathL, text representations (also marked-up text) can be produced, and the graph-representation can be generated from various text representations. This adds flexibility as the text representation can be customized using annotated type sheets.
- In FMathL, unlike in tree-based XML based languages, cycles in the graph are permitted and play an important role (see Sections 4 and 5).
- FMathL does not build upon existing systems and hence does not inherit certain limitations of the underlying systems, see [7, 8].
- FMathL also represents algorithms that perform tasks on this representation. In particular, a virtual machine to perform tasks on the semantic memory was defined and implemented, for details see [19].
- The approach is different, as FMathL is intended to work with interfaces to different systems (e.g., theorem provers, solvers for partial differential equations, etc.) based on different paradigms. For example, a user should be able to formulate numerical problems in FMathL and the system would call the appropriate solvers and return the answer in the user-requested format. Hence it does not provide the user with a lot of high-level built in functionality, but rather aims to be flexible enough to serve as a representation framework for a wide variety of mathematical knowledge of different formality level.

10 Conclusion

We presented a concept of typing that serves both as a specification of representations in a special semantic network, and as an ontology. We introduced the specification language and presented an application to the representation of optimization problems.
A number of optimization problems have been represented according to this specification, and both LaTeX and AMPL [3] output has been produced to prove practicability. The type system is used in Concise, a graphical user interface

for the semantic memory of FMathL [23]. The fact that the type specifications are themselves part of the semantic memory and the availability of a dynamic parser [9] allows in principle a dynamical processing of mathematical content. However, before the system will become useful for the working mathematician, further work needs to be done to automatically read in mathematical content from existing representations. We are currently working on a flexible formula parser, and on adapting the MathNat grammar by Humayoun [5] to meet this requirement.

References

1. Andreas, K., et al.: MBase: Representing knowledge and context for the integration of mathematical software systems. Journal of Symbolic Computation 32(4), 365–402 (2001)
2. Caprotti, O., Carlisle, D.: OpenMath and MathML: semantic markup for mathematics. Crossroads 6(2), 14 (1999)
3. Fourer, R., Gay, D., Kernighan, B.: A modeling language for mathematical programming. Management Science 36(5), 519–554 (1990)
4. Gyssens, M., Paredaens, J., Van den Bussche, J., Van Gucht, D.: A graph-oriented object database model. Knowledge and Data Engineering, IEEE Transactions on 6(4), 572–586 (1994)
5. Humayoun, M., Raffalli, C.: Mathnat-mathematical text in a controlled natural language. Special issue: Natural Language Processing and its Applications 46 (2010)
6. Klarlund, N., Schwartzbach, M.: Graph types. In: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 196–205. ACM (1993)
7. Kofler, K., Schodl, P., Neumaier, A.: Limitations in Content MathML. Technical report (2009), `http://www.mat.univie.ac.at/~neum/FMathL/content-mathml-limitations.pdf`
8. Kofler, K., Schodl, P., Neumaier, A.: Limitations in OpenMath. Technical report (2009), `http://www.mat.univie.ac.at/~neum/FMathL/openmath-limitations.pdf`
9. Kofler, K., Neumaier, A.: A Dynamic Generalized Parser for Common Mathematical Language. Work-in-progress paper at CICM 2011 (2011), `http://www.tigen.org/kevin.kofler/fmathl/dyngenpar-wip.pdf`
10. Kohlhase, M.: OMDoc: An infrastructure for OpenMath content dictionary information. ACM SIGSAM Bulletin 34(2), 48 (2000)
11. Kohlhase, M.: OMDoc: Towards an internet standard for the administration, distribution, and teaching of mathematical knowledge. In: Artificial Intelligence and Symbolic Computation. pp. 32–52. Springer (2001)
12. Kohlhase, M.: OMDoc–an open markup format for mathematical documents, vol. 4180. Springer-Verlag New York Inc (2006)
13. Kohlhase, M.: Using LATEX as a semantic markup format. Mathematics in Computer Science 2(2), 279–304 (2008)
14. Lassila, O., Swick, R., et al.: Resource Description Framework (RDF) Model and Syntax Specification (1999)
15. Lee, D., Chu, W.: Comparative analysis of six XML schema languages. ACM Sigmod Record 29(3), 76–87 (2000)

16. Lee, T., Hendler, J., Lassila, O., et al.: The semantic web. Scientific American 284(5), 34–43 (2001)
17. Manola, F., Miller, E., et al.: RDF Primer. Web document (2004), http://www.w3.org/TR/2004/REC-rdf-primer-20040210/
18. Neumaier, A.: The FMathL mathematical framework. Draft Version 1.04 (2009), http://www.mat.univie.ac.at/~neum/ms/fmathl.pdf
19. Neumaier, A., Schodl, P.: A semantic virtual machine. Manuscript (2011), http://www.mat.univie.ac.at/~schodl/SVM.pdf
20. Richens, R.H.: Preprogramming for mechanical translation. Mechanical Translation 3(1), 20–28 (1956)
21. Schodl, P.: Foundations for a Self-Reflective, Context-Aware Semantic Representation of Mathematical Specifications. PhD thesis (2011), http://www.mat.univie.ac.at/~schodl/pdfs/diss_online.pdf
22. Schodl, P., Neumaier, A.: The FMathL type system. Manuscript (2011), http://www.mat.univie.ac.at/~schodl/pdfs/types.pdf
23. Schodl, P., Neumaier, A., K.Kofler, Domes, F., Schichl, H.: Towards a Self-reflective, Context-aware Semantic Representation of Mathematical Specifications. Springer (to appear)
24. Shapiro, S.: An introduction to SNePS 3. Conceptual Structures: Logical, Linguistic, and Computational Issues pp. 510–524 (2000)