universität
wien

DISSERTATION

Titel der Dissertation

Foundations for a self-reflective, context-aware semantic representation of mathematical specifications

Verfasser

Peter Schodl

angestrebter akademischer Grad

Doktor der Naturwissenschaften (Dr.rer.nat)

Wien, im Juli 2011

Studienkennzahl lt. Studienblatt:	A 091 405
Dissertationsgebiet lt. Studienblatt:	Mathematik
Betreuer:	Univ.-Prof. Dr. Arnold Neumaier

Contents

Abstract

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 and processed inside a framework and can then be communicated to numerical solvers or other systems.

As a first step towards a general purpose tool for representing and interfacing general mathematics on the computer (the FMathL project), we developed a representation of mathematics in a semantic network called the semantic memory, together with a type system that checks validity of the representation and a virtual machine that can execute algorithms.

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.

Chapter 1

Introduction

Mathematicians nowadays rely heavily on computers. They use them to communicate with colleagues, search the web for information, create documents they want to publish, perform numerical and symbolic computations, check their proofs, store the work they have done, etc. However, since mathematicians address very diverse parties with their writing, a mathematician usually has to formulate the same idea (e.g., a proof, a numerical problem, a conjunction) multiple times, depending on the recipient: for a student in great detail, for a foreign colleague working in the same field in less detail but a common language, for a publication in a document markup language, for a numerical solver in an algebraic modeling language, for a proof checker in a special language and at a tremendous level of detail.

If a general representation with rich possibilities to interface the information proves feasible, it may also contribute to an electronic database containing essential amounts of the known mathematics. This vision is not new, it dates back at least to the QED project [3]. Its goal was to represent all important mathematical knowledge, conforming to the highest standards of mathematical rigor. Another vision in this direction was the universal automated information system for all sciences [1]. This is even more ambitious than a universal mathematical database, but the prominent role of mathematics in such a system, also discussed in [1], would make a mathematical database probably a corner stone of such a system and could be a starting point.

While mathematicians usually represented their work in LaTeX documents or algorithms written in programming languages, these forms are very rigid and problems arise concerning semantic searches, re-usability, consistency checks, etc.

To overcome these problems, several representations of mathematics which at least include semantics have been developed; we mention some of the most important ones:

- MathML and OpenMath [4], XML-based mathematical markup languages for mathematical formulas. Both are, however, restricted, see [20] and [21].
- OMDoc [22], an XML-based format based on MathML and OpenMath for general mathematical objects.
- Languages for formalized mathematics, such as Mizar [49], which require formal definitions of all occurring symbols, functions, relations, etc.
- Annotated documents such as MathLang [17] or sLaTeX [23], in which LaTeX documents are enriched with semantic content.

Our approach aims at simplicity and transparency of the internal representation, and so we chose to represent mathematics in a labeled graph, and more particular in a semantic network, introduced by Richens [38] in 1956. Semantic networks and akin concepts are discussed in detail by Sowa [45]. The semantic memory is compatible with, and implementable in the semantic web [27].

Also, we want to keep the processing as flexible as possible, therefore we did not implement a fixed set of algorithms to perform actions on the semantic network, but instead we defined a virtual machine discussed in Section 3. This aim is comparable to the GOOD database model [12], which also offers a data structure based on labeled graphs, some form of typing, and a transformation language to execute changes on the data.

When using a graph, or passing it to some algorithm, we need information about the structure of this graph, as we do not want to examine the whole graph every time it is used. For this reason we define a procedure to determine that a given graph is **well-typed** of a certain type, or **ill-typed**. These assignments are always made with respect to a particular type system.

Thus a concept of typing is needed that covers

- syntactically correct mathematical formulas,
- well-formed sentences built according to a linguistic grammar, and
- structured records in the programming sense.

1.1 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 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 [24] and MathNat [14], but with a different target), it is designed to be as expressive and natural as currently feasible.

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 [29] 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 [9] 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 [37]: A programming language for multilingual grammar applications, which allows us to produce grammatically correct sentences in multiple languages.

Naproche [24]: An interface from a controlled natural language to proof checking software, which can be used to interface proof checkers.

TPTP [47]: The library "Thousands of problems for theorem provers" provides facilities to interface interactive theorem provers.

1.2 Vision: The FMathL project

The MoSMath project is embedded into a far more ambitious long-term vision – the FMathL project, described in [33] and the extensive FMathL web site[1] (for a summary, see [34]).

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:

- 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:

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

- 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 [49]), where they only serve a single purpose.
- An assistant in the sense of WALSH [52] 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.

1.3 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 for the 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 program-

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

ming 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.

1.4 Overview

In Chapter 2 we describe a graph-based, implementation independent model of knowledge representation called the "semantic memory". It is representable as a special case of a semantic network, and influenced by, or akin to concept maps, the semantic web, parse trees for dependency parsing, etc. When regarded as a graph, the semantic memory is a directed, labeled graph where the nodes can have arbitrary data associated to them. The only restriction on the graph is that there cannot be two distinct arcs starting from the same node and having the same label.

Chapter 3 introduces a virtual machine called the "semantic virtual machine" (SVM) that can perform operations on the semantic memory. We do so to be able to rigorously argue about processes in the SM, and to be able to proof properties. Programs that are executed by the semantic virtual machine can be written in an assembler-style programming language, but are represented in the semantic memory. We formally define the action of the SVM by giving an operational semantics.

To show Turing completeness of the semantic virtual machine we give a program that simulates any given Turing machine. Also, we discuss a SVM program that interprets any given SVM program. Since this self-interpreter for the SVM is analogous to the a universal Turing machine, we call this program the universal semantic virtual machine.

In Chapter 4 we define a type system for the semantic memory. When data is to be processed by an algorithm, or we want to be able to control well-formedness of the data to prevent run-time errors.

Types are objects in the semantic memory, and they are associated to requirements, which can be defined using plain text documents called "type sheets". The requirements that can be posed via type sheets are comparable to the typing of XML documents. Also, the type system uses subtypes and

unions of types, inheritance, and objects that have meaning on their own. The requirements associated to a type are also represented in the semantic memory, and we give a formal definition of the algorithm that check whether or not data in the semantic memory is well-typed or not.

Since the requirements are also represented in the semantic memory, we can give the type of types, which is the analogon of a "meta schema" of type systems for XML.

Chapter 5 introduces actual application done with this framework:

- We gathered a set of different kinds of mathematical expression and represented them in the semantic memory. From this representation, the according LaTeX-formulas can be generated automatically.
- Informal mathematical text was represented in the semantic memory.
- Optimization problems from the OR Library [2] were represented in the semantic memory, and we are able to automatically generate a natural problem description, and a machine-processable AMPL model description.
- The all problems files from the TPTP Library [47] were represented in the semantic memory.
- Two example texts used in the Naproche project [24] were represented in the semantic memory. The automatically generated output is accepted by the Naproche web interface[3].

Appendix A gives examples of SVM programs. It contains an example of a simple program as it is represented in the semantic memory, and the code of the USVM.

Appendix B contains several type sheets. Since type sheets express requirements of data in the semantic memory, these type sheets can also be seen as a definition of the representation of several sorts of data in the semantic memory. We give type sheets for mathematical expressions, for Turing machines, for problems sets from the TPTP, and for optimization problems from the OR Library.

Appendices C and D give examples from the applications discussed in Chapter 5. Appendix C contains 31 mathematical expressions, both as represented in the semantic memory, and the LaTeX-output automatically created from this representation. Appendix D gives 10 optimization problem from the OR-Library. For each problem it contains both the automatically created description of the data and the numerical data in the OR Library, as well as the automatically created AMPL-file.

[3] `http://naproche.net/inc/webinterface.php`

Earlier results that overlap with this thesis have been published in [34] or are accepted for publication in [16]. Large parts of Sections 2 and 3 are submitted for publication as [35], and large parts of Sections 2 and 4 are submitted for publication as [42].

Chapter 2

Framework: 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 [11]). 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 [20], [21].

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 [11, 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 [27]. In particular, we have implemented it in RDF [28].

We define the abstract data structure we use to represent mathematics.

It can be regarded as a special case of a semantic network, introduced by Richens [38] in 1956. This and akin concepts are discussed in detail by Sowa [45]. Also, it is inspired by, and representable in, the semantic web [27]. A standardized and widely used example of a semantic net with the aim to be used in the World Wide Web is the Resource Description Framework (RDF), described by Manola et al. [28] and specified by Lassila et al. [25].

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 [43]. 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 [7]. However, a parse tree is always a tree and does not allow structure sharing.

2.1 The semantic memory

There is an unlimited number of **objects**, but only finitely many of them are represented explicitly in stored memory. Objects can be compared for equality, which is an equivalence relation. On the meta level, we refer to objects by strings not beginning with a hash (#); different objects are referred to by different strings. `Empty` is an object. **Object variables** are variables in the usual sense, ranging over the set of objects. We refer to object variables via a string beginning with a hash (#) followed by some alphanumeric string. For example, in the statement

#`name`.`type` = `String` for every object #`name` representing a string,

`type` and `Name` are specific objects, and #`name` is a variable in the same sense as x is a variable in

$$x^2 \text{ is even for every even integer } x.$$

Usually, we will use suggestive strings for variables, e.g., we use #`handle` or #`h` for an object that is intended to be a handle.

A **semantic mapping** (abbreviated SM) assigns to any two objects #`h` and #`f` a unique object #`h`.#`f` such that

if #`f` = `Empty` or #`h` = `Empty` then #`f`.#`h` = `Empty`.

A **semantic unit** (short **sem**) is an equation of the form #h.#f = #e with nonempty #h, #f, and #e; we call #h the **handle**, #f the **field**, and #e the **entry** of the sem. The **constituents** of an object #a are the sems in which #a is the handle.

Semantic mappings are used to store mathematics, but to be able to alter the data we need a dynamical framework. The semantic mapping that changes over time (formally, a semantic mapping valued function of time) is called the **semantic memory**.

A **position** is a pair (#h/#f) consisting of two objects #h and #f. We call #h the **handle**, #f the **field** and #h.#f the **entry** of (#h/#f). This position is called **occupied** if #h.#f is not Empty.

We say that the sem #d.#e=#f **follows** the sem #a.#b = #c if #d = #c. Using a left-associative notation, we then write #a.#b.#e = #f; thus #a.#b.#e stands for (#a.#b).#e. This notation naturally extends to more dots.

A short-hand notation for k repetitions ($k = 0, 1, 2, \ldots$) of a field:

$$\#\mathtt{a}.\underbrace{\#\mathtt{b}.\cdots.\#\mathtt{b}}_{k \text{ times}}.\#\mathtt{e}$$

is written as $\#\mathtt{a}.\#\mathtt{b}^k.\#\mathtt{e}$.

A **path of sems** starting at #h and ending at #e is a sequence of sems such that the first sem has the handle #h, each later sem follows the previous one, and the last sem has entry #e, and no sem has the field type. An object #e is **reachable** from a handle #h if there is some path of sems starting at #h and ending in #e. A sem is **reachable** from a handle #h if there is some path of sems starting at #h that contains that sem. A position is **reachable** from a handle #h if the handle of that position is an object reachable from #h.

If the set of sems reachable from an object #h is finite, then the set of sems reachable from #h defines the **record** with handle #h.

Clearly, a SM allows one to construct arbitrarily complex records. In contrast to records in programming languages such as Pascal, records in a SM may contain cycles. Indeed, back references are an important part of the design of the type system; for example, they allow labeled context-free grammars to be defined as type systems.

2.2 Semantic graphs

For graphical illustration of a semantic mapping, we will 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**. Objects may, but need not have

external values, i.e., data of arbitrary form, associated with the object, but stored outside the semantic memory. We refer to the value of an object #`obj` by `VALUE`(#`obj`). 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 constituents have importance for the typing, and bold edges for edges labeled with `next`, since this makes linked lists more readable. Different nodes of the semantic graph may represent the same object. For example, the information $\frac{12}{4} = 3$ may be represented as a list of sems as given in Figure 2.1, or equivalently as the semantic graph in Figure 2.2.

```
$380.type=Binary      $370.type=Fraction    VALUE($244) = 12
$380.lhs=$370         $370.num=$244         VALUE($246) = 3
$380.rhs=$246         $370.denom=$248       VALUE($248) = 4
$380.relation=Equal   $244.type=Integer
$246.type=Integer     $248.type=Integer
```

Figure 2.1: A list of sems and values

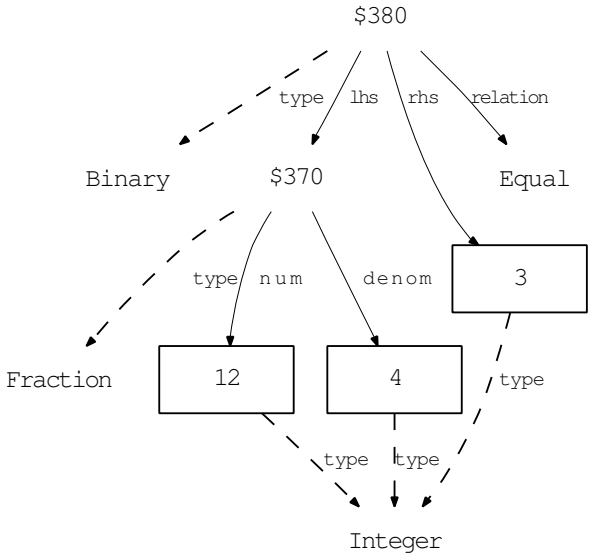


Figure 2.2: A semantic graph

Chapter 3

Algorithms: 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.

A Turing machine, introduced originally in 1936 by Turing [51], is a commonly used abstract model of a simple computer. Informally, we think of a Turing machine (TM) as a reading/writing head that moves along an arbitrary long tape which is divided into cells, each containing one character. The Turing machine is always in some state, and it has a list of instructions, usually called the transition table. Determined by the character currently read from the tape and the state the TM is currently in, the transition table assigns to the TM some character to write on the tape, to move one cell to the left or the right, and some state to enter. For a rigorous definition and properties, see, e.g., the classic book by Rogers [40] or Aho et al. [13] or almost any other computability book; see also Section 3.7 below.

The concept of a Turing machine is very simple and at the same time very powerful (we remind of Church's Thesis, discussed, e.g., by Odifreddi [36]),

but it has two disadvantages that prevent the use of a TM as a device for efficiently performing calculations:

(i) The instructions of the TM are too primitive, their formulation is not intuitive in terms of semantically important actions. Given a set of instructions of some TM, it is very laborious to find out what this TM does.

(ii) The representation of information on the one-dimensional tape is adequate only in some cases. Usually the result of a calculation cannot be interpreted easily.

We alter the concept of a TM concerning those two issues, and the resulting machine is a **semantic virtual machine** (SVM):

Concerning item 1, the SVM is able to execute an **SVM program**, i.e., a sequence of commands written in an assembler-like language. Each command performs a comprehensible action on the memory.

Concerning item 2, the SVM represents information by semantic relations between objects represented by a binary operator, the **semantic memory**. Using the semantic memory, complex relations can be represented in a simple and user-friendly way, and be visualized as a directed, labeled graph. Thus an SVM allows the expression of semantics in a very natural form.

Altogether, we think of the SVM as a machine that performs some basic actions on the semantic memory. The SVM has random access to this memory, and the actions it performs (like writing, copying, deleting,...) are determined by a human-readable program.

That the SVM is as least as powerful as an ordinary Turing machine is shown in Section 3.7, but we give the SVM even more power by allowing it to access the capabilities of the physical device it is implemented on: external memory and external processors, see Section 3.4. This has the consequence that the SVM is no longer equivalent to an ordinary Turing machine, or in other words, not every SVM program, regarded as a function on the context, is Turing computable. For example, external processors might have access to the system clock etc. However, the main reason for enabling the SVM to call external processors is higher performance and reusability of trusted algorithms. The SVM command that calls external processes is essentially a foreign function interface (FFI) of the SVM.

A cornerstone in the creation of the SVM is the proof that the SVM is powerful enough to simulate itself in a very simple way. This is done by giving an SVM program that can simulate every other SVM program. Since this is analogous to the role of a universal Turing machine, we call this program the **universal semantic virtual machine** (USVM).

The USVM is a program short and transparent enough to be checked by hand. It has only 166 lines of code, see Section 3.8 (compare this, e.g.,

to the reflective interpreter by Jefferson & Friedman in [15], which has 273 lines). The USVM gives us a possibility to check 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.

All this makes the SVM a semantically self-contained, transparent and easily usable tool that can be a trustworthy foundation for any computer system that deals with semantic content.

3.1 The semantic virtual machine

A **semantic virtual machine** (SVM) is a machine manipulating semantic information in a semantic memory. Independent of the interpretation of the semantic memory either as semantic mapping or as graph, we will refer to it as the **memory** of the SVM.

Since there are equivalent formulations of Turing machines which use a 2-dimensional memory instead of the tape (a proof is given by Cohen [6]) the change to a binary operator instead of a tape alone would not go beyond the scope of a Turing machine. But by allowing the SVM to manipulate its external environment, the scope of an SVM becomes strictly bigger cf. Section 3.4.

The external values are handled exclusively by **external processors**, i.e., algorithms executed by the physical machine. External values are discussed in more detail in Section 3.4.

The memory of the SVM contains the **program** to execute, and the information about flow control as well, all represented via a semantic mapping. To enable the processing of more than one program in the same memory, each execution of the SVM has its own **core**, i.e., a record reserved for the input, the output and temporary data.

Since the core is the most important record for a program, will simplify the notation for it: We use the **caret** `^` to abbreviate reference to the core of the execution under consideration. Hence `^a` means `#core.a`, where `#core` is the core of the execution under consideration. The caret binds stronger than the semantic mapping, hence `a.^b` means `a.(#core.b)`.

To start processing a program, the SVM needs to know the object that contains the program, and the object that serves as the core. Therefore the call of an SVM program has two arguments: the name of the program and the core.

3.2 The SVM programming language

The most elementary part of the SVM programming language is a command. There are 24 different commands; a list of the commands and their action is given in Section 3.6. The commands fall into four groups: commands that structure the program but have no influence on the memory at runtime, commands for flow control, assignments, which make alterations in the memory of the SVM, and commands handling or external values.

Compared to transition tables of Turing machines, SVM programs are much less intricate. In fact, the SVM programming language is much more akin to an assembler-style language.

Before describing the commands in detail, we say something about the structure of the language and external processors and values. This is the content of this and the next section.

The SVM programming language has the reserved names

```
program         process         start
```

for structuring the program, and the reserved names

```
check           fields          refset          refin
create          get             set             refout
exist           goto            setconst        unset
existref        if              stop
external        move            in
externalref     refget          out
```

for commands making certain alterations in the memory or in flow control. The meaning of these names will be discussed in Section 3.3. The name `type` should also not be used as a name in a SVM program to prevent collusion with the typing system.

All other names and more general alphanumeric strings may be used as variables for objects.

The SVM programming language is the lowest level of a fully comfortable programming language that we are in the process of developing, see [34] and [16].

3.2.1 The grammar of the SVM programming language

The complete grammar of the SVM programming language is defined by the following grammar, using partially labeled, BNF like productions. A line beginning with a percent sign `%` is treated as a comment without any

effect on the program. To ease readability, white spaces at the beginning of a line are ignored.

We define the following macros in the grammar:

```
: macro(lines of $1)
macro: $1 | macro newline $1

: macro(string of $1)
macro: $1 | macro $1
```

The tokens BLANK, CHARACTER and ALPHANUMERIC in the grammar stand for a blank space, any character and any alphanumeric character respectively. The token COMMENT is a string beginning with a percent sign (`%`) and not containing a newline (`\n`).

STMPROGRAM	→	HEADER lines of PROCESS STARTPROCESS
HEADER	→	`program(` NAME `)`
NAME	→	string of ALPHANUMERIC
PROCESS	→	PROCESSHEADER lines of COMMAND PROCESSEND
PROCESSHEADER	→	`process(` NAME `)`
COMMAND	→	NC | GC | SC | string of BLANK COMMAND | COMMENT
PROCESSEND	→	GC | SC | string of BLANK PROCESSEND | COMMENT
STARTPROCESS	→	`start(` NAME `)`
NC	→	NAME `= check(` NAME `,` NAME `)`
NC	→	`create(` NAME `)`
NC	→	NAME `=exist(` NAME `,` NAME `)`
NC	→	NAME `=existref(` NAME `,` NAME `)`
NC	→	NAME `=external(` NAME `,` NAME `)`
NC	→	NAME `=externalref(` NAME `,` NAME `)`
NC	→	NAME `=fields(` NAME `)`
NC	→	NAME `=get(` NAME `,` NAME `)`
NC	→	`goto(` NAME `)`
NC	→	`if(` NAME `,` NAME `)`
NC	→	`move(` NAME `)`
NC	→	NAME `=refget(` NAME `,` NAME `)`
NC	→	`(` NAME `,` NAME `)=refset(` NAME `)`
NC	→	`(` NAME `,` NAME `)=set(` NAME `)`
NC	→	`(` NAME `,` NAME `)=setconst(` NAME `)`
NC	→	NAME `=in(` NAME `,` NAME `)`
NC	→	NAME `=out(` NAME `,` NAME `)`
NC	→	NAME `=refin(` NAME `,` NAME `)`
NC	→	NAME `=refout(` NAME `,` NAME `)`

NC → `unset( NAME , NAME )`
SC → `stop`

3.2.2 Representation of SVM programs in the SM

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.

Each process is represented in the memory by a linked list of commands, and the first command of each process is accessible by:

```
#program.#processname = #process
```

where `#program` is the record containing the program, `#processname` is the name of the process, and `#process` is a linked list of the commands of process `#processname`.

Each command is represented in the memory by a record `#command` with

```
#command.#part = #object

#command.next = #nextcommand
```

and `#nextcommand` is the command in the line below this command. The object `#part` is one of the following: `comm` refers to the name of the command, `arg1` to the first argument, `arg2` to the second argument, and `arg3` to the third argument. Since not all commands have three arguments, some of these may be empty.

The object `#program.start` contains the object referring to the first process, i.e., the process that has to be executed first in the program `#program`.

The SVM is untyped. However, to further specify the representation of SVM programs in the SM, we give typesheets for SVM programs:

```
SVM::

SvmProgram:
  allOf> start = Object
         processes = Processes
  nothingElse>

Processes:
```

```
  someOfType> Object = SvmCommand

CommandName:
  atomic> check, create, exist, existref, external, externalref
  atomic> fields, get, goto, if, move, refget, refset, set
  atomic> setconst, transportin, transportout, transportrefin
  atomic> transportrefout, unset, stop

SvmCommand:
  allOf> comm = CommandName
  optional> arg1 = Object
            arg2 = Object
            arg3 = Object
            next = SvmCommand
  nothingElse>
```

The semantic graph in Figure 3.1 is the record that represents the SVM program `copyFields` as given in text form in Section 3.6, page 33. Note that for transparency, the sems with field `type` are not printed.

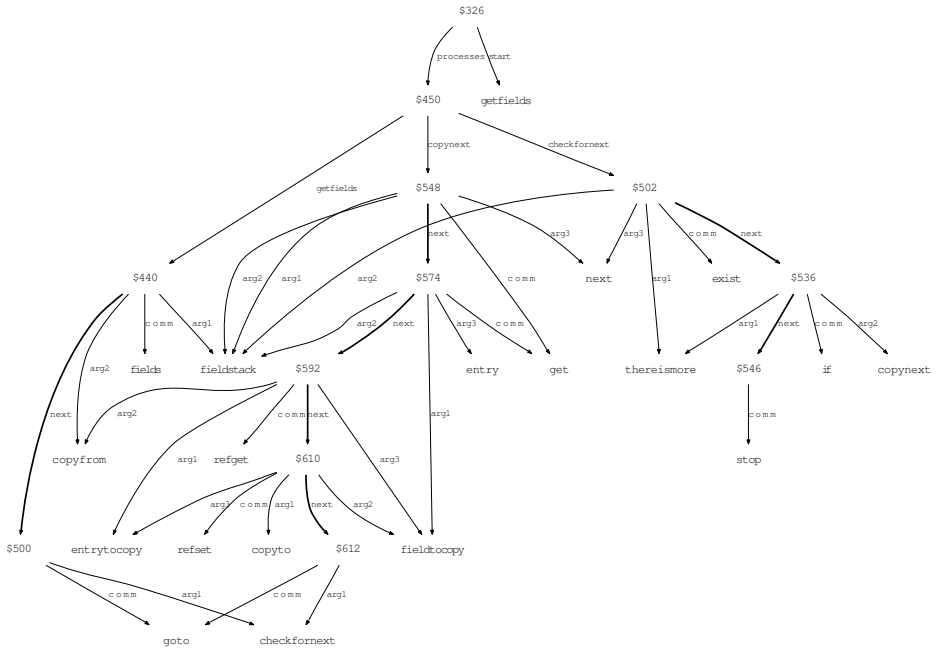


Figure 3.1: Representation of an SVM program in the SM

3.3 Flow control

This section describes how the information for flow control is represented in the memory of the SVM.

The SVM command currently executed is called the **focus**, it may change after each program execution. A process can be entered only at its first command, but it is possible to leave a process before its last command line is reached.

During runtime, the focus is represented in the object `^focus`. Setting the entry of `^focus` to `^focus.next` means to proceed one line forward in the program. Setting the entry of `^focus` to `#program.#process` as done by the `goto` and `if` command, sets the focus to the first line of the process `#process`.

It is assumed that `^core` always contains the current core, i.e., `#core.core=#core`. This allows us to reduce the number of different commands.

3.4 External values and external processors

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 existing algorithms written in different programming languages.

Every object can have an **external value**, which is some data associated to this object, but not part of the memory of the SVM. Instead, it is managed by the physical device which executes the SVM. In descriptions of commands, we refer to the external value of the object `#object` by `VALUE(#object)`.

The values of objects are directly processed by the physical device. Hence one can benefit from the full computational power of the physical device. A computation on the external values is said to be realized by an **external processor**. External processors have no access to the memory of the SVM, but may be called from the SVM as the command `external`. From a theoretical point of view, external processors are oracles to the SVM, as defined by Shoenfield [44].

External values can be imported into the memory of the SVM, and conversely. This is done by the commands `in`, `out`, `refin` and `refout`. 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 transport-commands. Figure 3.2 displays the interactions between the SVM and the physical device, and the SVM commands involved. Our current implementation includes protocols for representing

- strings,
- SVM programs (see Section 3.2),
- tapes and transition tables for Turing machines (see Section 3.7),
- Concise programs (see [16]).

There may be an arbitrary number of protocols, as long as the device on which the SVM is implemented knows how to interpret them.

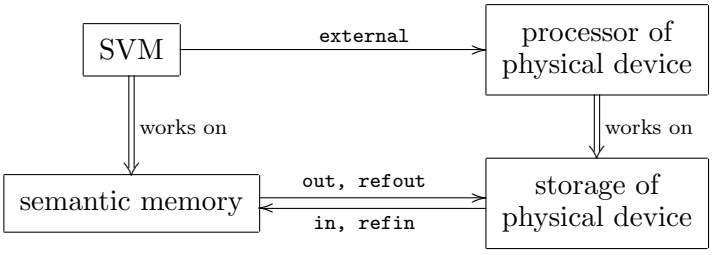


Figure 3.2: Interaction of the SVM with the physical device

3.5 An operational semantics for the SVM

We formally define the action of the SVM, in particular of every SVM command, by giving an operational semantics.

Let V the set of values.

A **state** is a triple (s, v, O) such that O is a sets of objects, $s : O \times O \to O$ is a semantic mapping, and $v : O \to V$ is a partial mapping that associates to some objects in O an external value. We define $\mathfrak{S}$ to be the set of states.

The object $p \in O$ denotes the record that contains the program to execute, and $c \in O$ denotes the core to use. In the following, we define $f := s(c, \texttt{focus})$.

For a semantic mapping s and a set $R \subseteq O \times O \times O$ we define replace(s, R) as a semantic mapping s' with

$$s'(h, f) := \begin{cases} e & \text{if } (h, f, e) \in R \\ s(h, f) & \text{otherwise} \end{cases}$$

For a mapping $v : O \to V$ and a set $R \subseteq O \times V$ we define replace(s, R) as mapping $v' \in O \to V$ with

$$v'(o) := \begin{cases} e & \text{if } (o, e) \in R \\ v(o) & \text{otherwise} \end{cases}$$

The execution of the SVM is a (possibly infinite) sequence of states $(S_1, S_2, \ldots, S_i, \ldots)$ determined by an initial state $S_0 = (s_0, v_0, O_0)$ and by $S_1 := \text{start}(S_0)$ and $S_{t+1} := \text{step}(S_t)$ for $t = 1, 2, \ldots$. The function start : $\mathfrak{S} \to \mathfrak{S}$ is defined by $\text{start}(s, v, O) = (\text{replace}(s, R), v, O)$ with $R := \{(c, \texttt{focus}, s(p, \texttt{start})), (c, \texttt{core}, c)\}$. The function step : $\mathfrak{S} \to \mathfrak{S}$ is defined by a case distinction over $s(f, \texttt{comm})$, as follows:

check: If $s(f, \texttt{comm}) = \texttt{check}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with

$$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), \texttt{True})\}$$

if $s(f, \texttt{arg2}) = s(f, \texttt{arg3})$ and

$$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), \texttt{False})\}$$

if $s(f, \texttt{arg2}) \neq s(f, \texttt{arg3})$

create: If $s(f, \texttt{comm}) = \texttt{create}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O \cup \{o\})$ with
$o \notin O$, and $R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), o)\}$.

exist: If $s(f, \texttt{comm}) = \texttt{exist}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with

$$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), \texttt{True})\}$$

if $s(s(c, s(f, \texttt{arg2})), s(f, \texttt{arg3})) \neq \texttt{Empty}$ and

$$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), \texttt{False})\}$$

if $s(s(c, s(f, \texttt{arg2})), s(f, \texttt{arg3})) = \texttt{Empty}$

existref: If $s(f, \texttt{comm}) = \texttt{existref}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with

$$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), \texttt{True})\}$$

if $s(s(c, s(f, \texttt{arg2})), s(c, s(f, \texttt{arg3}))) \neq \texttt{Empty}$ and

$$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), \texttt{False})\}$$

if $s(s(c, s(f, \texttt{arg2})), s(c, s(f, \texttt{arg3}))) = \texttt{Empty}$

external: If $s(f, \texttt{comm}) = \texttt{external}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v', O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next}))\}$
and $v'(s(c, s(f, \texttt{arg1})))$ is the value calculated by the external process associated to $s(f, \texttt{arg2})$ with input $v(s(c, s(f, \texttt{arg3})))$ and $v' = v$ for all other objects.

externalref: If $s(f, \texttt{comm}) = \texttt{external}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v', O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next}))\}$
and $v'(s(c, s(f, \texttt{arg1})))$ is the value calculated by the external process associated to $s(c, s(f, \texttt{arg2}))$ with input $v(s(c, s(f, \texttt{arg3})))$ and $v' = v$ for all other objects.

fields: Let $(e_1, \ldots, e_n)$ a sequence containing every field of $s(c, s(f, \texttt{arg2}))$. If $s(f, \texttt{comm}) = \texttt{fields}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O \cup \{o_1, \ldots, o_n\})$ with
$o_i \notin O$ for all $i = 1, \ldots, n$, and

$$\begin{aligned} R = &\{(c, \texttt{focus}, s(f, \texttt{next})), (s(s(c, s(f, \texttt{arg1})), \texttt{next}), o_1)\} \\ &\cup \{(o_i, \texttt{entry}, e_i) \mid i = 1, \ldots, n\} \\ &\cup \{(o_i, \texttt{next}, o_{i+1}) \mid i = 1, \ldots, n-1\} \end{aligned}$$

get: If $s(f, \texttt{comm}) = \texttt{get}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), s(s(c, s(f, \texttt{arg2})), s(f, \texttt{arg3})))\}$.

goto: If $s(f, \texttt{comm}) = \texttt{goto}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(p, s(f, \texttt{arg1})))\}$.

if: If $s(f, \texttt{comm}) = \texttt{if}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(p, s(f, \texttt{arg2})))\}$ if $s(c, s(f, \texttt{arg1})) = \texttt{True}$
$R = \{(c, \texttt{focus}, s(f, \texttt{next}))\}$ if $s(c, s(f, \texttt{arg1})) = \texttt{False}$
For every other value of $s(c, s(f, \texttt{arg1}))$ the SVM stops and issues an error.

move: If $s(f, \texttt{comm}) = \texttt{move}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(p, s(f, s(c, \texttt{arg1}))))\}$.

refget: If $s(f, \texttt{comm}) = \texttt{refget}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with

$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (c, s(f, \texttt{arg1}), s(s(c, s(f, \texttt{arg2})), s(c, s(f, \texttt{arg3}))))\}$

refset: If $s(f, \texttt{comm}) = \texttt{refset}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (s(c, s(f, \texttt{arg1})), s(c, s(f, \texttt{arg2})), s(c, s(f, \texttt{arg3})))\}$

set: If $s(f, \texttt{comm}) = \texttt{set}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (s(c, s(f, \texttt{arg1})), s(f, \texttt{arg2}), s(c, s(f, \texttt{arg3})))\}$.

setconst: If $s(f, \texttt{comm}) = \texttt{setconst}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (s(c, s(f, \texttt{arg1})), s(f, \texttt{arg2}), s(f, \texttt{arg3}))\}$.

stop: If $s(f, \texttt{comm}) = \texttt{stop}$, then $\text{step}(s, v, O)$ is not defined and the sequence of states ends.

in: If $s(f, \texttt{comm}) = \texttt{in}$, then $\text{step}(s, v, O) = (s', v, O \cup \{o_1, \ldots, o_n\})$ with
$s'(c, s(f, \texttt{arg1}))$ is the semantic mapping that represents the value $v(s(c, s(f, \texttt{arg3})))$ in a record with handle $s(c, s(f, \texttt{arg1}))$ according to the protocol associated with $s(f, \texttt{arg2})$, using objects $\{o_1, \ldots, o_n\} \notin O$, and $s'(c, \texttt{focus}) = s(f, \texttt{next})$.

refin: If $s(f, \texttt{comm}) = \texttt{refin}$, then $\text{step}(s, v, O) = (s', v, O \cup \{o_1, \ldots, o_n\})$ with
$s'(c, s(f, \texttt{arg1}))$ is the semantic mapping that represents the value $v(s(c, s(f, \texttt{arg3})))$ in a record with handle $s(c, s(f, \texttt{arg1}))$ according to the protocol associated with $s(c, s(f, \texttt{arg2}))$, using objects $\{o_1, \ldots, o_n\} \notin O$, and $s'(c, \texttt{focus}) = s(f, \texttt{next})$.

out: If $s(f, \texttt{comm}) = \texttt{out}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), \text{replace}(v, Q), O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next}))\}$
$Q = \{(s(c, s(f, \texttt{arg1})), e)\}$ and e is the value that represents the record with handle $s(c, s(f, \texttt{arg3}))$ according to the protocol associated with $s(f, \texttt{arg2})$

refout: If $s(f, \texttt{comm}) = \texttt{refout}$, then
$\text{step}(s, v, O) = (\text{replace}(s, R), \text{replace}(v, Q), O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next}))\}$
$Q = \{(s(c, s(f, \texttt{arg1})), e)\}$ and e is the value that represents the record with handle $s(c, s(f, \texttt{arg3}))$ according to the protocol associated with $s(c, s(f, \texttt{arg2}))$

unset: If $s(f, \texttt{comm}) = \texttt{unset}$, then $\text{step}(s, v, O) = (\text{replace}(s, R), v, O)$ with
$R = \{(c, \texttt{focus}, s(f, \texttt{next})), (s(f, \texttt{arg1}), s(f, \texttt{arg2}), \texttt{Empty})\}$.
For every other value of $s(f, \texttt{comm})$, the execution of the SVM stops and issues an error.

3.5.1 Garbage collection

For some state $S = (s, v, O) \in \mathfrak{S}$ we define the relation $o_1 \rhd_S o_2$ if there exists an object $x \neq \texttt{Empty}$ with either $s(o_1, x) = o_2$ or $s(o_1, o_2) = x$.
We define $o_1 \rhd\rhd_S o_2$ if there exists a sequence $(x_1, \ldots, x_n)$ of objects with $o_1 \rhd_{s_t} x_1 \rhd_S \ldots \rhd_{s_t} x_n \rhd_S o_2$.
For a set X of objects, we define $X \rhd\rhd_S o$ if there exists an $x_i \in X$ with $x_i \rhd\rhd_S o$.

Theorem (Garbage collection) For given state S and $p, c \in O$, we define $O^{\text{ess}} = \{o \in O | \{c, p\} \rhd\rhd_S o\}$.
Let s^{ess} and v^{ess} the restrictions of s and v to O^{ess}, let $S^{\text{ess}} = (s^{\text{ess}}, v^{\text{ess}}, O^{\text{ess}})$.
Then

$$\text{step}(S) = \text{step}(S^{\text{ess}})$$

Proof: It is easy to check that in all cases of the definition of step, the result $\text{step}(S)$ only depends on objects o with $\{c, p\} \rhd\rhd_S o$. □
This theorem allows us to perform garbage collection after every step of the SVM, i.e., delete every sem `a.b=c` with `a`$\notin O^{\text{ess}}$, and delete every object `o`$\notin O^{\text{ess}}$, and its value.

3.6 Description of the SVM commands

We now introduce the commands of the SVM language and describe their effect. There are four groups of commands: Table 3.2 describes the commands that are needed to give the program an appropriate structure. Table 3.3 contains the assignments, i.e., those commands that perform alterations in the memory of the SVM. Table 3.4 gives the commands used for flow control, and Table 3.5 the commands that establish communication with the physical device, namely call external processes and access external values.

3.6.1 Example: shallow copy

The following SVM program performs a shallow copy from the record passed to the SVM in `#core.copyfrom` to the object in `#core.copyto`.

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 3.2: Structuring commands

SVM command	comment
`#1=check(#2,#3)`	sets `^#1` to `True` if `^#2 = ^#3`, else to `False`
`create(#1)`	assigns some free object to `^#1`
`#1=exist(#2,#3)`	sets `^#1` to `True` if `^#1.#2` exists, else to `False`
`#1=existref(#2,#3)`	sets `^#1` to `'T` if `^#1.^#2` exists, else to `False`
`#1=fields(#2)`	creates a linked list with handle `^#1` containing all fields of `^#2`
`#1=get(#2,#3)`	assigns `^#2.#3` to `^#1`
`#1=refget(#2,#3)`	assigns `^#2.^#3` to `^#1`
`(#1,#2)=set(#3)`	assigns `^#3` to `^#1.#2`
`(#1,#2)=refset(#3)`	assigns `^#3` to `^#1.^#2`
`(#1,#2)=setconst(#3)`	writes `#3` to `^#1.#2`
`unset(#1,#2)`	deletes the position `(^#1,^ #2)`, i.e., sets it to `Empty`

Table 3.3: Assignment commands

SVM command	comment
`goto(#1)`	sets the focus to the first line of process `#1`
`move(#1)`	sets the focus to the first line of process `^#1`
`if(#1,#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

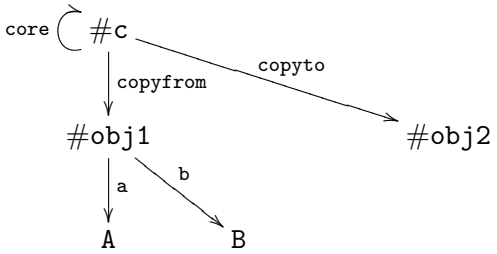
Table 3.4: Commands for flow control

SVM command	comment
`#1=external(#2,#3)`	calls external processor `#2` with input `VALUE(^#3)` and output `VALUE(^#1)`
`#1=externalref(#2,#3)`	calls external processor `^#2` with input `VALUE(^#3)` and output `VALUE(^#1)`
`#1=in(#2,#3)`	imports `VALUE(^#2)` into `^#1` by protocol `#3`
`#1=out(#2,#3)`	exports record `^#2` into `VALUE(^#1)` by protocol `#3`
`#1=refin(#2,#3)`	imports `VALUE(^#2)` into `^#1` by protocol `^#3`
`#1=refout(#2,#3)`	exports record `^#2` into `VALUE(^#1)` by protocol `^#3`

Table 3.5: Commands for external communication

```
program(copyFields)
process(getfields)
  fieldstack=fields(copyfrom)
  goto(checkfornext)
process(checkfornext)
  thereismore=exist(fieldstack,next)
  if(thereismore,copynext)
  stop
process(copynext)
  fieldstack=get(fieldstack,next)
  fieldtocopy=get(fieldstack,entry)
  entrytocopy=refget(copyfrom,fieldtocopy)
  (copyto,fieldtocopy)=refset(entrytocopy)
  goto(checkfornext)
start(getfields)
```

For example, if the SVM is called with the program above and core `#c` and a semantic memory containing the following sems:



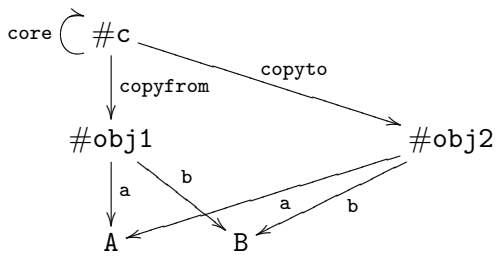
Upon execution of this program the following happens:

First, the process `getfields` is entered (due to the line `start(getfields)`), and this process generates a linked list beginning in `^fieldstack` such that each field `#f` of `^copyfrom` is represented as `^fieldstack.next`^{k}`.entry=#f` for some $k \in 0, 1, 2, \ldots$. Then, process `checkfornext` is entered.

The process `checkfornext` checks if `^fieldstack.next` is nonempty. If this is the case, process `copynext` is entered; otherwise the SVM halts.

Process `copynext` first sets `^fieldstack` to `^fieldstack.next`, i.e., we advance one link in the linked list that contains all the fields of `^copyfrom`. Then the field of `^copyfrom` (stored in `^fieldstack.entry`) and the entry `^copyfrom.^fieldstack.entry` are stored in `^fieldtocopy` and `^entrytocopy` respectively. Finally this field and entry are written into a new sem with handle `^copyto`.

The result after execution of the SVM is a semantic memory containing:



3.7 Turing machines and their simulation

In this section we formally introduce Turing machines and show that the SVM is Turing complete by giving an SVM program that simulates any ordinary Turing machine.

3.7.1 The tape of the Turing machine

The cells on the tape of the Turing machine are simulated by a linked list in `^context.tape` where `^context.tape.entry` is the left end of the tape, and initially holds the delimiting symbol "`>`". Objects that do not exist are interpreted by the Turing machine as blank spaces. The alphabet of the Turing machine is arbitrary, but must not contain the pipe `|`. Furthermore, the characters `>` and the blank space are reserved. At the beginning of execution, the head of the TM is assumed to be on `^context.tape.next`, and the TM to be in state "1".

3.7.2 The instructions of the Turing machine

The action of the Turing machine is determined by a finite list of instructions of the form

$$S \mid R \mid W \mid M \mid S'$$

which applies if S is the state the TM is currently in, and R is the symbol currently read by the TM. In this case, the following actions are performed, in this order:

(i) The symbol W is written. If W is the empty string, then nothing is written.

(ii) The head moves one cell to the right if $M =$ `R` and one cell to the left if $M =$ `L`. If M is the empty string, then no movement is performed.

(iii) The state of the TM changes to S'.

If no instruction applies, the TM halts.

3.7.3 Example: Replacement

This is an example of a very simple Turing machine, which simply replaces in a string of `a`'s and `b`'s every occurring `a` by `c`. It has only one state, and runs through the tape from left to right, replacing every `a` it runs along. When it reaches the end of the string, it reads a blank, and no instruction applies, hence the Turing machine halts.

The two instructions are:

```
1|b||R|1
1|a|c|R|1
```

Thus, the tape with initial content

```
> b b b a b b a b b a
```

has, when the Turing machine halts, the content:

```
> b b b c b b c b b c
```

3.7.4 Example: Division with remainder

This is a more complicated example of a Turing machine that performs a division with remainder. The number of `a`'s at the beginning of the tape is divided by the number of the `b`'s following. The result is represented as the number of `q`'s for the quotient, and the number of `r`'s for the remainder, when the Turing machine halts.

For example, if we want to perform 8 divided by 3, the tape initially looks like this:

```
> a a a a a a a a b b b
```

where the eight `a`'s represent the dividend and the three `b`'s the divisor.

When the Turing machine halts, the tape contains:

```
> A A A A A A A A b b b q q r r
```

which tells us that the quotient is 2 (represented by the two `q`'s), and the remainder is also 2 (represented by the two `r`'s). Note that the `a`'s have changed to `A`'s in order for the processed `a`'s to be distinguishable from those not yet processed.

This is the transition table of the Turing machine that performs a division with remainder:

```
1|a||R|1   3|a||R|3   4|b||L|2   6|B|b|L|6  7|q||R|7   9|B||R|9
1|b||L|2   3|A||R|3   4|q|||5    6|A|||2    7| ||L|8   9|b||R|9
2|B||L|2   3|B||R|3   4| |||5    6|a|||2    8|q||L|8   9|q||R|9
2|A||L|2   3|b|B|R|4  5|q||R|5   7|A||R|7   8|b||L|8   9|r||R|9
2|a|A||3   3| |||5    5| |q||6   7|B||R|7   8|r||L|8   9| |r||8
2| ||R|7   3|q|||5    6|q||L|6   7|b||R|7   8|B|b||9
```

This Turing Machine performs the division by the following steps:

State 1 just brings the head in the right position to start:
The head moves to the right until the first `b` is read, then moves one cell to the left and enters state 2.

State 2, 3, 4 and 5 determine the quotient, i.e., the number of `q`'s on the tape: For every `b`, an `a` is replaced by an `A`, and the `b` is replaced by a `B`. If there is no more `b` on the tape, one `q` is written and the `B`'s are replaced by `b`'s:
In **state 2**, the head is moved to the left, until the rightmost `a` is reached. If there is no `a` on the tape (because all `a`'s have been replaced by `A`'s to mark them as processed), the TM changes to state 7. Else the rightmost `a` is changed to `A`, and the TM enters state 4. **State 3** then replaces the leftmost `b` by a `B` and changes to state 4. If there is no `b` on the tape (every `b` has been replaced by a `B`), the TM changes to state 5. **State 4** is only a case distinction: if the `b` just replaced was the last one, then change to state 5, and if there is still a `b` on the tape, then change to state 2, i.e., perform a loop. **State 5** moves the head to the right until it reaches an empty cell, writes a `q`, and changes to state 6.

State 6 changes all `B`'s back to `b`'s and puts the head on the rightmost cell containing either `A` or `a`. The TM is then set to state 2 again and this is repeated (and every time a `q` is written) until there are no more `a`'s on the tape, and state 7 is entered.

State 7 then simply puts the head to the last nonempty cell of the tape

and changes to state 8.

States 8 and 9 determine the remainder of the division (i.e., `b`'s which has not been replaced by `B`'s in state 3) and write the corresponding number of `r`'s to the tape:
The head is put to the rightmost `B` by **state 8**, and before entering state 9, this `B` is replaced by a `b`. If there is no `B` on the tape, then no instruction applies and the TM halts. In **state 9**, the head is put to the first empty cell of the tape, an `r` is written there, and the TM enters state 8 again, hence loops.

There has been put much effort in constructing smaller and smaller **universal Turing machines**, i.e., special Turing machines that can simulate every other Turing machine, from the 1950's until today[1]. Small universal Turing machines were studied, e.g., in the influential paper [30] by MINSKY, by ROBINSON [39], and recently by NEARY & WOODS [31].

Apparently no effort was put in making universal Turing machines user-friendly while keeping them small, which would be related to the goal of our work.

3.7.5 The SVM-code of a universal TM

To prove Turing completeness we now specify a SVM program that simulates an arbitrary Turing machine. The reader should interpret it with the help of Tables 3.2 – 3.5 .

The instructions of the Turing machine are represented in the memory of the SVM as follows:

`^transtable`	contains the linked list of instructions
`^transtable.next`^{k}`.state`	contains S of the kth instruction
`^transtable.next`^{k}`.reading`	contains R of the kth instruction
`^transtable.next`^{k}`.towrite`	contains W of the kth instruction
`^transtable.next`^{k}`.tomove`	contains M of the kth instruction
`^transtable.next`^{k}`.tostate`	contains S' of the kth instruction
`^transtable.next`^{k}`.next`	contains the $k+1$th instruction

Note that if W in the instruction is the empty string, `#obj.towrite` is set to `#obj.reading`, hence no alteration is done by writing. If M in the instruction is the empty string, `#obj.tomove` is set to `X`, just to distinguish it from `L` and `R`. The position of the head is stored in `^position`, and the state of the Turing Machine is stored in `^state`.

```
program(UTM)
process(init)
```

[1] In October 2007, Alex Smith claimed to have found the smallest Universal Turing Machine possible, having 2 states and 3 symbols, see `www.wolframscience.com/prizes/tm23/solved.html`

```
% makes the necessary nodes available in the core
    tape=in(tapename,tm_tape)
    transtable=in(tablename,tm_instructions)
    position=get(core,tape)
    (core,state)=setconst(1)
    (core,cR)=setconst(R)
    (core,cL)=setconst(L)
    (core,blank)=setconst( )
    goto(nextcommand)
process(nextcommand)
% initializes the comparing, reads the tape
    trycommand=get(core,transtable)
    reading=get(position,entry)
    goto(checkstate)
process(trynext)
% halts the TM if there are no instructions left
    therearemorecommands=exist(trycommand,next)
    if(therearemorecommands,increase)
    goto(endprocess)
process(increase)
% go to next command
    trycommand=get(trycommand,next)
    goto(checkstate)
process(checkstate)
% checks if the state of the TM is equal to the state in the command
    stateincommand=get(trycommand,state)
    samestate=check(state,stateincommand)
    if(samestate,checksymbol)
    goto(trynext)
process(checksymbol)
% checks if the symbol read by the TM is equal to
% the symbol in the command
    symbolincommand=get(trycommand,reading)
    samesymbol=check(symbolincommand,reading)
    if(samesymbol,executecommand)
    goto(trynext)
process(executecommand)
% executes the instructions in the command
    state=get(trycommand,tostate)
    towrite=get(trycommand,towrite)
    (position,entry)=set(towrite)
    tomove=get(trycommand,tomove)
    moveleft=check(tomove,cL)
    moveright=check(tomove,cR)
```

```
    if(moveleft,left)
    if(moveright,right)
    goto(nextcommand)
process(left)
% moves the head to the left
    notleftend=exist(position,last)
    if(notleftend,goleft)
    create(newfield)
    (position,last)=set(newfield)
    (newfield,entry)=set(blank)
    (newfield,next)=set(position)
    position=get(position,last)
    goto(nextcommand)
process(goleft)
    position=get(position,last)
    goto(nextcommand)
process(right)
% moves the head to the right
    notrightend=exist(position,next)
    if(notrightend,goright)
    create(newfield)
    (position,next)=set(newfield)
    (newfield,entry)=set(blank)
    (newfield,last)=set(position)
    position=get(position,next)
    goto(nextcommand)
process(goright)
    position=get(position,next)
    goto(nextcommand)
process(endprocess)
    tapeasvalue=out(tape,tm_tape)
    copyoftape=external(valuecopyname,tapeasvalue)
    filename=external(writetofilename,copyoftape)
    stop
start(init)
```

When invoked, it expects to have the following objects in its core:

`^tapename`	is an object that has a tape as external value,
`^tablename`	is an object that has a transition table as external value,
`^valuecopyname`	is an object associated to an external processor that makes a copy of the external value,
`^filename`	is an object that has a valid filename as external value,

`^writetofilename`	is an object associated to an external processor that writes a type of into a file

The SVM-program `UTM` essentially searches for a command that applies, then performs the instructions, and then loops. In more detail,

`init`	initially sets up the records so that processing can begin, including loading the tape (stored as external value in `^tapename`) and the transition table (stored as external value in `^tablename`)
`nextcommand`	resets the records and replaces an empty cell on the tape by a blank.
`trynext`	halts if there is no next instruction and else calls the process
`increase`	which brings the next instruction into consideration.
`checkstate`	and
`checksymbol`	compares the state of the Turing Machine with S in the instruction, and the symbol currently read on the tape with R in the instruction, respectively. In other words, these two processes check if the instruction under consideration applies.
`executecommand`	performs the actions given in the instruction, except for moving the head to the left or the right. Since `^position` contains a counter representing the position of the head on the tape,
`left`	and
`goleft`	move the head to the left (if the head is not already on the leftmost cell), while
`right`	and
`goright`	move the head to the right.
`endprocess`	then writes the result first as the value of `^tapeasvalue`, then makes a copy of that value to `^copyoftape` (which is redundant, but serves for an example for an external processor), and then the value is written into the file that is the value of `^filename`.

Correctness is straightforward to prove.

The SVM program above simulates an arbitrary TM and does not use external storage. Since an ordinary TM has no external storage and it is not specified how an external processor should behave, it is impossible to give an ordinary TM that simulates an arbitrary SVM program.

3.8 The universal semantic virtual machine

A universal SVM (USVM) is the semantic analogon to a universal Turing machine. It is a special SVM program capable of 'simulating' the processing of an other SVM program `P` in the following sense: 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.

When the USVM is started, objects for `program`, `context` and `library` have to be passed to the USVM as part of its core. It is assumed that this information is stored in the objects `^sim_prog`, `^sim_context` and `^sim_lib` before calling the USVM.

The SVM code of the USVM is given in Appendix A.2. Without blank lines and comment lines, the USVM contains 166 lines.

Chapter 4

Typing

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.

Types can be defined using plain text documents called **type sheets**. Example type sheets can be found in the Appendix. Tables 4.2 and 4.1 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.

In order to provide a good basis for the semantical analysis of mathematical language expressing arbitrary mathematical content, a system needs to represent mathematics specified in a (perhaps controlled) natural language. Thus a concept of typing is needed that covers

(i) syntactically correct mathematical formulas,

(ii) well-formed sentences built according to a linguistic grammar, and

(iii) structured records in the programming sense.

The typing must be such that, using an appropriate type system, one can define and easily check their well-formedness. In particular, grammatical categories must be representable in the type system. To serve as a foundation in the sense of the FMathL framework [33], everything is set up in a way

that it can reflect itself without the need to augment the basic structure.

The essential formal structure achieving this is the introduction of a **semantic memory** as the abstract representation medium, **categories** as the fundamental structuring concept, and of **types** as a particular form of categories.

Comparison with the type system of XML. Typing in FMathL and typing in XML bear significant similarities, most notably with DTD and Relax NG [5]. (For a comparison of DTD, Relax NG and other XML schemas, see Lee & Chu [26].) 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.

4.1 The type structure

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, as we do not want to examine the whole graph every time a record is used. For this reason we define a procedure to determine that a given record is **well-typed** of a certain type, or **ill-typed**. These assignments are always made with respect to a particular type system.

A **type system** is a set of objects which are called the **categories** of that type system. The object `Empty` is never a category.

In the following, until mentioned otherwise, we always consider an arbitrary but fixed type system and its associated order relations. The set of categories in a type system is ordered by an irreflexive partial order relation $<$. If for the categories `#C1` and `#C2` the relation `#C1` $<$ `#C2` holds, we say that `#C1` is a **subtype** of `#C2`.

We define the relation $<<$ to be the reflexive and transitive closure of the relation $<$, i.e., `#C1` $<<$ `#C2` if either `#C1` $=$ `#C2` or there exist categories $\#\mathtt{c}_1, \ldots, \#\mathtt{c}_n$ such that `#C1` $< \#\mathtt{c}_1 < \ldots < \#\mathtt{c}_n <$ `#C2`. If `#C1` $<<$ `#C2` we say that `#C2` **contains** `#C1`.

A category is called a **type** if it is minimal in the ordering $<<$, and a **union** otherwise. Each type is either the **default type** `Object`, an **atomic type**, or a **proper type**.

Objects of an atomic type have no constituents; they 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.

4.1.1 The type of an object and matching

Every object #`obj` has a **type**, defined by the following rules:

(i) If #`obj.type` is a proper type, then the type of #`obj` is #`obj.type`.

(ii) If #`obj.type` = `Empty`, and #`obj` is an atomic type, then the type of #`obj` is #`obj`.

(iii) Otherwise, the type of #`obj` is `Object`.

The fact that the type of an atomic type is object itself is the reason why we use the word "atomic type" and "atomic object" (or just "atomic") synonymously.

We say that an object #`obj` **matches** a category #`C`, in symbols: **m**(#`obj`/#`C`) if either #`C` = `Object`, or #`T` << #`C` for #`T` the type of #`obj`. Note that since `Empty` is not a category, no object matches `Empty`. Note also that which type matches which category depends on the type system used. Thus in an implementation, the type system appears as an extra argument.

For the basic type structure as presented here, the naming convention is to use names with an upper case initial for categories (and hence for types), but names with a lower case initial for fields unless they are also names of categories. Non initial letters are capitalized if they represent the first letter of an independent word in the name. (This is sometimes called "camel case" or "medial capitals".) Users who define their own type systems are of course not bound to this convention.

4.2 Type sheets

Categories can be defined by text called a **type declaration**. A document that contains one or more type declarations is called a **type sheet**.

The first line of a type sheet declares the name of the type system to be specified. Every of the following lines either creates a new category (via the name of the category followed by a colon), or specifies the category, by a keyword possibly followed by further specifications.

4.2.1 Proper types

A type declaration of a proper type has the following structure:

(i) the name of the proper type (followed by a colon)

(ii) then optional other proper types (each followed by a `+`) to inherit requirements from

(iii) then lines of requirements starting with certain keywords listed in Table 4.1 followed by a greater sign (`>`) and together with some arguments.

In (ii), the final `+` is missing if no lines of the form (iii) follow.

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 4.1: Keywords in declarations of proper types

Example. We give a simple type declaration of some proper type `Norm`, to get acquainted with the syntax and the meaning of a type declaration. The type declaration

```
Norm:
allOf> entry=Expression
optional> index=Expression
```

expresses that any object #`obj` with #`obj.type` = `Norm` is required to have a constituent #`obj.entry` = #`e` with #`e` matching type `Expression`, and optionally it may have a sem #`obj.index` = #`i` with #`i` also matching type `Expression`.

4.2.2 Unions and atomics

Since a type declaration for a union may also declare a number of objects as atomic types, they are treated together in this subsection.

A type declaration of a union has the following structure:

(i) the name of the union type (followed by a colon)

(ii) followed by lines starting with certain keywords listed in Table 4.2 possibly followed by further specifications.

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 4.2: Keywords in declarations of unions and atomics

4.2.3 Inheritance

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
```

4.2.4 Templates

Assume a type declaration of a declared type #D containing the line `template> #C`. In this case, #T is called the **template** of #D. All the requirements form #T apply to #D, and additionally the requirements for #D.

There are several differences to inheritance:

- The SM stores the fact that the template of #D is #T, while inheritance is visible only on the type sheet level but (without closer analysis) not in the SM.
- Only proper types can have templates, while inheritance is also defined for unions.
- The proper type and the template may pose requirements on the same constituent.
- A proper type has at most one template of, while inheritance from multiple proper types is possible.

Templates are important for efficient programming with records. Indeed, graph walkers may handle all types with the same template using a single program rather than one for each such type; see [8]. If a type declaration does not specify a template, then the type is assumed to be its own template.

4.2.5 A grammar for type sheets

A text document containing a number of type declarations is called a **type sheet**. The first line of a type sheet contains the type system it defines or enlarges. Every line in a type sheet beginning with an exclamation mark (`!`) is a comment.

The following context-free grammar defines type sheets as the sentences derivable from the starting symbol TYPESHEET. The token COMMENT is an arbitrary string beginning with an exclamation mark (`!`) and not containing a newline (`\n`).

TYPESHEET	→	HEADER BODY
HEADER	→	NAME `::`
BODY	→	LINE | BODY `\n` LINE
LINE	→	UNION | DECLARED | `\n` | COMMENT `\n`
UNION	→	UNIONHEADER `\n` UNIONLINES
UNIONHEADER	→	NAME `:`
UNIONLINES	→	`union>` NAMESEQ | `atomic>` NAMESEQ | `complete>` | `index>` EQUATIONLINES
DECLARED	→	DECHEADER `\n` DECLINES
DECHEADER	→	NAME `:` | NAME `:` NAMESUM `+` | NAME `:` NAMESUM
DECLINES	→	DECKEYWORD `>` EQUATIONLINES | `itself>` NAMELINES | `template>` NAME | `nothingElse>` | `nothing>`
DECKEYWORD	→	`allOf` | `oneOf` | `someOf` | `optional` | `someOfType` | `fixed` | `array` | `index` | `only`
EQUATIONSEQ	→	EQUATION | EQUATIONSEQ `,` EQUATION
EQUATIONLINES	→	EQUATION | EQUATIONSEQ `\n` EQUATION
EQUATION	→	NAME `=` NAME
NAMESEQ	→	NAME | NAMESEQ `,` NAME
NAMELINES	→	NAME | NAMESEQ `\n` NAME
NAMESUM	→	NAME | NAMESEQ `+` NAME
NAME	→	A-z | NAME A-z | NAME 0-9

Each line contains a production with a nonterminal on the left side of the arrow (→), and a disjunction of strings of terminals and/or nonterminals on the right side, separated by a pipe (|). All words in capital letters are nonterminals, A-z and 0-9 denote the letters and digits respectively, `\n` denotes the "newline" character, and all other non blank characters – in particular, `>`, `:`, `+`, `=`, `(`, `)` and `,` denote themselves.

4.2.6 Example: Typesheets for the SVM

The SVM is untyped. However, to further specify the representation of SVM programs in the SM, and to give examples for type sheets, we give type sheets for SVM programs:

```
SVM::

SvmProgram:
  allOf> start = Object
         processes = Processes
  nothingElse>
```

```
Processes:
  someOfType> Object = SvmCommand

CommandName:
  atomic> check, create, exist, existref, external, externalref
  atomic> fields, get, goto, if, move, refget, refset, set
  atomic> setconst, transportin, transportout, transportrefin
  atomic> transportrefout, unset, stop

SvmCommand:
  allOf> comm = CommandName
  optional> arg1 = Object
            arg2 = Object
            arg3 = Object
            next = SvmCommand
  nothingElse>
```

4.2.7 Consistency of type sheets

Assume that the type declaration of proper type #D specifies a template #T. A program that reads the type sheet must perform the following consistency checks:

- All atomic types declared in a type sheet have to be declared as subtypes of a union `Atomic`.
- If a union declares atomic types, these must not already exist.
- The order relation $<$ must be irreflective.
- When using a template, the added requirements are actually restrictions of the template.
- In a type declaration the left-hand sides following a particular operator (including those types declarations to inherit from) have to be unique.

If any of these requirements is violated, the type sheet reader issues an error.

4.3 Type declarations in the SM

We will now describe informally which requirements each keyword in a type declaration poses on the object that has this type. We also define how type declarations and type systems are represented in the semantic memory.

4.3.1 Type declarations of proper types

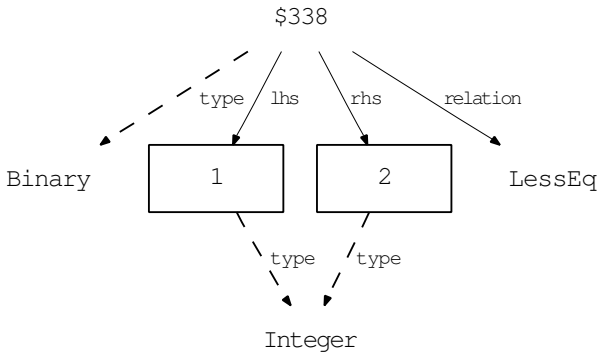
allOf

Via `allOf`, we require an object to have all of a collection of fields with entries of a certain kind.

Example. Consider a category `binary`, which we want to use to represent binary relations, e.g., in the representation of

$$1 \leq 2,$$

given in the following semantic graph:



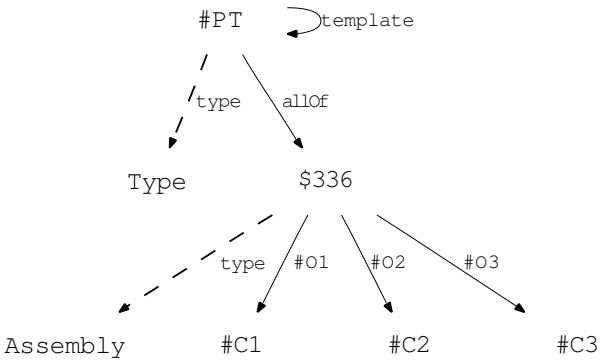
We require constituents with fields both `lhs` and `rhs` and entries of type `Integer`, and we require a constituent with field `relation` and an entry which matches the type `RelationAtomic`
(assuming `LessEq` << `RelationAtomic`).
We can express these restrictions via the following type declaration:

```
LessEq:
allOf> lhs=Integer
       rhs=Integer
       relation=RelationAtomic
```

Representation in the SM. Consider a proper type #`PT` using `allOf`:

```
Test(TypeSystem)::

#PT:
allOf> #O1=#C1
       #O2=#C2
       #O3=#C3 ! etc.
```

This is stored in the SM as the following semantic graph:

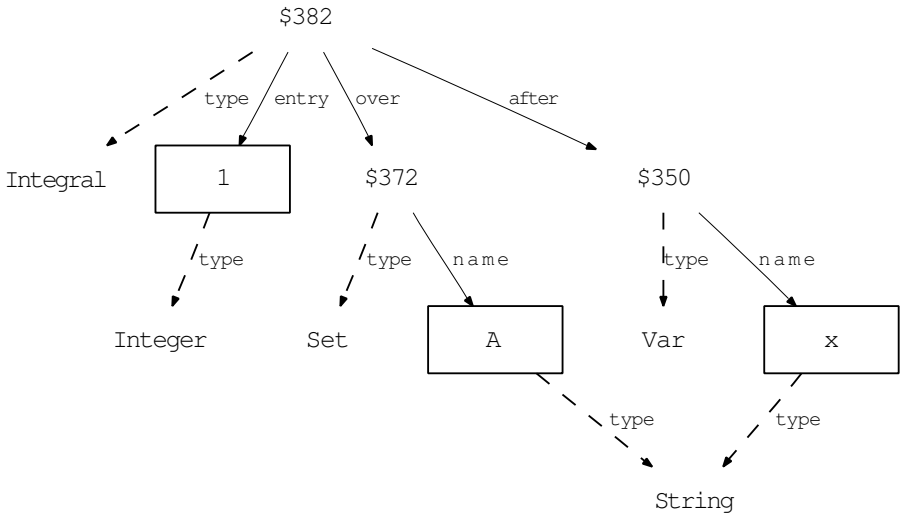


oneOf

Via `oneOf`, we require an object to have exactly one of a collection of fields with entries of a certain kind.

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

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



The restrictions we want to express are that the entry of the sem with field `over` must be a set, and the entry of the sem with field `fromTo` must be an

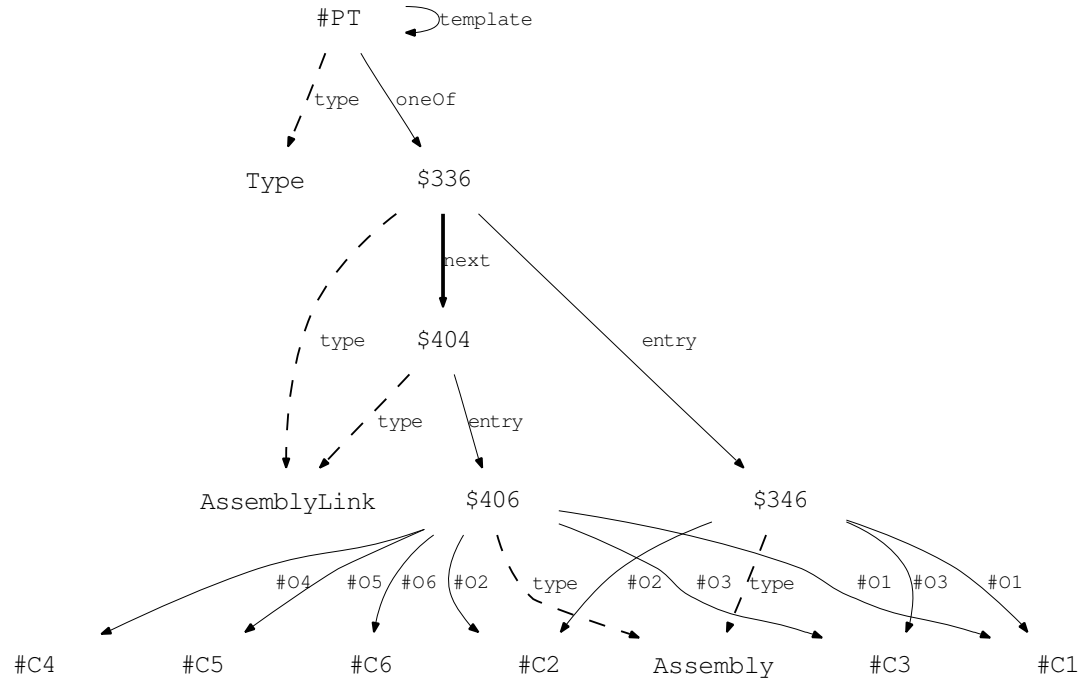
expression. For this, we use the quantifier `oneOf` in the type declaration of `Integral`. We assume `Integer` << `Expression`.

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

Representation in the SM. Consider a proper type #PT using `oneOf`:

```
Test(TypeSystem)::

#PT:
oneOf > #O1=#C1
        #O2=#C2
        #O3=#C3 ! etc.
oneOf > #O4=#C4
        #O5=#C5
        #O6=#C6 ! etc.
```

This is stored in the SM as the following semantic graph:



someOf

Via `someOf`, we require an object to have at least one constituent with a field from a collection of fields with entries of a certain kind.

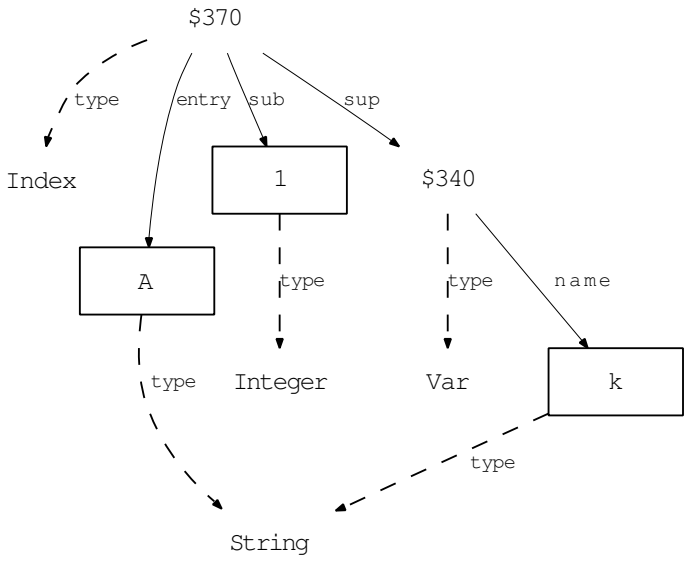
Example. The type declaration `Index` requires a subscript, a superscript, or a subscript or superscript on the left side, i.e, at least one of the positions **m**(#`obj`/`sub`), **m**(#`obj`/`sup`), **m**(#`obj`/`lsub`) and **m**(#`obj`/`lsup`) to be occupied by an expression. We express these requirements in the type declaration:

```
Index:
someOf> sub=Expression, sup=Expression
        lsub=Expression, lsup=Expression
allOf> entry=Expression
```

We assume that the union `Expression` contains `Integer`, `String` and `Var`. The expression

$$A_1^k$$

has both indices below and above, and is represented as:

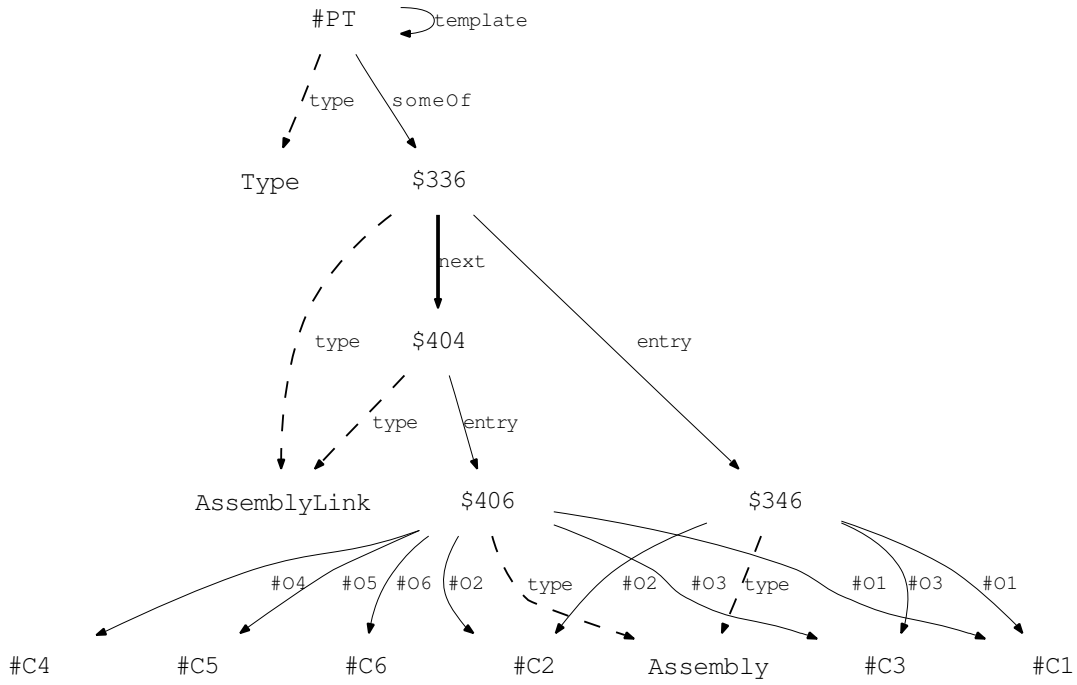


Representation in the SM. Consider a proper type #`PT` using `someOf`:

```
Test(TypeSystem)::
```

```
#PT:
someOf > #O1=#C1
         #O2=#C2
         #O3=#C3 ! etc.
someOf > #O4=#C4
         #O5=#C5
         #O6=#C6 ! etc.
```

This is stored in the SM as the following semantic graph:



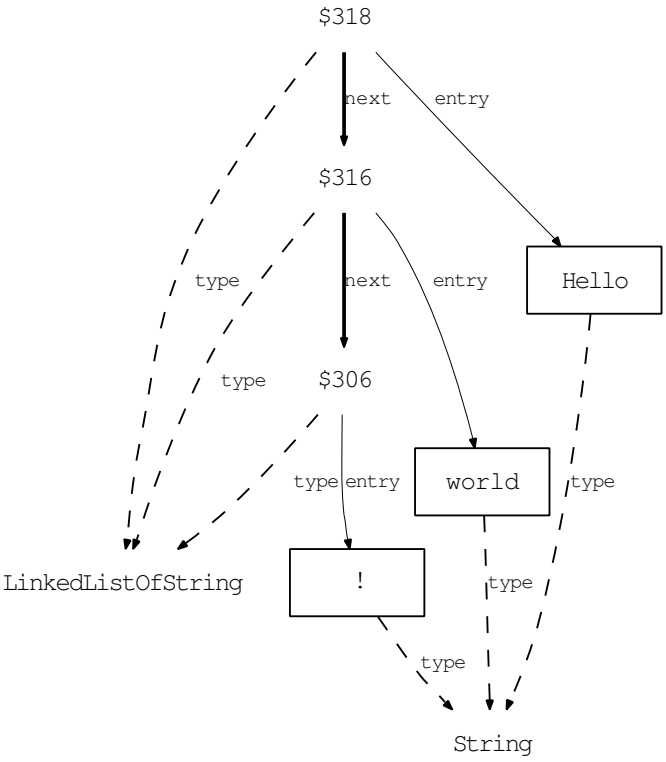
optional

Via `optional`, we require an object, if it has certain fields, to have entries of a certain kind.

Example. A linked list is a data structure in which there is a first value given, and every value, except the last value of the list, has a pointer to the next value. In the SM, a linked list consists of objects that all have a constituent with field `entry` and entry of some kind, and may have a constituent with field `next` that has another object of the linked list as entry. We express these restrictions for a linked list of strings in the type declaration:

```
LinkedListOfString:
allOf> entry=String
optional> next=LinkedListOfString
```

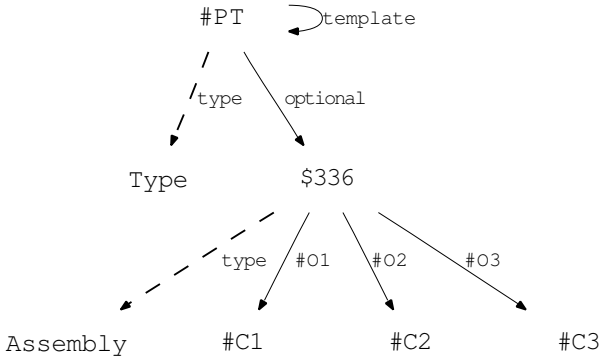
The linked list with entries “Hello”, “world” and “!” is then given by the semantic graph:



Representation in the SM. Consider a proper type #PT using `optional`:

```
Test(TypeSystem)::

#PT:
optional > #O1=#C1
           #O2=#C2
           #O3=#C3 ! etc.
```

This is stored in the SM as the following semantic graph:



fixed

Via `fixed`, we require an object to have a sem with given field and given entry.

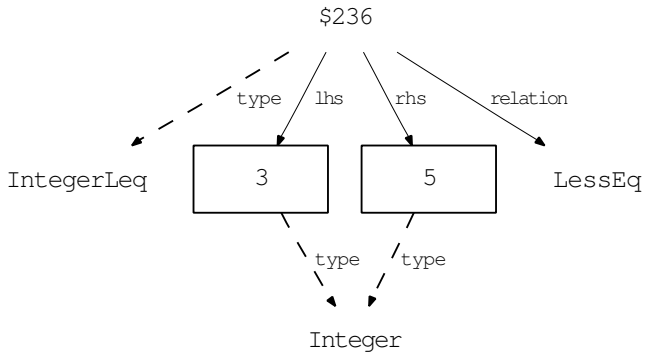
Example. We define a special binary relation `IntegerLessEq`:

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

Then the relation

$$3 \leq 5$$

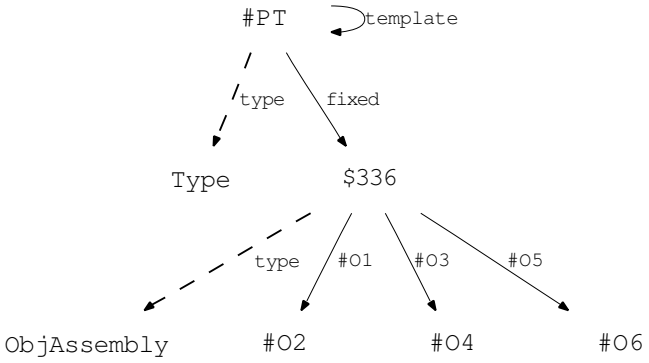
would be represented by:



Representation in the SM. Consider a proper type #PT using `fixed`:

```
Test(TypeSystem)::

#PT:
fixed> #01=#02
       #03=#04
       #05=#06  ! etc.
```

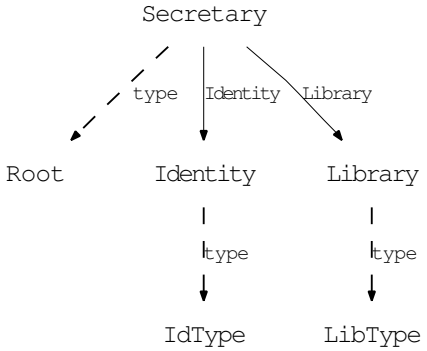
This is stored in the SM as the following semantic graph:



only

Via `only` we require an object to have all of a collection of sems where the field is the same object as the entry, and the entries need to be of a certain kind.

Example. Consider a category `Root`, which we want to use to represent root nodes of the semantic memory.



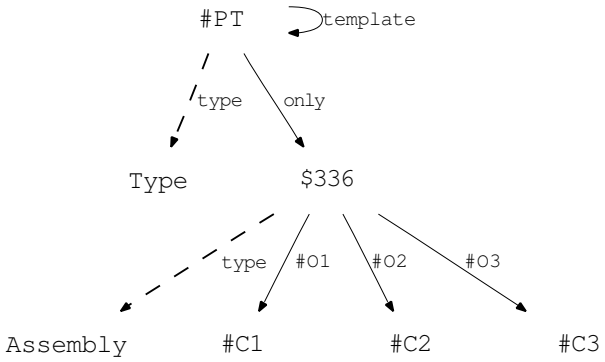
We require constituents with fields `Identity` and `Library` and entries equal to the fields, of type `IdType` and `LibType` respectively. We can express these restrictions via the following type declaration:

```
Root:
only> Identity=IdType
      Library=LibType
```

Representation in the SM. Consider a proper type #`PT` using `only`:

```
Test(TypeSystem)::

#PT:
only> #O1=#C1
      #O2=#C2
      #O3=#C3 ! etc.
```

This is stored in the SM as the following semantic graph:



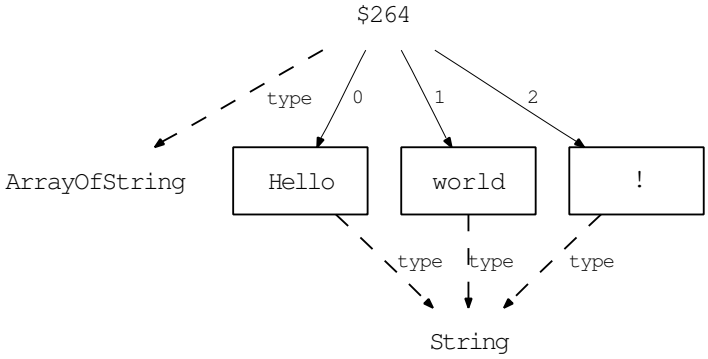
someOfType

Via `someOfType`, we require an object to have fields of a certain kind and entries of a certain kind.

Example. We define a 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
```

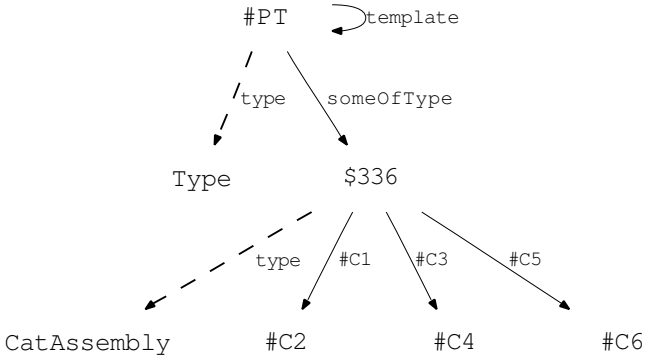
The following is the representation of the array of strings with entry 0 is "Hello", entry 1 is "world", and entry 2 is "!".



Representation in the SM. Consider a proper type #PT using `someOfType`:

```
Test(TypeSystem)::

#PT:
someOfType > #C1=#C2
             #C3=#C4
             #C5=#C6  ! etc.
```

This is stored in the SM as the following semantic graph:



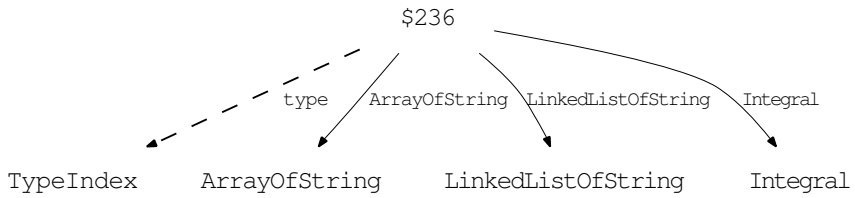
itself

Via `itself`, we require an object to have fields of a certain kind and entries equal to the field.

Example. We define an index of types:

```
TypeIndex:
itself> Type
```

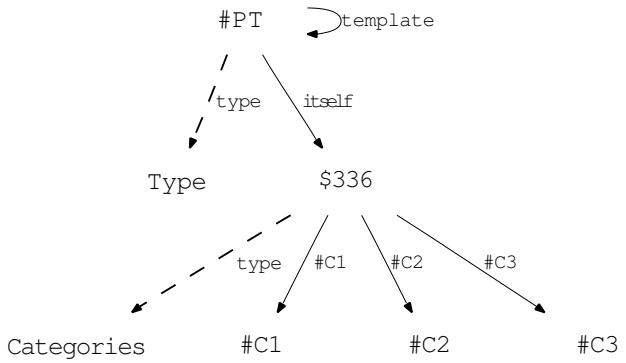
We give an example of such an index containing the proper types `ArrayOfString` and `LinkedListOfString` and `Integer`.



Representation in the SM. Consider a proper type #`DT` using `itself`:

```
Test(TypeSystem)::

#PT:
itself> #C1
        #C2
        #C3  ! etc.
```

This is stored in the SM as the following semantic graph:



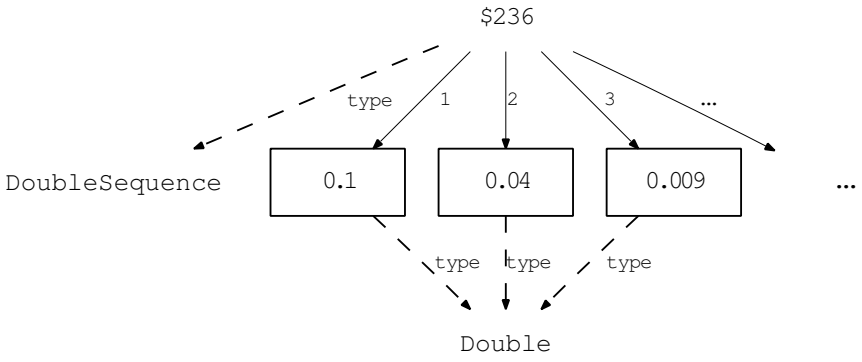
array

Scopes are objects describing a well-ordered set ob objects contained in the scope; see [8]. Via `array`, we require an object to have all the fields in a finite scope, and that their entries are of a certain kind.

Example. Consider a category `DoubleSequence` where the scope is the set of integers between 1 and 10 (represented by the object `From1To10`) and the entries are double precision floats:

```
DoubleSequence:
array> From1To10=Double
```

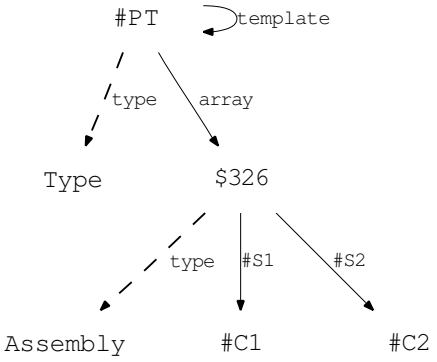
The sequence $(\frac{n^2}{10^n})_{n=1:10}$ would be represented by:



Representation in the SM. Consider a proper type #PT using `array`:

```
Test(TypeSystem)::

#PT:
array> #S1=#C1
       #S2=#C2  ! etc.
```

This is stored in the SM as the following semantic graph:



index

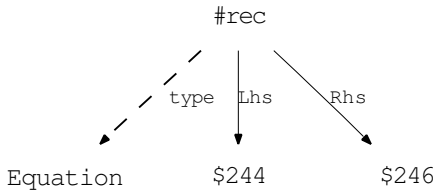
For some categories, we want to keep track of all their instances. Via `index`, we require each instance of some category to be listed in some assigned record.

Example. Consider a category `Equation`, with an object in `Lhs` and an object in `Rhs`, which can be expressed via `allOf`. But furthermore, each object that is of type `Equation` should be listed in a record `EquationList`, such that all objects in the semantic memory which are of type `Equation` can be found as fields of user defined object.

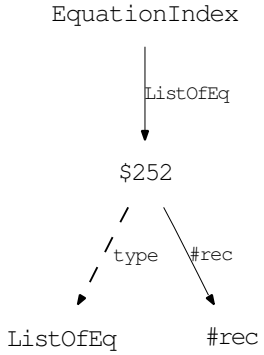
Assume the equation

$$x = y$$

represented in record #`rec`



which should be listed in the object `EquationIndex` :



Besides the requirements on the constituents of the record of type `Equation` we now add requirements to an object that acts as an index of all records of this type, in this case, the object `EquationIndex`. All the equations are to be stored in `EquationIndex.ListOfEq`, and `EquationIndex.ListOfEq.type` = `ListOfEq` We do this by the type declaration:

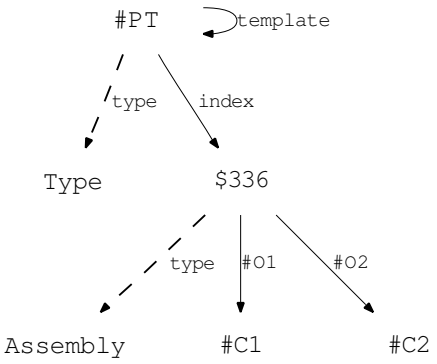
`ListOfEq:`

```
itself> Equation

Equation:
allof> Lhs=Object, Rhs=Object
index> EquationIndex = ListOfEq
```

Representation in the SM. Consider a proper type #PT using `index`:

```
Test(TypeSystem)::

#PT:
index> #O1=#C1
       #O2=#C2  ! etc.
```

This is stored in the SM as the following semantic graph:



template

For representation of a type declaration containing the line `template> #C` is equivalent to inserting the body of the type declaration of #C in place of this line.

Example. The general form of a binary relation is required in the type `BinaryRel`.

```
BinaryRel:
allOf> lhs=Expression
       rhs=Expression
       relation=Type
```

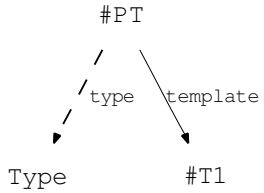
This is used as a template for the more specific proper type `LessEq` which is used to express the relation $\leq$:

```
LessEq:
template> BinaryRel
allOf> lhs=Term
       rhs=Term
fixed> relation=LessEq
```

Representation in the SM. Consider a proper type #PT using `template`:

```
Test(TypeSystem)::

#PT:
template> #T1
```

This is stored in the SM as the following semantic graph:

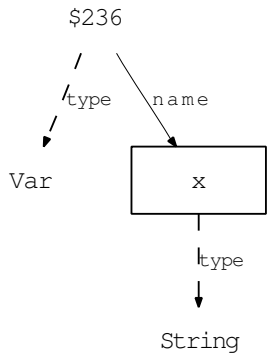


nothingElse

Via `nothingElse`, we require an object to have *only* the required constituents and the field `type`.

Example. The type declaration `Var` should only have a constituent with field `name` and a string as entry, but no other constituents (except for the field `type` which is always present in a proper type).

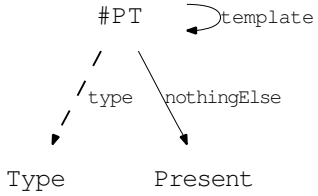
```
Var:
allOf> name=String
nothingElse>
```



Representation in the SM. Consider a proper type #DT using `nothingElse`:

```
Test(TypeSystem)::

#PT:
nothingElse>
```

This is stored in the SM as the following semantic graph:



4.3.2 Type systems

A type system is an object #TS in the semantic memory with #TS.type = TypeSystem. All categories #C belonging to the type system are stored in #TS.#C = #C.

4.3.3 Type declarations of atomics

nothing

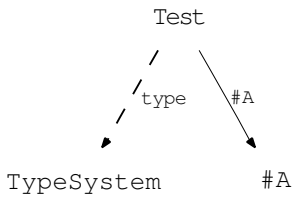
The operator `nothing>` defines an atomic type. Atomic types are objects that have a fixed semantic meaning, and must not have any constituents, not even a field `type`.

An atomic type does not pose any requirements on objects except itself, hence #obj.type = #A for some atomic type #A is meaningless.

Representation in the SM. Consider a type declaration of type #A using nothing, which defines the atomic type #A. Since #A must not have any constituents, it suffices to store that #A is part of the type system.
(Note that the typesheet reader also ensures that the type sheet contains a union Atomic that has all the atomic types of this type system as subtypes.)

```
Test(TypeSystem)::

#A:
nothing>
```

This is stored in the SM as the following semantic graph:



4.3.4 Type declarations of unions

union

A union defines the relation $<$, and hence also the relation $<<$ for a type system.

Example. 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

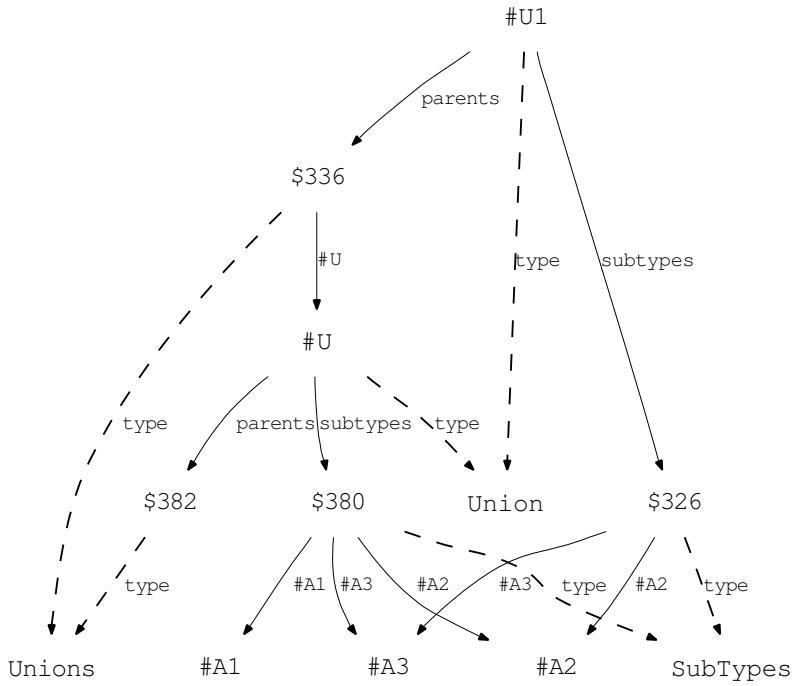
```
Rational:
union> Integer, Float, Double

Number:
union> Real, Rational
```

Note that Float, Double, Real and Integer have to be categories.
In the type sheet above, e.g., Real $<$ Number and Float $<$ Real are defined. Due to transitivity, e.g., Float $<<$ Number follows.

Representation in the SM. Consider a union #U:

```
Test(TypeSystem)::

#U:
union> #A1, #U1 !etc.
```

where #U1 is a union of the atomics #A2 and #A3.
This is stored in the SM as the following semantic graph:



A union #U knows about all minimal categories #C with #C << #U, this is necessary for matching. And #U has to know its immediate parents, i.e., categories #C with #U < #C to be able to recursively propagate new categories contained by #U upwards.

atomic

The operator `atomic` defines a type as a union of atomics. The atomics need not exist at that point, so as a byproduct, this may result in the definition of new atomic types.

```
#TD:
atomic> #A1, ... , #Ak
```

This is a short-hand notation for

```
#A1, ... , #Ak:
nothing>

#TD:
union> #A1, ... , #Ak
```

complete

This operator declares that no further categories can be added to a union. Usually, one may add more categories to a union later, e.g.:

```
Number: Number +
union> Complex
```

But this is forbidden if the definition of `Number` contains a line `complete>`.

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

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

While it is still possible to later define a new category `Shortbook` with the property `Shortbook` << `Documents`, e.g., with the declaration

```
Book:
union> Shortbook
```

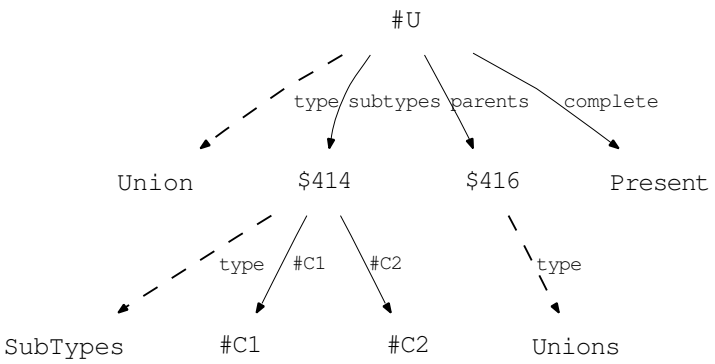
it is not possible to add `Shortbook` with the property `Shortbook` < `Documents`, e.g., with the declaration

```
Documents: Documents +
union> Shortbook
```

Representation in the SM. Consider a union #`U` using `complete`:

```
Test(TypeSystem)::

#U:
union> #C1, #C2  ! etc.
complete>
```

This is stored in the SM as the following semantic graph:



index

This applies the requirement to index all instances to all the subtypes of a union.

Example. Assume we want to have an index that lists all instances of inequalities. Inequalities are the objects that have either type `LessEq`, `Less`, `GreaterEq` or `Greater`, and we want them to be indexed in the object `IndexOfIneq`.

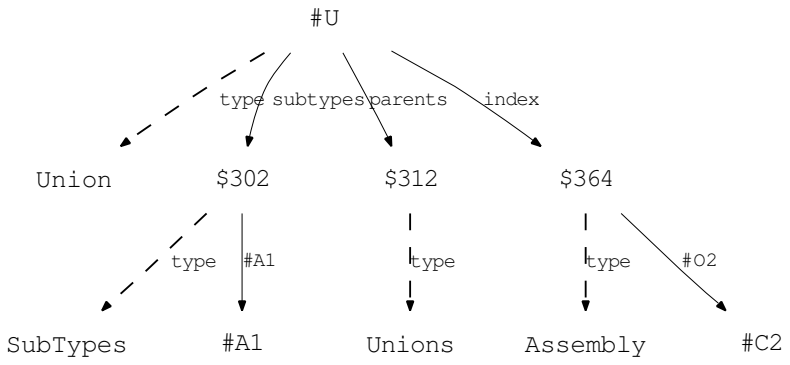
```
Inequality:
union> LessEq, Less, GreaterEq, Greater
index> IndexOfIneq=Inequalities
```

Representation in the SM. Consider a union #`U` using `index`:

```
Test(TypeSystem)::

#U:
atomic> #A1
index> #O1=#C1
       #O2=#C2  ! etc.
```

(This type declaration also declares an atomic type, which is necessary to characterize it as a union, else it would be treated as a proper type according to Subsection 4.3.1.)

This is stored in the SM as the following semantic graph:



This information is propagated to all the subtypes of #U, and represented in the semantic memory according to Subsection 4.3.1.

4.4 Well-typed records

Assume a record with handle #rec, and let the type of #rec be #T.

To define when this record is well-typed, we first define which position of an object are a **declared position** and in which case a position is **faulty**.

If no position reachable from the handle #rec is faulty, then the record is **well-typed** of type #T.

Otherwise the record is **ill-typed**.

Note that it can be checked in time linear in the number of sems that are reachable whether or not the record is well-typed.

We consider an object #obj with #obj.type = #TD and #TD.type = Type.

4.4.1 allOf

For every field #f, #f≠type of #TD.allOf, (#obj/#f) is a declared position of #obj.

If for one of the declared positions (#obj/#f) is empty, or not **m**(#obj.#f/#TD.allOf.#f) then (#obj/#f) is faulty.

4.4.2 oneOf

For all $k = 0, 1, 2, \ldots$ and for every field #f, #f ≠ type of #TD.oneOf.next^k.entry, the position (#obj/#f) is a declared position of #obj.

If a declared position (#obj/#f) is nonempty and not **m**(#obj.#f/#TD.oneOf.next^k.entry.#f) then it is faulty.

If not for all k exactly one of the declared positions (#obj/#TD.oneOf.next^k.entry.#f) is occupied then the the position (#obj/#f) is faulty.

4.4.3 someOf

For all $k = 0, 1, 2, \ldots$ and for every field #f, #f $\neq$ type of #TD.someOf.next^k.entry, the position (#obj/#f) is a declared position of #obj.

If a declared position (#obj/#f) is nonempty and not **m**(#obj.#f/#TD.someOf.next^k.entry.#f) then it is faulty.

If not for all k at least one of the declared positions (#obj/#TD.someOf.next^k.entry.#f) is occupied then the the position (#obj/#f) is faulty.

4.4.4 optional

For every field #f, #f $\neq$ type of #TD.optional, (#obj/#f) is a declared position of #obj.

If a declared position (#obj/#f) is nonempty and not **m**(#obj.#f/#TD.optional.#f) then it is faulty.

4.4.5 fixed

For every field #f, #f$\neq$type of #TD.fixed, (#obj/#f) is a declared position of #obj. If a declared positions (#obj/#f) is either empty or does not satisfy #obj.#f = #TD.fixed.#f then (#obj/#f) is faulty.

4.4.6 only

For every field #f, #f $\neq$ type of #TD.only, every position (#obj/#f) is a declared position of #obj. If a declared positions #obj.#f does not satisfy both #obj.#f = #f and **m**(#obj.#f/#TD.only.#f) then (#obj/#f) is faulty.

4.4.7 array

Not implemented yet.

4.4.8 itself

For every field #f, #f ≠ type of #TD.itself, every position (#obj/#F) with **m**(#F/#f) is a declared position of #obj. For all declared positions (#obj/#F), if #obj.#F ≠ #F then (#obj/#F) is faulty.

4.4.9 someOfType

For every field #f, #f ≠ type of #TD.someOfType, every position (#obj/#F) with **m**(#F/#f) is a declared position of #obj. If a declared position (#obj/#F) does not satisfy **m**(#obj.#F/#TD.someOfType.#f) then (#obj/#f) is faulty.

4.4.10 nothingelse

If #TD.nothingelse = Present, and there exists an occupied position (#obj/#f) of #obj that is not a declared position, then (#obj/#f) is faulty.

4.5 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 40 lines. When represented in the semantic memory, the record has 126 sems.

```
BasicTypes::

Type:
  index> Index = Types
  allOf> template=Type
  someOf> 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
  ! array can be tightened when specified

Atomic:
  atomic> Present
  index> Index = Atomics

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

SubType:
  union> Atomic, Type

Category:
  union> Union, Type, Atomic

TypeSystem:
  index> Index = TypeSystems
  itself> Category


! * BASIC COLLECTIONS *

Atomics:
  itself> Atomic

Types:
  itself> Type

SubTypes:
  itself> SubType

Unions:
  itself> Union

Categories:
```

```
  itself> Category

TypeSystems:
  itself> TypeSystem

! * BASIC DEFINITIONS *

Assembly:
  someOfType> Object=Category

ObjAssembly:
  someOfType> Object=Object      ! This does nothing

CatAssembly:
  someOfType> Category=Category

AssemblyLink:
  allOf> entry=Assembly
  optional> next=AssemblyLink
```

Chapter 5

Applications

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:

- Different types of mathematical formulas were extracted from lecture notes about basic analysis and linear algebra [32]. These were manually fed into the semantic memory to assure generality of the representation of formulas. Partial work on the grammar of the text part of the lecture notes can be found in [41]. Some of the expressions from the lecture notes are presented in Section 5.

- As a test for representing informal mathematical text in the SM we represented the informal description of program taken from the 1991 ACM International Collegiate Programming Contest.

- 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. 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.

- An interface was written to automatically import formulas from the TPTP library [47] ("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 [24] ("Natural Language Proof Checking").

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) [37] are the subject of the PhD thesis by Kevin Kofler [19]. This parser will handle updates to the grammar, a feature necessary to handle mathematical definitions that introduce new syntax.

5.1 Mathematical formulas

We extracted 30 different types of mathematical formulas from mathematical texts, primarily from the lecture notes about basic analysis and linear algebra [32] by the second author. These formulas were manually fed into the semantic memory to assure generality of the representation of formulas.

The following formulas have been chosen and their representation can be found in Appendix C:

$$\left(\sqrt{x}, x+y+z\right)$$

$$(\sqrt{x_1}, x_1+x_2+x_3)$$

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

$$\begin{pmatrix} + & - & + & - \\ - & + & - & + \\ + & - & + & - \\ - & + & - & + \end{pmatrix}$$

$$\begin{pmatrix} \sigma_1 & & & \\ & \ddots & & \\ & & \sigma_r & \\ & & & 0 \end{pmatrix}$$

$$\{x \in \mathbb{R} \mid 0 \leq x \leq 1 \vee x = 2\}$$

$$\{\leq, =, \geq\}$$

$$\{f\,(x) \mid x \geq 1\}$$

$$\{x{\in}X\left(k\right)\mid f\left(x,k\right){=}0\}$$

$$a\overset{\text{Vor.}}{=}b$$

$$f\left(x\right)|_{x=a}{=}f\left(a\right)$$

$$a\overset{(1)}{=}b{\geq}c{\geq}d\overset{(2)}{=}e$$

$$\sum_{k=1}^{n}A_{ik}{=}b_{i}\qquad(i{=}1,\ldots,n)$$

$$\sum_{k\in K}\Pr(i|k)\Pr(k)$$

$$\left\|A\right\|_{F}{:=}\sqrt{\sum_{i=1:m,k=1:n}{A_{ik}}^{2}}$$

$$\int_{B}\int_{A}f\left(x_{1},x_{2}\right)\ \mathrm{d}x_{1}\ \mathrm{d}x_{2}{=}\int_{A\times B}f\left(x\right)\ \mathrm{d}x$$

$$\int_{0}^{x}t\ \mathrm{d}t{=}\left.\frac{t^{2}}{2}\right|_{0}^{x}{=}\frac{x^{2}}{2}$$

$${K_{B_{ij}}}^{2}{=}{}^{2}\Phi^{i,j}$$

$${A^{i}}_{;k}{=}{A^{i}}_{,k}+{A^{i}}_{,k}\Gamma_{ka}^{i}$$

$$[0,1]{=}\{x{\in}\mathbb{R}\mid0{\leq}x{\leq}1\}$$

$$(0,2]{=}(0,1)\cup[1,2]$$

$$\lambda x.x+1$$

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

$$x^{l}\qquad(l{=}1:n)$$

$$X\left(k\right){=}\lambda x.P\left(x,k\right)$$

$$f'$$

$$\begin{cases}0 & \text{if } x{<}0\\ x^{2} & \text{if } x{>}1\\ x & \text{otherwise}\end{cases}$$

$$\frac{\partial^{2}}{\partial x\partial y}2x^{2}y$$

$$\max\left\{x+y,y+z,x+z\right\}{=}x+y+z-\min\left\{x,y,z\right\}$$

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

The Table 5.1 gives a small statistic of the examples:

In the representation of mathematical formulas, the type acts as an operator.

An **operation** is anything that can be applied to mathematical expressions $E_1, E_2, \ldots$ such that the result E is an expression again. We call $E_1, E_2, \ldots$ the **subexpressions** of E. In particular, all standard functions, binary operations, and relations are operations, and so are quantification, merging expressions to form a set, a vector, etc.

We store the information in a fashion inspired by automatic differentiation. Thus we proceed from the most elementary subexpressions (its variables and constants) to the more complicated subexpressions by applying operations until the expression is fully covered.

An **operation** is anything that can be applied to mathematical expressions $E_1, E_2, \ldots$ such that the result E is an expression again. We call $E_1, E_2, \ldots$ the **subexpressions** of E. In particular, all standard functions, binary operations, and relations are operations, and so are quantification, merging expressions to form a set, a vector, etc. The operations are those categories that match the category `Expression` in the type sheet for expressions, see Appendix B.1.

We store the information in a fashion inspired by automatic differentiation, meaning we proceed from the most elementary subexpressions (its variables and constants) to the more complicated subexpressions by applying operations until the expression is fully covered.

Let the record `#handle` contain the expression E. Then we say that `#handle` is the **handle** of E. From the handle of some expression, the expression E itself and the free variables of E have to be accessible easily from `#handle`. The nodes representing the free variables of E are stored in `#handle.free` in the following fashion: For every node `#var` representing a free variable of E, we have `#handle.free.#var=#var`. If some expression does not have any free variables, then `#handle.VAR` is nonempty but does not have children.

The expression itself is constructed from its subexpressions in a recursive way, with constants and variables being expressions without subexpressions. The operation that is applied to the subexpressions of E is represented in the object `#handle.type`, the same object that is used for the typing of `#handle`. How the subexpressions of E are represented in relation to `#handle` depends on the kind of operation, see below.

When an operation is applied to subexpressions, the free variables of the combined expression form the union of the free variables of the subexpressions, minus the variables that are bound by the operation. Every variable `#var` that is bound by application of the operation represented in `#handle.type` is stored as `#handle.binds.#var=#var`.

5.1.1 Types of expressions

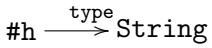
We now illustrate the different types of expressions: since we build up all expressions from variables, constants and the application of operations to subexpressions, we have to describe the representation of these.

The handle of the expression is always denoted by `#handle` or `#h`.

Constants

There are currently three types of constants: strings, integers and floats, and all of them are represented in a similar fashion. The actual constant is always stored as the value of the handle of the constant. The record `#h` representing a constant has `#h.type=String` if `#h` is a string, `#h.type=Integer` if `#h` is an integer, and `#h.type=Float` if `#h` is a float.

For example, the string "Hello world" is represented as



where `#h` is some anonymous node with `VALUE(#h) = Hello world`.

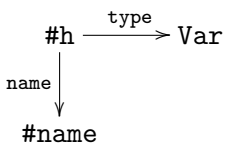
The type declaration of the constant types are:

```
String, Integer, Float:
nothingElse>
```

Variables

The record `#h` representing a variable has `#h.type=Var`.

A name can, but need not be assigned to the variable. A variable with name `x1` is represented as



where the object `#name` is a string containing `x1`.

The type declaration of variables are:

```
Var:
optinal> name=String
nothingElse>
```

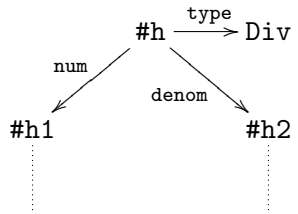
Operations with fixed arguments

These are the operations that only allow a certain number of subexpressions to be applied to, and these subexpressions have a known role in the resulting expression E.

For example, the operation "square root" has one argument, the *radicand*, a fraction has two arguments, the *numerator* and the *denominator*, etc.

This reflects in the way these expressions are represented. For an expression E represented as record `#h` the subexpressions will be represented in `#h.#field` where the name of `#field` will usually unambiguously clarify the role of the subexpression in `#h.#field` for the expression in `#h`.

For example, consider an expression E with E_1 being it numerator and E_2 the denominator, hence $E = \frac{E_1}{E_2}$.
If E_1 is represented in `#h1` and E_2 is represented in `#h2` then the representation of the expression E in record `#h` (omitting the free variables) is:



The type declaration of a division is:

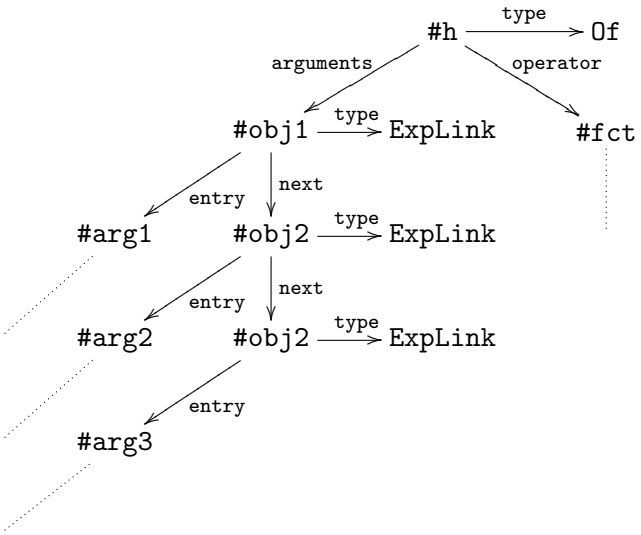
```
Div:
allOf> num=Expression, denom=Expression
```

General n-ary Operations

Another kind of operations are those that admit an arbitrary number of subexpressions to be applied to, but all of these are treated equally. But there may still be a known number of subexpressions aside of these that have a fixed role.

For example, a case distinction between n cases, and as an extra argument the case "otherwise", or the application of a function f to n arguments. In these cases, the n arguments are always represented as a linked list.

For example, consider the expression $f(x_1, x_2, x_3)$ where f is represented in `#fct` and x_i is represented in `#arg`i. Then the representation of the expression $f(x_1, x_2, x_3)$ in record `#h` (again omitting the free variables) is:



The type declaration of this application is:

```
Of:
allOf> operator=Expression, arguments=ExpLink

ExpLink:
allOf> entry=Expression
optional> next=ExpLink
```

Example

We give a complete record, with the bound variables and the free variables. Consider the expression:

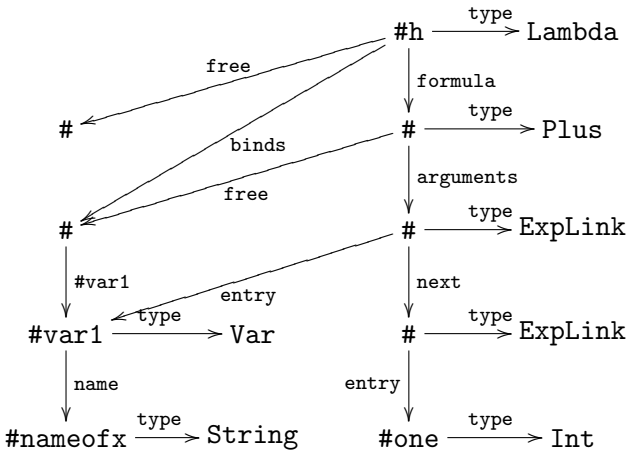
$$\lambda x.x + 1$$

The subexpressions are the variable x and the constant 1, the result of the operator `Plus` to these, resulting in $x + 1$ (having free variable x) and lastly the application of the operator `Lambda` binding variable x, hence resulting in $\lambda x.x + 1$ which has no free variable.

Anonymous nodes that are not referred to are simply denoted by `#`. Assume that `VALUE(#one) = 1` and `VALUE(#nameofx) = x`.

5.2 The representation of informal mathematical text

For being attractive for a working mathematician, the ability to interface existing systems is one key feature, another one is communication in an



almost natural language.

There is a consensus among mathematicians and linguists that the communication of mathematics to a computer is much easier than the communication of arbitrary content:

- Mathematical discourse has a well-defined domain, is highly structured, and has relatively small set of discourse relations. The reasoning patterns applied in mathematics are widely studied and understood [53]. Building an ontology for, say, number theory, is much easier than for a natural domain, because mathematicians *define* concepts before they use them. It was even claimed that "[...] if we fail to construct an understander for mathematical discourse, then we will also fail to write one for other (non-trivial) domains", see p. 8 in [53].

- Due to the fact that mathematicians want to communicate unambiguously, they tend to use a relatively small set of phrases to express their ideas, and there is a standard interpretation for these phrases. About 700 phrases suffice for the essential part of mathematics (definitions, theorems, proofs, etc.) but this does not include the more informal motivational part [50].

- Mathematicians use words and phrases in a very rigid way. The language of mathematics is simple: very few variety in time, person, etc. [10].

- Another reason why mathematics is apt to be represented by a machine is that in mathematics we are in the (probably unique) position that every meaningful rigorous statement can, at least in principle, be translated into a formal language. Therefore, it is possible for a machine to faithfully represent the complete content of an arbitrary (but meaningful) mathematical statement.

However, we do not intend to allow general natural language as input, even though we expect only relatively simple sentences, but we intend to exploit the fact that mathematical language is simple by defining a controlled natural language (CNL) that is expressive enough to fulfill the needs of mathematicians, while still sounding like natural language.

As a test for representing informal mathematical text in the SM we chose the description of program taken from the 1991 ACM International Collegiate Programming Contest, as printed in [46]. The text describes the use, input and output of a program a participant had to write.

The automatically created output contains grammatical errors. We plan to overcome these problems by interfacing the Grammatical Framework [37].

Original text: The Center City fire department collaborates with the transportation department to maintain maps of the city that reflect the current status of the city streets. On any given day, several streets are closed for repairs or construction. Firefighters need to be able to select routes from the fire stations to fires that do not use closed streets.

Central City is divided into non-overlapping fire districts, each containing a single fire station. When a fire is reported, a central dispatcher alerts the fire station of the district where the fire is located and gives a list of possible routes from the fire station to the fire. You must write a program that the central dispatcher can use to generate routes from the district fire stations to the fires.

The city has a separate map for each fire district. Street corners of each map are identified by positive integers less than 21, with the fire station always on corner #1. The input file contains several test cases representing different fires in different districts. The first line of a test case consists of a single intege which is the number of the street corner closest to the fire. The next several lines consist of pairs of positive integers separated by blank which are the adjacent street corners of open streets. The final line of each test case consists of a pair of 0's.

For each test case, your output must identify the case by number (case #1, case #2, etc). It must list each route on a separate line, with the street corners written in the order in which they appear on the route. And it must give the total number routes from fire station to the fire. Include only routes which do not pass through any street corner more than once. Output from separate cases must appear on separate lines. The following sample input and corresponding correct output represents two test cases.

Automatically generated output (currently without proper inflections and articles): Fire truck.

Fire department collaborate with transportation department in order to

maintain map of city with map of city reflect status of street. For all day , several street is closed in order to repair or construction. Firefighter must be able to select route under the constraint route start at fire station , route end at fire and not route use closed street. City is divided into not overlapping fire district with for all fire district , fire district contain 1 fire station. If fire is reported then central dispatcher alert fire department of fire district of fire and give list of route with route start at fire station and route end at fire.

Write program such that central dispatcher is able to generate route with route start at fire station and route end at fire.

City has separate map for all fire district. Street corner identified as positive integer less than 21 and street corner of fire station identified as 1. Input file contains several test case with test case represent different fire in different fire district. First line of test case consist of single positive integer (number of street corner that is street corner closest to fire). Next several line consist of pair of positive integer with positive integer separated by blank (adjacent street corner of open street). Last line of test case consist of pair of 0.

For all test case , create output such that output identify test case by number. Output must give each route has separate line for all route and order of street corner of output is equal to order of street corner of route. Output must give number of route. Output must not include route with number of route has route pass street corner greater than 1. Output of separate test case must appear at separate line.

5.3 The representation of optimization problems

A significant fraction of the optimization problems from the OR Library [2] were represented manually in the semantic memory. 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 printed in Appendix B.3.

There are algorithms that produce LaTeX descriptions from the representation in the semantic matrix, and others that produce a model description in the algebraic modeling language AMPL [9] specifying a valid, numerically solvable model.

The Operations Research Library (OR-Library), maintained by J. E. Beasley and originally described in [2], is an online resource of test data sets for a variety of Operations Research problems.[1] It contains 111 problem classes, 59

[1] The OR-Library is available at `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

problem classes are downloadable directly from the OR-Library and 52 are links to data sets outside the OR-Library. For our project, we concentrate on the 59 problem classes actually contained in the OR-Library.

The OR-Library contains data for well-known optimization problems like the traveling salesman problem, the bin packing problem, set covering, Hamiltonian cycle etc. For example, many of the (NP-complete) problems in the seminal work [18] are included in the OR-Library.

For one specific problem, the OR-Library contains:

- the reference to a publication where this data set was originally described and used
- information about number and size of the files
- a description of the structure of the data in the files
- the data files itself.

Appendix D contains the LaTeX-description and the AMPL-output of 7 problems in the ORLib.

5.4 The TPTP Library

An interface was written to automatically import formulas from the TPTP library [47, 48] ("Thousands of Problems for Theorem Provers"), a library of formulas for theorem provers, taken from different branches of mathematics.

We implemented a parser for problem files of the TPTP, and parsed, represented and typechecked the complete TPTP library (version 3.5.0), which adds up to more than 10,000 problem files.

As an example we give one small problem file from the TPTP, `SET002+4.p` (note that we excluded some comments):

```
%--------------------------------------------------------
% File     : SET002+4 : TPTP v3.5.0. Released v2.2.0.
% Domain   : Set Theory (Naive)
% Problem  : Idempotency of union
% Version  : [Pas99] axioms.
% Comments :
%--------------------------------------------------------
%----Include set theory definitions
include('Axioms/SET006+0.ax').
%--------------------------------------------------------
fof(thI14,conjecture,
    ( ! [A] : equal_set(union(A,A),A) )).
%--------------------------------------------------------
```

The graph that represents this problem file in the semantic memory is given in Figure 5.2. Note that not only the formula itself is represented, but also the domain, the axioms to be included, etc.

Appendix B.4 contains the type sheet for problems from the TPTP.

5.5 Naproche

The Naproche project (Natural language Proof Checking, [24]) is carried out at the University of Bonn. It provides a controlled natural language for mathematical texts with formulas. Texts written in this language can be checked for syntactical and mathematical correctness.

The web interface of the Naproche project[2] offers three texts as examples. We represented two of the three examples in the semantic memory. The automatically generated output was accepted and successfully checked for correctness by the web interface.

The Burali-Forti paradox was represented using 521 sems and the output is given in Appendix E.1.

The example from elementary group theory was represented using 448 sems and the output is given in Appendix E.2.

[2] `http://naproche.net/inc/webinterface.php`

Example	# visible symbols	# LaTeXcharacters	# sems
1	10	38	31
2	14	78	37
3	11	88	54
4	18	113	39
5	6	95	34
6	15	65	53
7	7	36	10
8	10	50	28
9	17	75	46
10	7	29	16
11	13	62	39
12	15	73	60
13	20	76	73
14	16	45	28
15	22	73	90
16	28	159	75
17	19	105	50
18	11	51	57
19	18	84	83
20	17	64	53
21	17	38	23
22	6	19	21
23	22	76	60
24	9	32	45
25	16	56	41
26	2	8	15
27	24	99	33
28	11	54	32
29	33	99	86
30	14	51	42

Figure 5.1: Statistics of examples of mathematical expressions

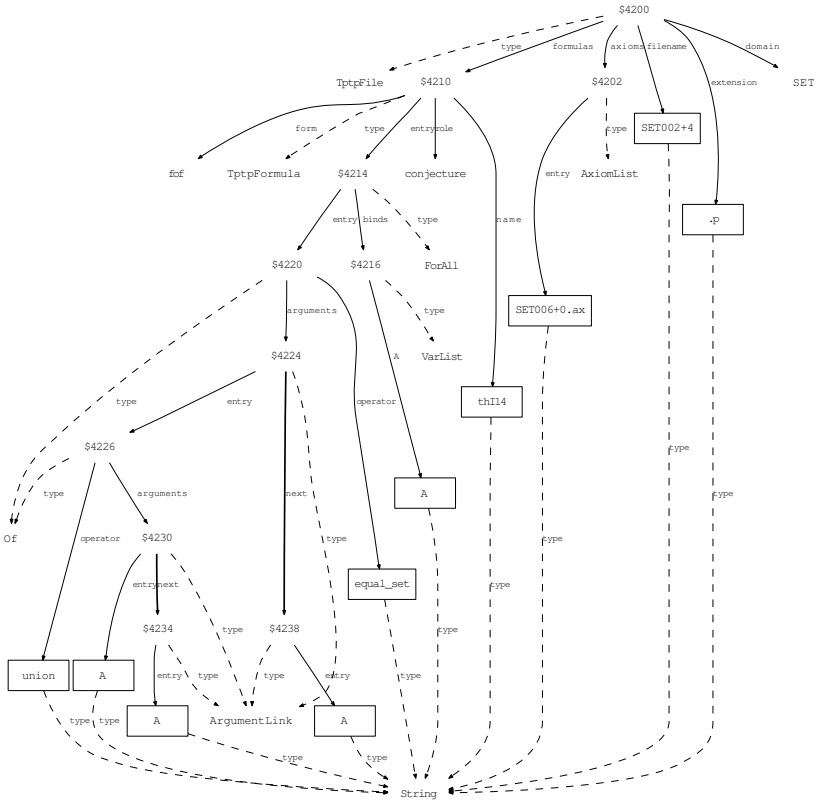


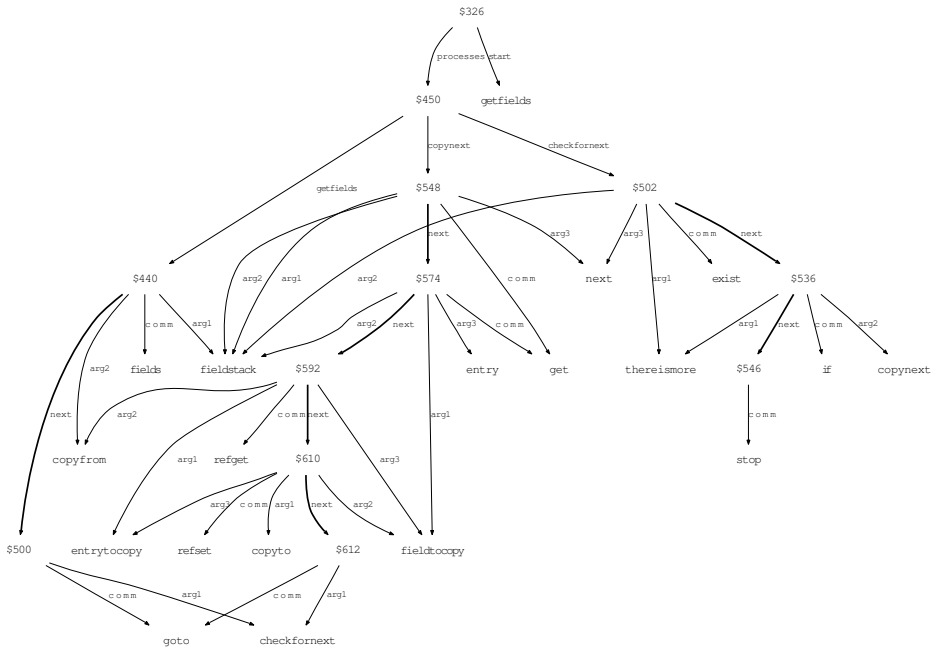
Figure 5.2: A problem file of the TPTP

Appendix A

SVM programs

A.1 The SVM program `copyFields` as a semantic graph

The following semantic graph is the record that represents the SVM program `copyFields` as given in text form in Section 3.6, page 33. Note that for transparency, the sems with field `type` are not printed.



A.2 The SVM code of the USVM

The example program below implements a simulator for the SVM, which shows that the SVM programming language is universal.

```
program(USVM)

%%%%%%%%%%%%%%%%%%%%%% control handling %%%%%%%%%

process(init)
% initialize nodes, initialize local and global frame
    (simcore,core)=set(simcore)
    (simcore,program)=set(simprog)
    simprocesses=get(simprog,processes)
    startfocus=get(simprog,start)
    simfocus=refget(simprocesses,startfocus)
    goto(load)

process(next)
% proceed to the next command to simulate
    simfocus=get(simfocus,next)
    goto(load)

process(load)
% load the information about the command
% to simulate to the core
    sim_comm=get(simfocus,comm)
    move(sim_comm)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

process(move)
    processname=get(simfocus,arg1)
    process=refget(simcore,processname)
    simfocus=refget(simprocesses,process)
    goto(load)

process(goto)
    process=get(simfocus,arg1)
    simfocus=refget(simprocesses,process)
    goto(load)

process(if)
% if(#criterion,#process)
```

```
    criterionname=get(simfocus,arg1)
    criterion=refget(simcore,criterionname)
    if(criterion,ifapplies)
    goto(ifappliesnot)
process(ifapplies)
    process=get(simfocus,arg2)
    simfocus=refget(simprocesses,process)
    goto(load)
process(ifappliesnot)
    goto(next)

process(stop)
    stop

%%%%%%%%%%%%%%%%%% internal handling %%%%%%%%%%%%

process(create)
    toassign=get(simfocus,arg1)
    create(newobj)
    (simcore,toassign)=refset(newobj)
    goto(next)

process(fields)
% #fieldlist=fields(#record)
    writetoname=get(simfocus,arg1)
    getfromname=get(simfocus,arg2)
    getfrom=refget(simcore,getfromname)
    stackoffields=fields(getfrom)
    (simcore,writetoname)=refset(stackoffields)
    goto(next)

process(check)
% #isequal=check(#left,#right)
    leftname=get(simfocus,arg2)
    left=refget(simcore,leftname)
    rightname=get(simfocus,arg3)
    right=refget(simcore,rightname)
    result=check(left,right)
    writeto=get(simfocus,arg1)
    (simcore,writeto)=refset(result)
    goto(next)

process(exist)
% #result=exist(#record,#field)
```

```
    leftname=get(simfocus,arg2)
    left=refget(simcore,leftname)
    right=get(simfocus,arg3)
    result=existref(left,right)
    writeto=get(simfocus,arg1)
    (simcore,writeto)=refset(result)
    goto(next)

process(existref)
% #result=exist(#recordname,#fieldname)
    leftname=get(simfocus,arg2)
    left=refget(simcore,leftname)
    rightname=get(simfocus,arg3)
    right=refget(simcore,rightname)
    result=existref(left,right)
    writeto=get(simfocus,arg1)
    (simcore,writeto)=refset(result)
    goto(next)

process(setconst)
% (#handle,#field)=const(#setto)
    handle=get(simfocus,arg1)
    sethandle=refget(simcore,handle)
    field=get(simfocus,arg2)
    setto=get(simfocus,arg3)
    (sethandle,field)=refset(setto)
    goto(next)

process(refset)
% (#handlename,#fieldname)=set(#entryname)
    handle=get(simfocus,arg1)
    sethandle=refget(simcore,handle)
    fieldname=get(simfocus,arg2)
    field=refget(simcore,fieldname)
    entry=get(simfocus,arg3)
    setto=refget(simcore,entry)
    (sethandle,field)=refset(setto)
    goto(next)

process(set)
% (#handlename,#field)=set(#entryname)
    handle=get(simfocus,arg1)
    sethandle=refget(simcore,handle)
    field=get(simfocus,arg2)
```

```
    entry=get(simfocus,arg3)
    setto=refget(simcore,entry)
    (sethandle,field)=refset(setto)
    goto(next)

process(refget)
% #towrite=get(#handlename,#fieldname)
    addressname=get(simfocus,arg1)
    handlename=get(simfocus,arg2)
    handle=refget(simcore,handlename)
    fieldname=get(simfocus,arg3)
    field=refget(simcore,fieldname)
    towrite=refget(handle,field)
    (simcore,addressname)=refset(towrite)
    goto(next)

process(get)
% #towrite=get(#handlename,#field)
    addressname=get(simfocus,arg1)
    handlename=get(simfocus,arg2)
    handle=refget(simcore,handlename)
    field=get(simfocus,arg3)
    towrite=refget(handle,field)
    (simcore,addressname)=refset(towrite)
    goto(next)

process(unset)
% unset(#handle.#field)
    firstname=get(simfocus,arg1)
    secondname=get(simfocus,arg2)
    first=refget(simcore,firstname)
    second=refget(simcore,secondname)
    unset(first,second)
    goto(next)

%%%%%%%%%%%%%%%%%%%%% external handling %%%%%%%%%%

process(external)
% #output = external(#process,#input)
    inputname=get(simfocus,arg3)
    processname=get(simfocus,arg2)
    outputname=get(simfocus,arg1)
    inputnode=refget(simcore,inputname)
    processobj=refget(simcore,processname)
```

```
    outputnode=refget(simcore,outputname)
    outputnode=external(processobj,inputnode)
    goto(next)

process(in)
% #writeto = in(#readfrom,#protocol)
    transto=get(simfocus,arg1)
    transfrom=get(simfocus,arg2)
    protocol=get(simfocus,arg3)
    objtotransfrom=refget(simcore,transfrom)
    transtotemp=refin(objtotransfrom,protocol)
    (simcore,transto)=refset(transtotemp)
    goto(next)

process(out)
% #writeto = out(#readfrom,#protocol)
    transto=get(simfocus,arg1)
    transfrom=get(simfocus,arg2)
    protocol=get(simfocus,arg3)
    objtotransfrom=refget(simcore,transfrom)
    transtotemp=refout(objtotransfrom,protocol)
    (simcore,transto)=refset(transtotemp)
    goto(next)

process(refin)
% #writeto = refin(#readfrom,#protocolname)
    transto=get(simfocus,arg1)
    transfrom=get(simfocus,arg2)
    protocolname=get(simfocus,arg3)
    protocol=refget(simcore,protocolname)
    objtotransfrom=refget(simcore,transfrom)
    transtotemp=refin(objtotransfrom,protocol)
    (simcore,transto)=refset(transtotemp)
    goto(next)

process(refout)
% #writeto = refout(#readfrom,#protocolname)
    transto=get(simfocus,arg1)
    transfrom=get(simfocus,arg2)
    protocolname=get(simfocus,arg3)
    protocol=refget(simcore,protocolname)
    objtotransfrom=refget(simcore,transfrom)
    transtotemp=refout(objtotransfrom,protocol)
    (simcore,transto)=refset(transtotemp)
```

```
    goto(next)

% info to start program:
start(init)
```

Appendix B

Typesheets

B.1 The typesheet for expressions

The following is the typesheet that defines all the types that can be considered as operators to form expressions:

```
Expressions::

! Expression types
! ----------------
!
! Peter Schodl
!
! Feb 21, 2011
!
! This type system defines the types needed to process expression.

! To type tighter:
!!! Make Term and Expression
!!! type narrower: from AND to  /  over  (a new type FromTo)


Alternative:
  allOf> linkedList = AlternativeLink

AlternativeLink:
  allOf> entry = Expression
  optional> next = AlternativeLink

Bracket:
 allOf> entry = Expression

Cases:
  allOf> linkedList = CasesLink
  optional> otherwise = Expression

CasesLink:
  allOf> formula = Expression
         condition = Expression
  optional> next = CasesLink

Chain:
```

```
  allOf> firstrel = Expression
         linkedList = ExpLink
ExpLink:
  allOf> entry = Expression
  optional> next = ExpLink

Diag:
  allOf> linkedList = ExpLink

Div:
  allOf> nom = Expression
         den = Expression

Dummy:
  allOf> entry = Expression

Equal:
  allOf> lhs = Expression
         rhs = Expression
  optional> above = Text

Eval:
  allOf> formula = Expression
         binds = VarList
  optional> index = Expression
            from = Expression
            to = Expression

Forall:
  allOf> formula = Expression
         scopedvar = Expression
         binds = VarList

InvisMult:
  allOf> linkedList = ExpLink

Interval:
  allOf> lower = Expression
         upper = Expression

Integral:
  allOf> formula = Expression
         variable = IndexedVar
         binds = VarList
  optional> index = Expression
            from = Expression
            to = Expression

List:
   allOf> linkedList = ExpLink
   optional> leftBr = Brackets
             separator = Separators
             rightBr = Brackets

Lambda:
  allOf> formula = Expression
         variable = IndexedVar
         binds = VarList

Max:
  allOf> formula = Expression
  optional> binds = VarList
```

```
            index = Expression

Min:
  allOf> formula = Expression
  optional> binds = VarList
            index = Expression

Mid:
  allOf> lhs = Expression
         rhs = Expression

Matrix:
  allOf> linkedList = RowLink

Norm:
  allOf> formula = Expression
  optional> index = Expression

Of:
  allOf> operator = Expression
         arguments = Expression

Or:
  allOf> linkedList = ExpLink

OtherInterval:
  someOf> lowerclosed = Expression
          loweropen = Expression
          upperopen = Expression
          upperclosed = Expression

Partial:
  allOf> linkedList = ExpLink

Prime:
  allOf> entry = Expression

Power:
  allOf> base = Expression
         exponent = Expression

Prob:
  allOf> event = Expression
  optional> condition = Expression

Row:
  allOf> linkedList = ExpLink

RowLink:
  allOf> entry = Row
  optional> next = RowLink

Restriction:
  allOf> formula = Expression
         restriction = Expression
  optional> binds = VarList
            if = Expression
            forsome = Expression

Relation:
  allOf> lhs = RelationLhs
         rhs = Expression
```

```
        relation = RelationSymbols
  optional> above = Text

Script:
  allOf> formula = Expression
  someOf> sub = Expression
          sup = Expression
          lsup = Expression
          lsub = Expression

Sqrt:
  allOf> radicand = Expression

Set2Exp:
  allOf> lhs = Expression
         rhs = Expression
  optional> binds = VarList

Sum:
  allOf> formula = Expression
         binds = VarList
  someOf> index = Expression
          from = Expression
          to = Expression

Set:
  allOf> scopedvar = Expression
         condition = Expression
  optional> binds = VarList

SetUnion:
  allOf> linkedList = ExpLink

SetProduct:
  allOf> linkedList = ExpLink

SetBucket:
  allOf> linkedList = ExpLink

SignedSum:
  allOf> linkedList = SignedSumLink

SignedSumLink:
  allOf> sign = Signs
         entry = Expression
  optional> next = SignedSumLink

Text:
  allOf> entry = Object

Var:
  nothingElse>
  optional> name = String

VarList:
  itself> IndexedVar

Vector:
  allOf> linkedList = ExpLink

Separators:
atomic> SepKomma,SepColon,SepSemicolon,SepBlank,None
```

```
Brackets:
atomic> BrLeftRound,BrRightRound,BrLeftSquare,BrRightSquare,None

RelationSymbols:
atomic> LessEq,Less,In,Greater,GreaterEq,EqualByDef,EqualSign

Signs:
atomic> InvisPlusSign,MinusSign,PlusSign

Expression:
  union> Alternative,Bracket,Cases,Chain,Diag,Div,Dummy,Equal
  union> Eval,Forall,InvisMult,Interval,Integral,List,Lambda,Max
  union> Min,Mid,Matrix,Norm,Of,Or,OtherInterval,Partial,Prime
  union> Power,Prob,Restriction,Relation,Script,Sqrt,Set2Exp,Sum
  union> Set,SetBucket,SetProduct,SignedSum,SetUnion,Var,Vector,Dummy
  union> String,Integer,Double,Separators,Signs,RelationSymbols

RelationLhs:
  union> Expression, VarList

IndexedVar:
  union> Var, Script
```

B.2 Typesheets for a Turing machine

Furthermore, we give type sheets for tapes and transition tables for Turing machines as represented in the SM and discussed in Section 3.7.

```
TuringMachine::

Tape:
  allOf> entry = Object
  optional> last = Tape
            next = Tape
  nothingElse>

Movement:
  atomic> L, R, X

TransTable:
  allOf> state = Object
         tostate = Object
         tomove = Movement
         towrite = Object
         reading = Object
  optional> next=TransTable
  nothingElse>
```

B.3 Type sheet for optimization problems

```
ORLibTypes::

Action:
  allOf> verb = String
         subject = Expression

AlgCode:
  allOf> linkedList = AlgCodeLink

AlgCodeLink:
  allOf> entry = AlgorithmStep
  optional> next = AlgCodeLink

AlgFor:
  allOf> from = Expression
         to = Expression
         running = Var
         linkedList = AlgCodeLink

AlgSet:
  allOf> entry = Expression
         setto = Expression

AlgInstance:
  allOf> linkedList = AlgInstanceLink

AlgInstanceLink:
  allOf> entry = AlgorithmStep
  optional> next = AlgInstanceLink

AlgRead:
  allOf> entry = Expression

AlgInc:
  allOf> entry = Expression

AlgReadSilent:
  allOf> entry = Expression

AlgUntilEOF:
  allOf> linkedList = AlgCodeLink

AlgUntilEOL:
  allOf> linkedList = AlgCodeLink

Concept:
  allOf> entry = SentencePart
  optional> specification = SentencePart
            adjective = ConceptGeneral

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

ConstraintList:
  allOf> linkedList = ConstraintListLink

ConstraintListLink:
  allOf> entry = Constraint
```

```
  optional> next = ConstraintListLink

Card:
  allOf> entry = Expression

Definition:
  allOf> defined = TextUnit
         definedas = Objects
  optional> with = Objects
            given = Expression

DefinitionRel:
  allOf> definethat = TextUnit
  optional> with = Expression
            given = Objects
            iff = Statement
            otherwise = TextUnit

Document:
  allOf> linkedList = DocumentLink
  optional> header = TextUnit
            mod = Problem
            dat = ORdata

DocumentInclude:
  allOf> linkedList = DocumentLink
  optional> header = TextUnit
            mod = Problem
            dat = ORdata

DocumentLink:
  allOf> entry = TextUnit
  optional> next = DocumentLink

Equivalent:
  allOf> lhs = Statement
         rhs = Statement

Files:
   allOf> linkedList = FilesLink

IncludeMod:
   allOf> entry = String

FilesLink:
  allOf> entry = String
  optional> next = FilesLink

Let:
  someOf> subject = Objects
          statement = ObjConc
          description = ConceptGeneral
  optional> qualification = Qualification

MatrixOfDim:
  allOf> rows = Expression
         columns = Expression
         rowindex = Var
         colindex = Var

ORdata:
  someOf> filenames = Files
```

```
          nrproblems = Expression
          algorithm = AlgCode

ORdataSolution:
  optional> filenames = Files
            algorithm = AlgCode

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

ParagraphList:
  allOf> linkedList = ParagraphListLink

ParagraphListLink:
  allOf> entry = TextUnit
  optional> next = ParagraphListLink

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

Qualification:
  optional> of = ObjConc

Quantification:
  someOf> quantity = Expression
          hyphenobj = ConceptGeneral
          object = ConceptGeneral

Sentence:
  optional> linkedList = SentencePartLink

SentenceLink:
  allOf> entry = Sentence
  optional> next = SentenceLink

SentencePartLink:
  allOf> entry = SentencePart
  optional> next = SentencePartLink

SentenceList:
  allOf> linkedList = SentenceLink
  optional> header = TextUnit

Statement:
  allOf> subject = Objects
```

```
  oneOf> isa = ConceptGeneral
         is = TextUnit
         has = TextUnit

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

VectorOfDim:
  allOf> dimension = Expression
         indexvar = Var

!!!!!!!!!!     ATOMICS AND UNIONS


AlgAtomics:
  atomic> ReadInstances, NewLine

PropertiesAtomics:
  atomic> NonNegative,Binary,Positive

ConceptAtomics:
  atomic> SetAsConcept,IntegerAsConcept,NumberAsConcept,VectorAsConcept
  atomic> MatrixAsConcept,RealNumberAsConcept,NaturalNumbersAsConcept
  atomic> EmptySetAsConcept,InfinityAsConcept,RealNumbersAsConcept
  atomic> SequenceAsConcept

MinMax:
  atomic> Minimize, Maximize

AlgorithmStep:
  union> AlgRead, AlgFor, AlgInstance, AlgUntilEOF, AlgUntilEOL
  union> AlgReadSilent, AlgAtomics, AlgSet, AlgInc

Objects:
  union> ObjList, Obj, Expression

Constraints:
  union> ConstraintList, Constraint

TextUnit:
  union> Sentence, String, SentenceList, ParagraphList
  union> PropertiesAtomics, Concept, IncludeMod, Objects

ConceptGeneral:
  union> Concept, String, Quantification, PropertiesAtomics, ConceptAtomics
  union> VectorOfDim, MatrixOfDim, Sequence

SentencePart:
  union> ConceptGeneral, Expression, Obj, Definition, Let
  union> ORdata, Problem, Action, ORdataSolution
  union> Statement, Equivalent, DefinitionRel

Condition:
  union> Statement

ObjConc:
  union> Objects, Concept
```

B.4 Type sheet for TPTP problems

```
TPTP::

ArgumentLink:
  allOf> entry = ArgumentEntry
  optional> next = ArgumentLink

AxiomList:
  allOf> entry = String
  optional> next = AxiomList

ForAll:
  allOf> entry = ArgumentEntry
         binds = VarList

Exists:
  allOf> entry = ArgumentEntry
         binds = VarList

Equal:
  allOf> arguments = ArgumentLink

NotEqual:
  allOf> arguments = ArgumentLink

Iff:
  allOf> arguments = ArgumentLink

NotIff:
  allOf> arguments = ArgumentLink

Implies:
  allOf> arguments = ArgumentLink

Requires:
  allOf> arguments = ArgumentLink

Equivalent:
  allOf> arguments = ArgumentLink

NotEquivalent:
  allOf> arguments = ArgumentLink

Or:
  allOf> arguments = ArgumentLink

NotOr:
  allOf> arguments = ArgumentLink

And:
  allOf> arguments = ArgumentLink

NotAnd:
  allOf> arguments = ArgumentLink

Not:
  allOf> entry = ArgumentEntry

Of:
  allOf> arguments = ArgumentLinkOrEntry
         operator = String
```

```
VarList:
  someOfType> String = String

TptpFiles:
  someOfType> String = TptpFile

TptpFile:
  allOf> domain = Domains
         extension = String
         filename = String
  optional> axioms = AxiomList
            formulas = TptpFormula

TptpFormula:
  allOf> entry = ArgumentEntry
         form = Forms
         name = String
         role = Roles
  optional> next = TptpFormula

ArgumentEntry:
  union> String,Not,Equal,NotEqual,Iff,NotIff
  union> Implies,Requires,Or,NotOr,And,NotAnd,Of
  union> Exists,ForAll,Equivalent,NotEquivalent

ArgumentLinkOrEntry:
  union> ArgumentLink,ArgumentEntry

Domains:
  atomic> AGT,ALG,ANA,BOO,CAT,COL,COM,CSR,FLD,GEO,GRA,GRP
  atomic> HAL,HEN,HWC,HWV,KRS,LAT,LCL,LDA,MED,MGT,MSC,NLP
  atomic> NUM,PLA,PUZ,RNG,ROB,SET,SEU,SWC,SWV,TOP

Forms:
  atomic> fof,cnf,thf

Roles:
  atomic> axiom,hypothesis,definition,assumption,lemma,theorem
  atomic> conjecture,negated_conjecture,plain,fi_domain
  atomic> fi_functors,fi_predicates,type,unknown
```

Appendix C

Examples of expressions

The running numbers of the examples correspond to Table 5.1 on page 89.
For each example, we give the automatically produced and rendered LaTeXoutput, the raw LaTeXoutput, the list of sems and values that represent the expression, and a semantic graph.

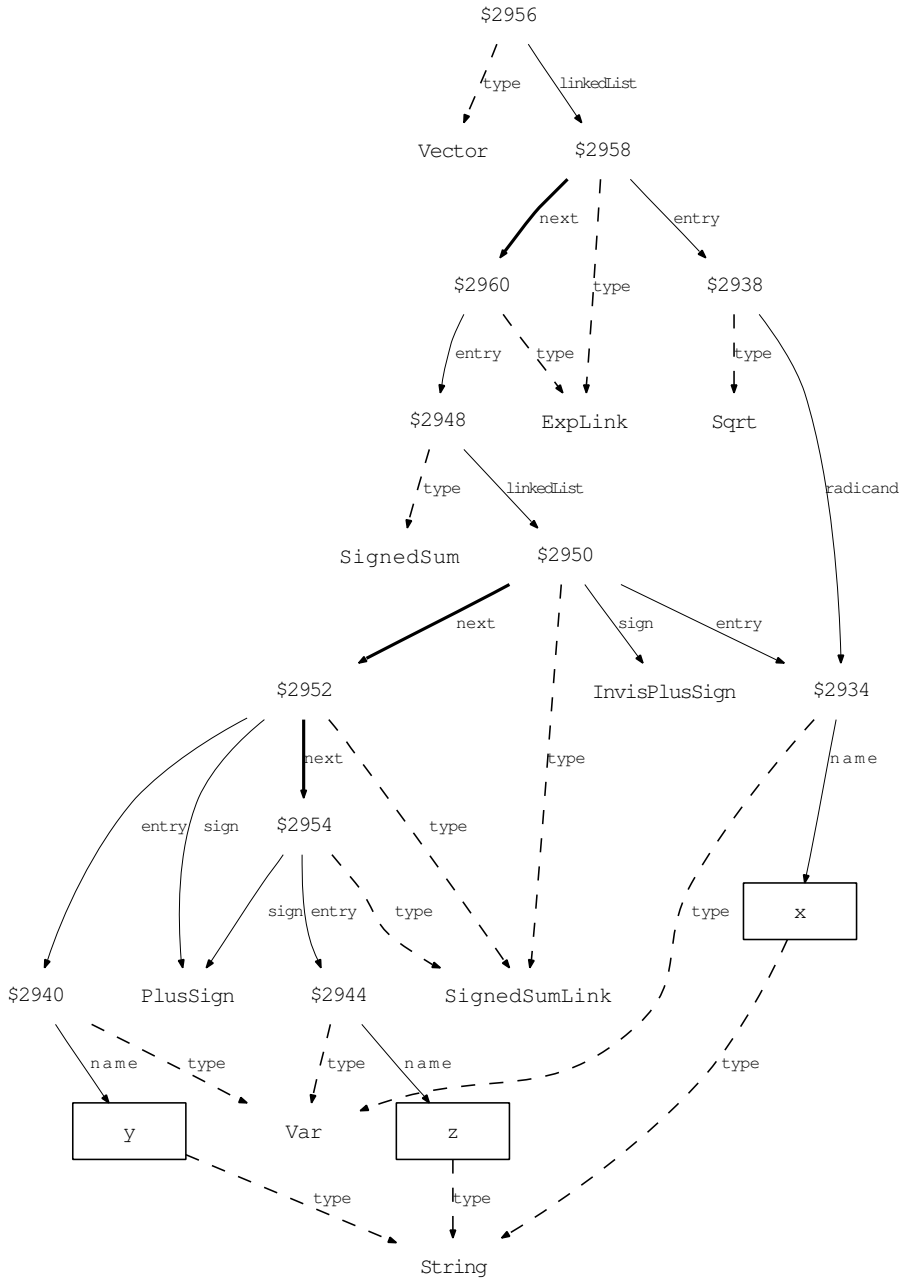
C.1 Vectors and matrices

Example 1.

$$\left(\sqrt{x}, x + y + z\right)$$

```
1 \left(\sqrt{ x } , x + y + z \right)
```

```
$2956.type=Vector
$2956.linkedList=$2958
$2958.type=ExpLink
$2958.next=$2960
$2958.entry=$2938
$2938.type=Sqrt
$2938.radicand=$2934
$2934.type=Var
$2934.name=$2936
$2936.type=String
$2960.type=ExpLink
$2960.entry=$2948
$2948.type=SignedSum
$2948.linkedList=$2950
$2950.type=SignedSumLink
$2950.next=$2952
$2950.sign=InvisPlusSign
$2950.entry=$2934
$2952.type=SignedSumLink
$2952.next=$2954
$2952.sign=PlusSign
$2952.entry=$2940
$2940.type=Var
$2940.name=$2942
$2942.type=String
$2954.type=SignedSumLink
$2954.sign=PlusSign
$2954.entry=$2944
$2944.type=Var
$2944.name=$2946
$2946.type=String
VALUE($2936) = x
VALUE($2942) = y
VALUE($2946) = z
```

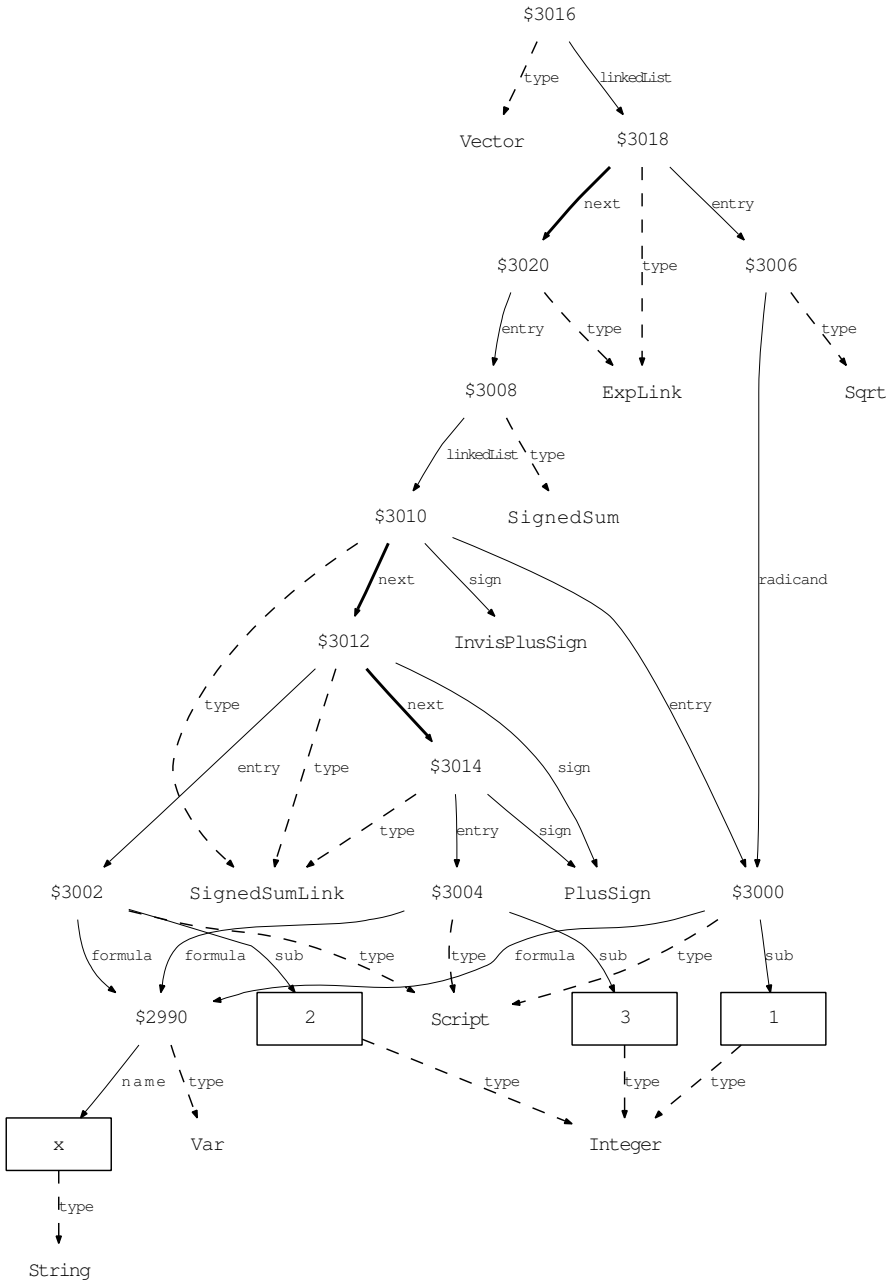

$2956
type
linkedList
Vector
$2958
next
entry
$2960
type
$2938
entry
type
type
$2948
ExpLink
Sqrt
type
linkedList
radicand
SignedSum
$2950
next
sign
entry
$2952
InvisPlusSign
$2934
next
type
name
entry
sign
$2954
type
sign
entry
type
type
x
$2940
PlusSign
$2944
SignedSumLink
name
type
type
name
type
y
Var
z
type
type
String

Example 2. Similar as before, but now the only variable is the vector x.

$$(\sqrt{x_1}, x_1 + x_2 + x_3)$$

```
\left(\sqrt{ {}{x}_{ 1 } } , {}{x}_{ 1 } + {}{x}_{
   2 } + {}{x}_{ 3 } \right)
```

```
$3016.type=Vector
$3016.linkedList=$3018
$3018.type=ExpLink
$3018.next=$3020
$3018.entry=$3006
$3006.type=Sqrt
$3006.radicand=$3000
$3000.type=Script
$3000.formula=$2990
$3000.sub=$2994
$2990.type=Var
$2990.name=$2992
$2992.type=String
$2994.type=Integer
$3020.type=ExpLink
$3020.entry=$3008
$3008.type=SignedSum
$3008.linkedList=$3010
$3010.type=SignedSumLink
$3010.next=$3012
$3010.sign=InvisPlusSign
$3010.entry=$3000
$3012.type=SignedSumLink
$3012.next=$3014
$3012.sign=PlusSign
$3012.entry=$3002
$3002.type=Script
$3002.formula=$2990
$3002.sub=$2996
$2996.type=Integer
$3014.type=SignedSumLink
$3014.sign=PlusSign
$3014.entry=$3004
$3004.type=Script
$3004.formula=$2990
$3004.sub=$2998
$2998.type=Integer
VALUE($2992) = x
VALUE($2994) = 1
VALUE($2996) = 2
VALUE($2998) = 3
```

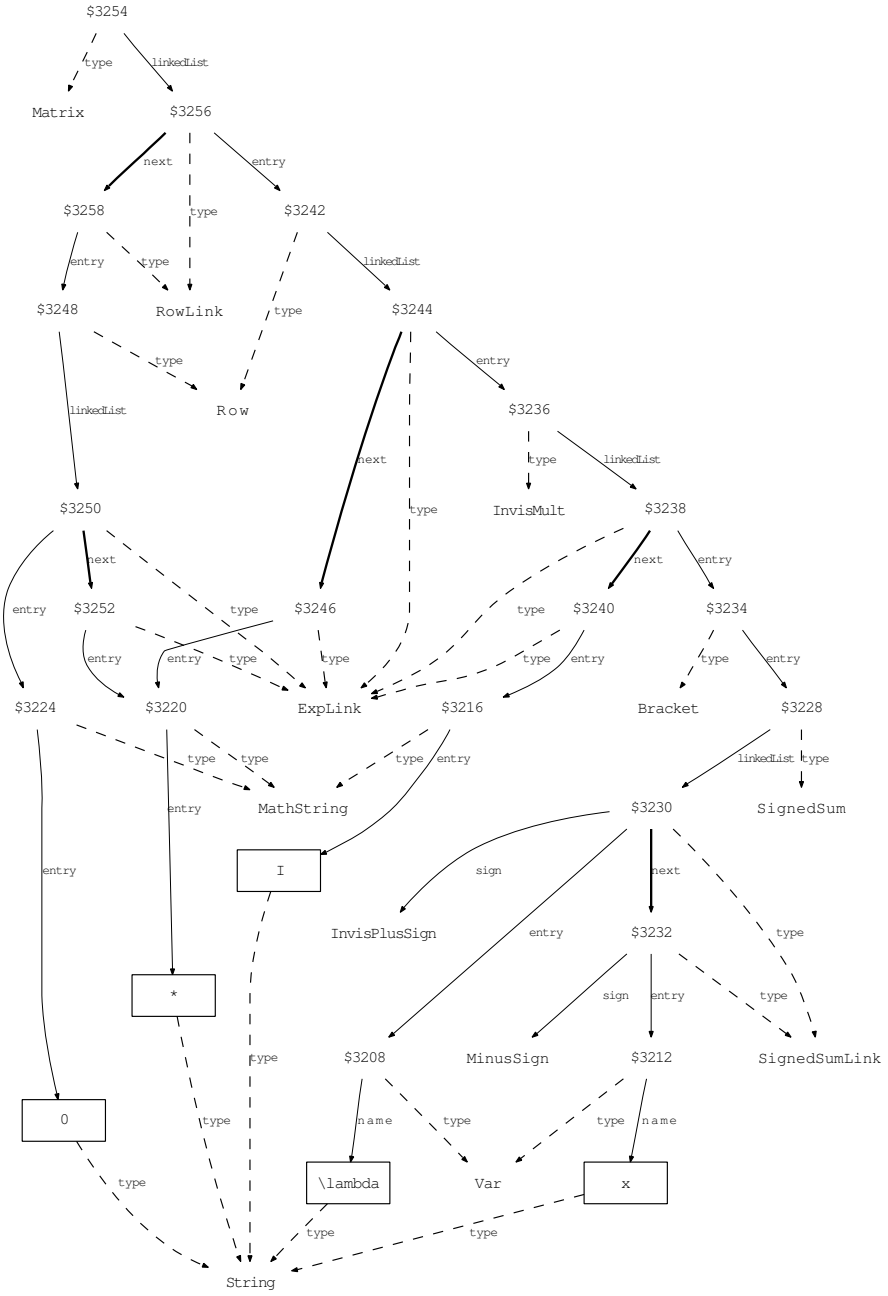

$3016
type
linkedList
Vector
$3018
next
entry
$3020
type
$3006
entry
type
type
$3008
ExpLink
Sqrt
linkedList
type
$3010
SignedSum
next
sign
radicand
$3012
InvisPlusSign
type
next
entry
entry
type
$3014
sign
type
entry
sign
$3002
SignedSumLink
$3004
PlusSign
$3000
formula
formula
sub
type
type
formula
sub
type
sub
$2990
2
Script
3
1
name
type
type
type
type
x
Var
Integer
type
String

Example 3. A matrix with wildcard characters, denoted by $*$.

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

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

```
$3254.type=Matrix
$3254.linkedList=$3256
$3256.type=RowLink
$3256.next=$3258
$3256.entry=$3242
$3242.type=Row
$3242.linkedList=$3244
$3244.type=ExpLink
$3244.next=$3246
$3244.entry=$3236
$3236.type=InvisMult
$3236.linkedList=$3238
$3238.type=ExpLink
$3238.next=$3240
$3238.entry=$3234
$3234.type=Bracket
$3234.entry=$3228
$3228.type=SignedSum
$3228.linkedList=$3230
$3230.type=SignedSumLink
$3230.next=$3232
$3230.sign=InvisPlusSign
$3230.entry=$3208
$3208.type=Var
$3208.name=$3210
$3210.type=String
$3232.type=SignedSumLink
$3232.sign=MinusSign
$3232.entry=$3212
$3212.type=Var
$3212.name=$3214
$3214.type=String
$3240.type=ExpLink
$3240.entry=$3216
$3216.type=MathString
$3216.entry=$3218
$3218.type=String
$3246.type=ExpLink
$3246.entry=$3220
$3220.type=MathString
$3220.entry=$3222
$3222.type=String
$3258.type=RowLink
$3258.entry=$3248
$3248.type=Row
$3248.linkedList=$3250
$3250.type=ExpLink
$3250.next=$3252
$3250.entry=$3224
$3224.type=MathString
$3224.entry=$3226
$3226.type=String
$3252.type=ExpLink
$3252.entry=$3220
VALUE($3210) = \lambda
VALUE($3214) = x
VALUE($3218) = I
VALUE($3222) = *
VALUE($3226) = 0
```

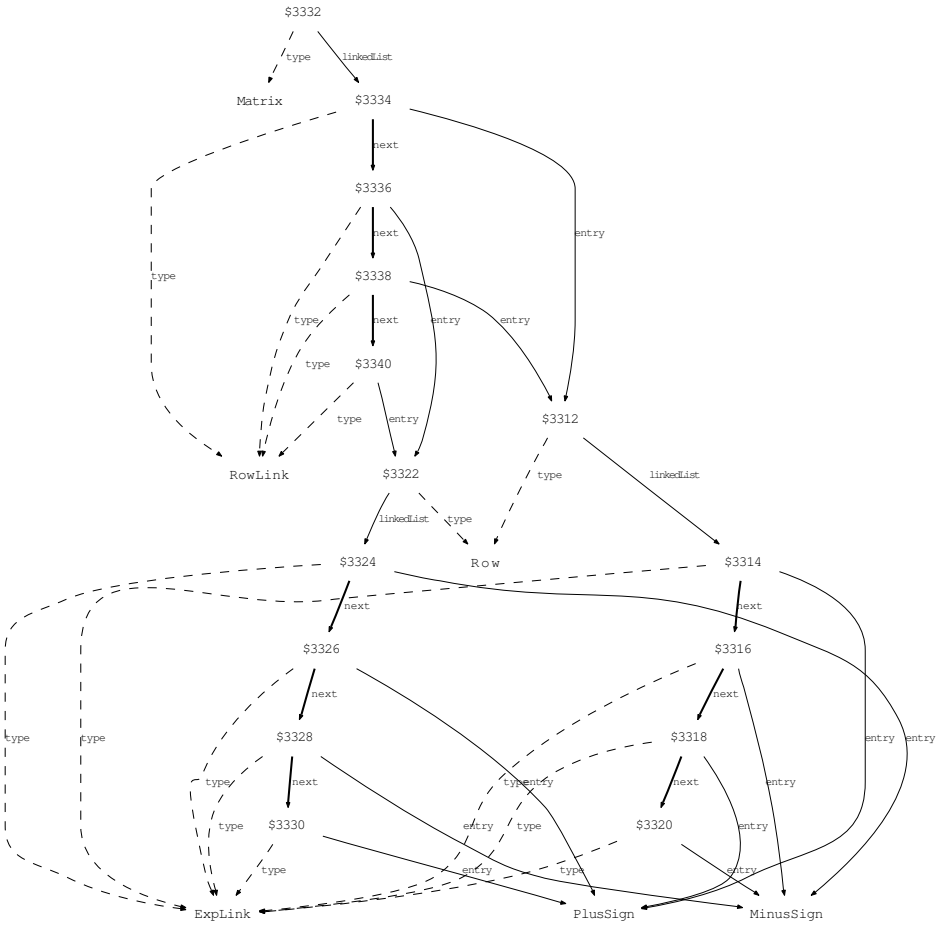

$3254
type
linkedList
Matrix
$3256
next
entry
$3258
type
$3242
entry
type
linkedList
$3248
RowLink
type
$3244
type
type
entry
linkedList
Row
$3236
next
type
linkedList
$3250
type
InvisMult
$3238
next
entry
next
$3252
type
$3246
type
$3240
$3234
entry
entry
type
type
type
type
entry
type
entry
$3224
$3220
ExpLink
$3216
Bracket
$3228
type
type
type
entry
linkedList
type
MathString
SignedSum
entry
entry
$3230
I
sign
next
InvisPlusSign
entry
$3232
type
*
sign
entry
type
type
$3208
MinusSign
$3212
SignedSumLink
0
type
name
type
type
name
\lambda
Var
x
type
type
type
String

Example 4.

$$\left(\begin{array}{cccc} + & - & + & - \\ - & + & - & + \\ + & - & + & - \\ - & + & - & + \end{array}\right)$$

```
\left(\begin{array}{cccc} + & - & + & - \\ - & + & - & +
    & + \\ + & - & + & - \\ - & + & - & + \\ \end{
   array} \right)
```

```
$3332.type=Matrix
$3332.linkedList=$3334
$3334.type=RowLink
$3334.next=$3336
$3334.entry=$3312
$3312.type=Row
$3312.linkedList=$3314
$3314.type=ExpLink
$3314.next=$3316
$3314.entry=PlusSign
$3316.type=ExpLink
$3316.next=$3318
$3316.entry=MinusSign
$3318.type=ExpLink
$3318.next=$3320
$3318.entry=PlusSign
$3320.type=ExpLink
$3320.entry=MinusSign
$3336.type=RowLink
$3336.next=$3338
$3336.entry=$3322
$3322.type=Row
$3322.linkedList=$3324
$3324.type=ExpLink
$3324.next=$3326
$3324.entry=MinusSign
$3326.type=ExpLink
$3326.next=$3328
$3326.entry=PlusSign
$3328.type=ExpLink
$3328.next=$3330
$3328.entry=MinusSign
$3330.type=ExpLink
$3330.entry=PlusSign
$3338.type=RowLink
$3338.next=$3340
$3338.entry=$3312
$3340.type=RowLink
$3340.entry=$3322
```

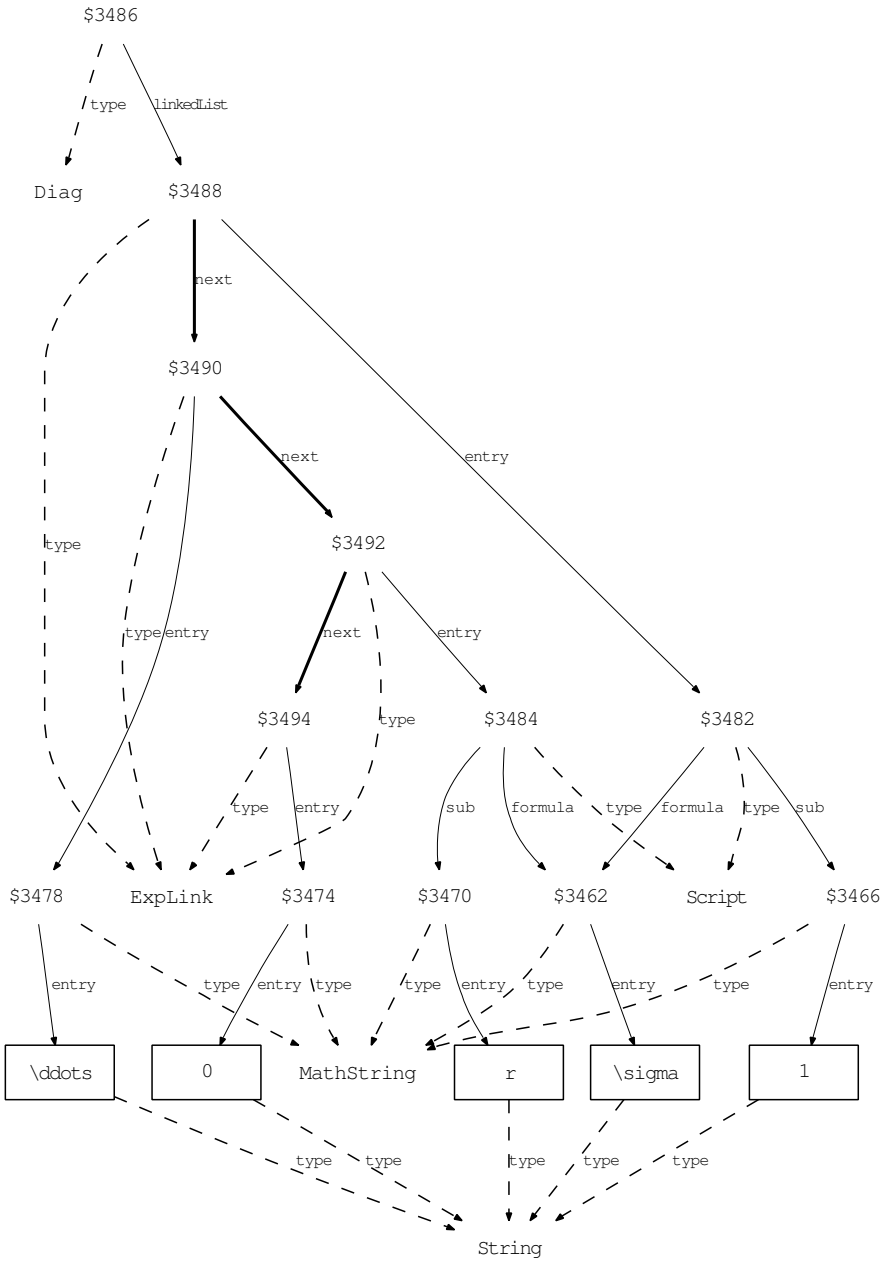

$3332
type
linkedList
Matrix
$3334
next
$3336
next
entry
type
$3338
type
next
entry
entry
type
$3340
type
entry
$3312
RowLink
$3322
type
linkedList
linkedList
type
$3324
Row
$3314
next
next
$3326
$3316
next
next
type
type
$3328
$3318
entry
entry
type
next
type
entry
next
entry
type
$3330
entry
type
$3320
entry
type
entry
type
entry
ExpLink
PlusSign
MinusSign

Example 5.

$$\left(\begin{array}{cccc} {}{\sigma}_{1}\\&\ddots \\&&{}{\sigma}_{r}\\&&&0\\ \end{array} \right)$$

```
\left(\begin{array}{cccc} {}{\sigma}_{1}\\&\ddots
   \\&&{}{\sigma}_{r}\\&&&0\\ \end{array} \right)
```

```
$3486.type=Diag
$3486.linkedList=$3488
$3488.type=ExpLink
$3488.next=$3490
$3488.entry=$3482
$3482.type=Script
$3482.formula=$3462
$3482.sub=$3466
$3462.type=MathString
$3462.entry=$3464
$3464.type=String
$3466.type=MathString
$3466.entry=$3468
$3468.type=String
$3490.type=ExpLink
$3490.next=$3492
$3490.entry=$3478
$3478.type=MathString
$3478.entry=$3480
$3480.type=String
$3492.type=ExpLink
$3492.next=$3494
$3492.entry=$3484
$3484.type=Script
$3484.formula=$3462
$3484.sub=$3470
$3470.type=MathString
$3470.entry=$3472
$3472.type=String
$3494.type=ExpLink
$3494.entry=$3474
$3474.type=MathString
$3474.entry=$3476
$3476.type=String
VALUE($3464) = \sigma
VALUE($3468) = 1
VALUE($3472) = r
VALUE($3476) = 0
VALUE($3480) = \ddots
```


$3486
type
linkedList
Diag
$3488
next
$3490
next
entry
type
$3492
type
entry
next
entry
$3494
$3484
$3482
type
type
entry
sub
formula
type
formula
type
sub
$3478
ExpLink
$3474
$3470
$3462
Script
$3466
entry
type
entry
type
type
entry
type
entry
type
entry
\ddots
0
MathString
r
\sigma
1
type
type
type
type
type
String

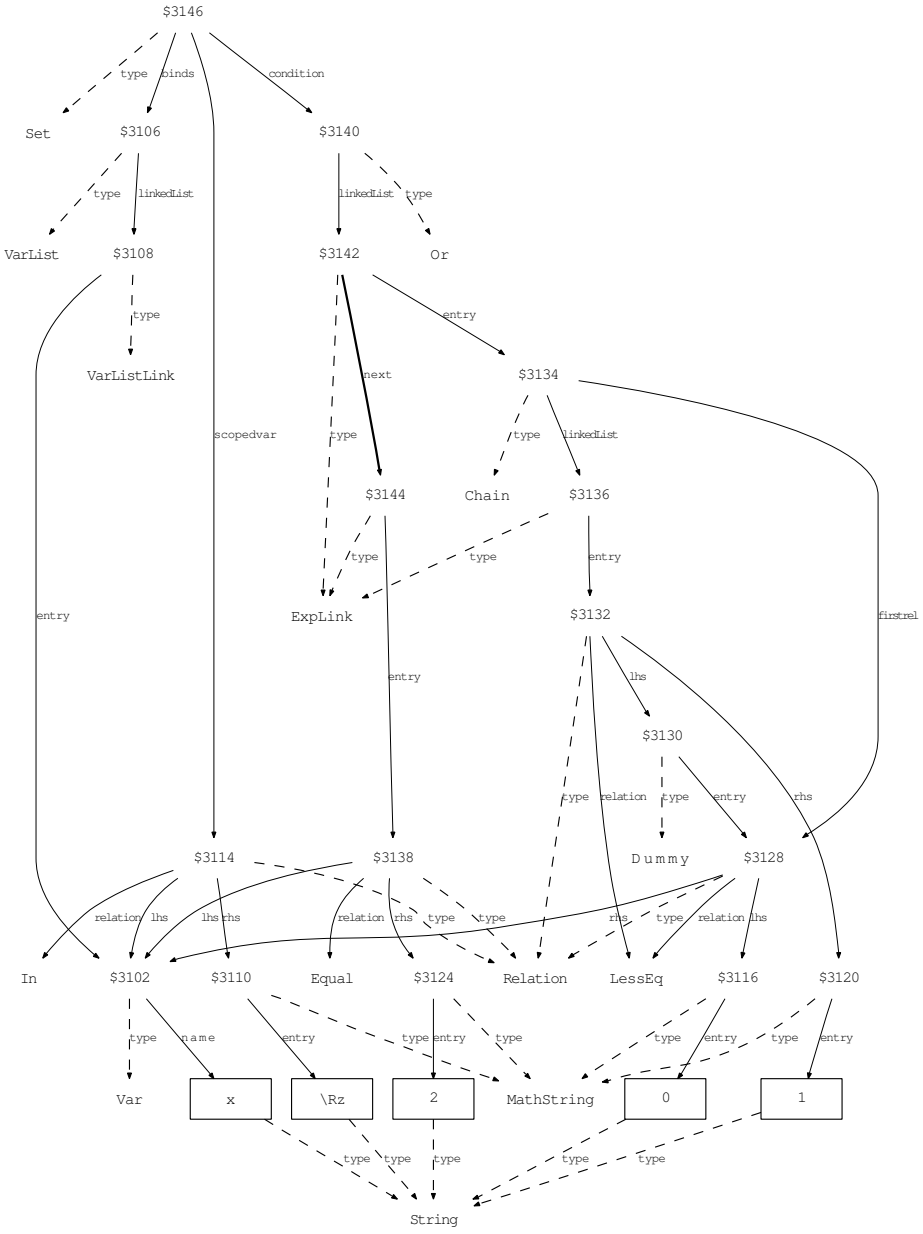
C.2 Sets

Example 6.

$$\{x{\in}\mathbb{R} \mid 0{\leq}x{\leq}1 \vee x{=}2\}$$

```
\left\{x{ \in }\Rz \mid 0{ \leq }x{ \leq }1 \vee x
   {{=}}2\right\}
```

```
$3146.type=Set
$3146.scopedvar=$3114
$3146.binds=$3106
$3146.condition=$3140
$3106.type=VarList
$3106.linkedList=$3108
$3108.type=VarListLink
$3108.entry=$3102
$3102.type=Var
$3102.name=$3104
$3104.type=String
$3114.type=Relation
$3114.lhs=$3102
$3114.relation=In
$3114.rhs=$3110
$3110.type=MathString
$3110.entry=$3112
$3112.type=String
$3140.type=Or
$3140.linkedList=$3142
$3142.type=ExpLink
$3142.next=$3144
$3142.entry=$3134
$3134.type=Chain
$3134.firstrel=$3128
$3134.linkedList=$3136
$3128.type=Relation
$3128.lhs=$3116
$3128.relation=LessEq
$3128.rhs=$3102
$3116.type=MathString
$3116.entry=$3118
$3118.type=String
$3136.type=ExpLink
$3136.entry=$3132
$3132.type=Relation
$3132.lhs=$3130
$3132.relation=LessEq
$3132.rhs=$3120
$3120.type=MathString
$3120.entry=$3122
$3122.type=String
$3130.type=Dummy
$3130.entry=$3128
$3144.type=ExpLink
$3144.entry=$3138
$3138.type=Relation
$3138.lhs=$3102
$3138.relation=Equal
$3138.rhs=$3124
$3124.type=MathString
$3124.entry=$3126
$3126.type=String
VALUE($3104) = x
VALUE($3112) = \Rz
VALUE($3118) = 0
VALUE($3122) = 1
VALUE($3126) = 2
```

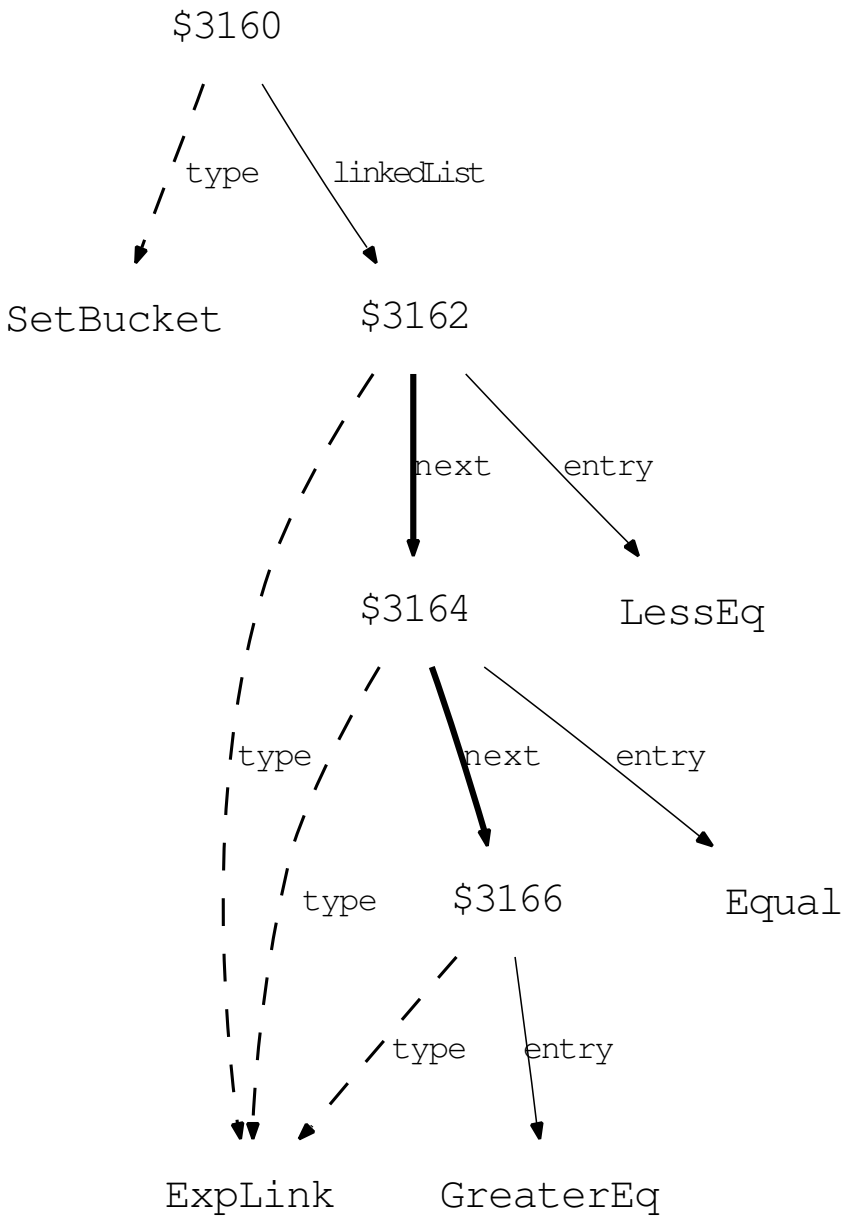

$3146
type
binds
condition
Set
$3106
$3140
type
linkedList
linkedList
type
VarList
$3108
$3142
Or
type
entry
VarListLink
$3134
next
scopedvar
type
type
linkedList
$3144
Chain
$3136
type
type
entry
entry
ExpLink
$3132
firstrel
entry
lhs
$3130
type
relation
type
entry
rhs
$3114
$3138
Dummy
$3128
relation
lhs
lhs
rhs
relation
rhs
type
type
rhs
type
relation
lhs
In
$3102
$3110
Equal
$3124
Relation
LessEq
$3116
$3120
type
name
entry
type
entry
type
type
entry
type
entry
Var
x
\Rz
2
MathString
0
1
type
type
type
type
type
String

Example 7. A set as list with characters as entries.

$$\{\leq, =, \geq\}$$

```
\left\{ \leq , {=} , \geq \right\}
```

```
$3160.type=SetBucket
$3160.linkedList=$3162
$3162.type=ExpLink
$3162.next=$3164
$3162.entry=LessEq
$3164.type=ExpLink
$3164.next=$3166
$3164.entry=Equal
$3166.type=ExpLink
$3166.entry=GreaterEq
```

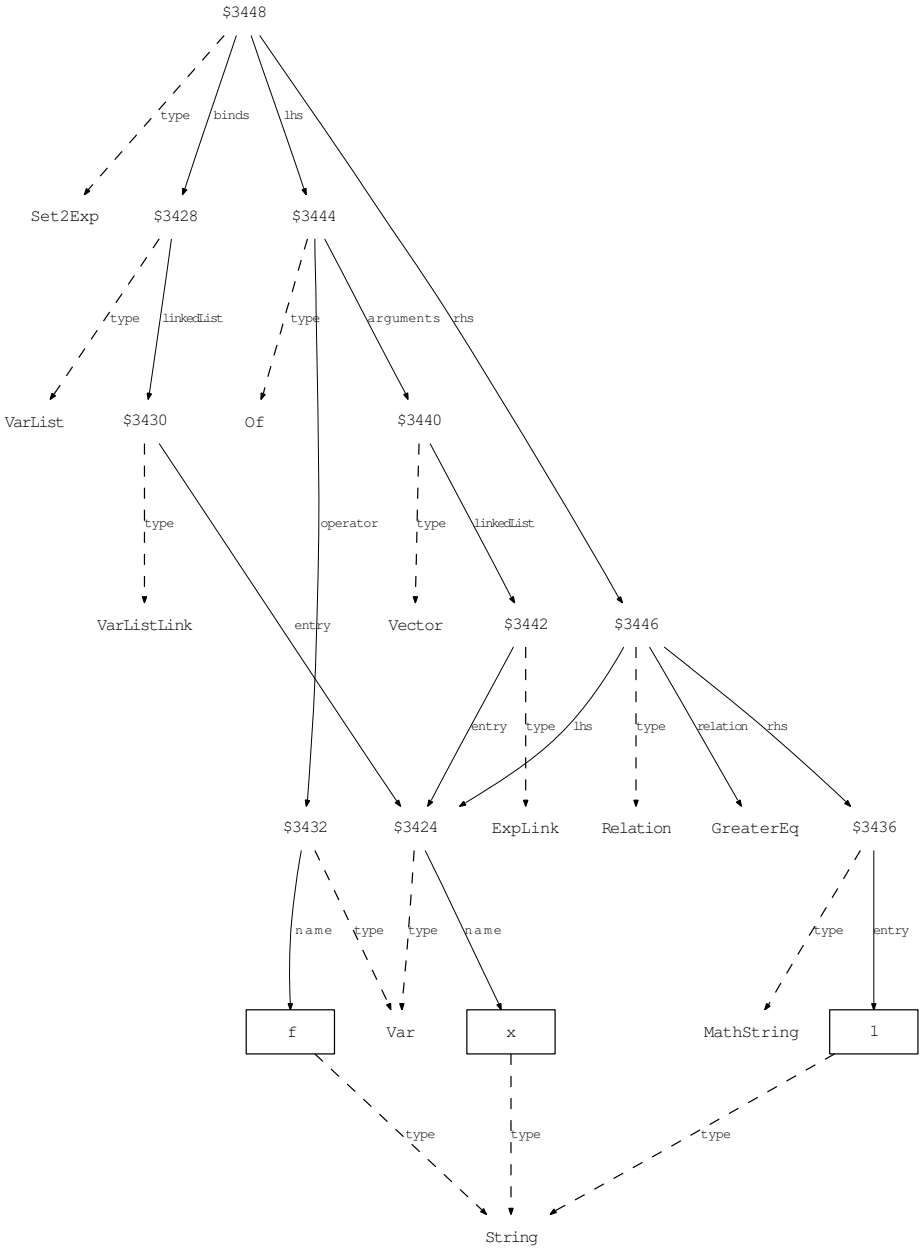

$3160
type
linkedList
SetBucket
$3162
next
entry
$3164
LessEq
type
next
entry
type
$3166
Equal
type
entry
ExpLink
GreaterEq

Example 8.

$$\{f\left(x\right) \mid x \geq 1\}$$

```
\left\{ f \left(x\right) \mid x{ \geq }1 \right\}
```

```
$3448.type=Set2Exp
$3448.binds=$3428
$3448.lhs=$3444
$3448.rhs=$3446
$3428.type=VarList
$3428.linkedList=$3430
$3430.type=VarListLink
$3430.entry=$3424
$3424.type=Var
$3424.name=$3426
$3426.type=String
$3444.type=Of
$3444.operator=$3432
$3444.arguments=$3440
$3432.type=Var
$3432.name=$3434
$3434.type=String
$3440.type=Vector
$3440.linkedList=$3442
$3442.type=ExpLink
$3442.entry=$3424
$3446.type=Relation
$3446.lhs=$3424
$3446.relation=GreaterEq
$3446.rhs=$3436
$3436.type=MathString
$3436.entry=$3438
$3438.type=String
VALUE($3426) = x
VALUE($3434) = f
VALUE($3438) = 1
```

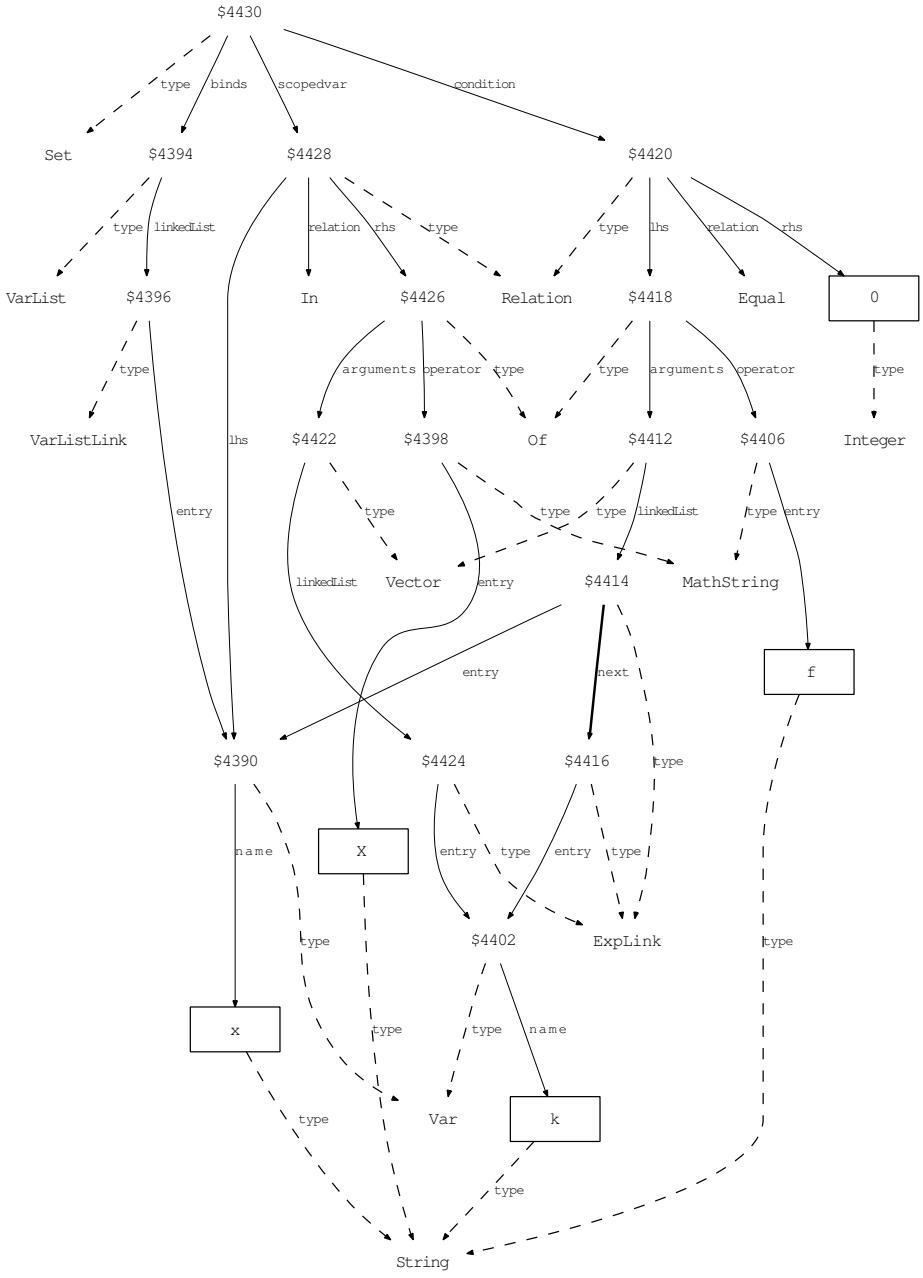

$3448
type
binds
lhs
Set2Exp
$3428
$3444
type
linkedList
type
arguments
rhs
VarList
$3430
Of
$3440
type
operator
type
linkedList
VarListLink
entry
Vector
$3442
$3446
entry
type
lhs
type
relation
rhs
$3432
$3424
ExpLink
Relation
GreaterEq
$3436
name
type
type
name
type
entry
f
Var
x
MathString
1
type
type
type
String

Example 9.

$$\{x{\in}X\,(k)\mid f\,(x,k)=0\}$$

```
\left\{x{ \in }X \left(k\right) \mid f \left(x , k\
   right) {{=}} 0 \right\}
```

```
$4430.type=Set
$4430.scopedvar=$4428
$4430.binds=$4394
$4430.condition=$4420
$4394.type=VarList
$4394.linkedList=$4396
$4396.type=VarListLink
$4396.entry=$4390
$4390.type=Var
$4390.name=$4392
$4392.type=String
$4420.type=Relation
$4420.lhs=$4418
$4420.relation=Equal
$4420.rhs=$4410
$4410.type=Integer
$4418.type=Of
$4418.operator=$4406
$4418.arguments=$4412
$4406.type=MathString
$4406.entry=$4408
$4408.type=String
$4412.type=Vector
$4412.linkedList=$4414
$4414.type=ExpLink
$4414.next=$4416
$4414.entry=$4390
$4416.type=ExpLink
$4416.entry=$4402
$4402.type=Var
$4402.name=$4404
$4404.type=String
$4428.type=Relation
$4428.lhs=$4390
$4428.relation=In
$4428.rhs=$4426
$4426.type=Of
$4426.operator=$4398
$4426.arguments=$4422
$4398.type=MathString
$4398.entry=$4400
$4400.type=String
$4422.type=Vector
$4422.linkedList=$4424
$4424.type=ExpLink
$4424.entry=$4402
VALUE($4392) = x
VALUE($4400) = X
VALUE($4404) = k
VALUE($4408) = f
VALUE($4410) = 0
```


$4430
type
binds
scopedvar
condition
Set
$4394
$4428
$4420
type
linkedList
relation
rhs
type
type
lhs
relation
rhs
VarList
$4396
In
$4426
Relation
$4418
Equal
0
type
arguments
operator
type
type
arguments
operator
type
VarListLink
lhs
$4422
$4398
Of
$4412
$4406
Integer
entry
type
type
type
linkedList
type
entry
linkedList
Vector
entry
$4414
MathString
entry
next
f
$4390
$4424
$4416
type
name
X
entry
type
entry
type
type
$4402
ExpLink
type
x
type
type
name
type
Var
k
type
String

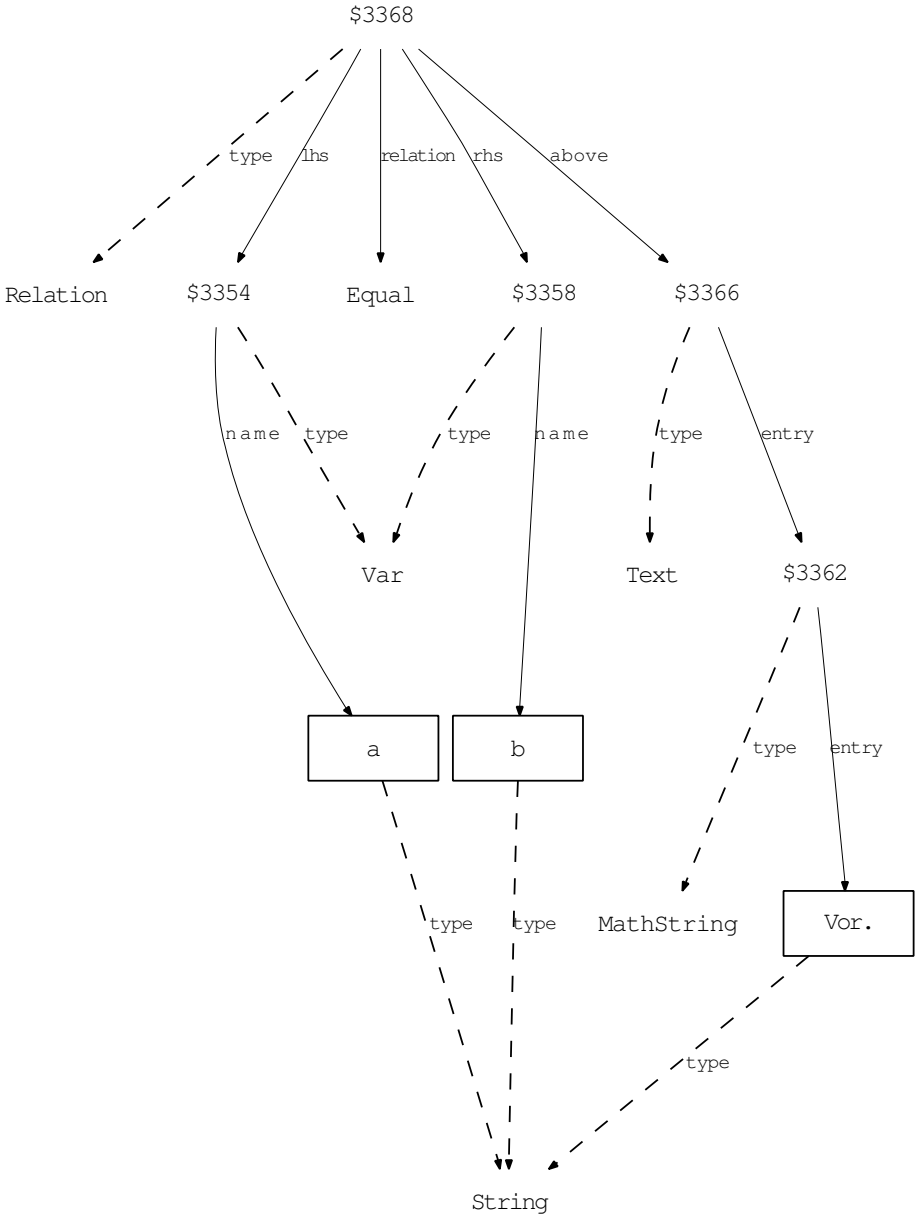
C.3 Equalities and inequalities

Example 10. Equality with auxiliary information, stored in the node `above`.

$$a \overset{\text{Vor.}}{=} b$$

```
a\stackrel{\text{Vor.}}{=}b
```

```
$3368.type=Relation
$3368.lhs=$3354
$3368.relation=Equal
$3368.rhs=$3358
$3368.above=$3366
$3354.type=Var
$3354.name=$3356
$3356.type=String
$3358.type=Var
$3358.name=$3360
$3360.type=String
$3366.type=Text
$3366.entry=$3362
$3362.type=MathString
$3362.entry=$3364
$3364.type=String
VALUE($3356) = a
VALUE($3360) = b
VALUE($3364) = Vor.
```

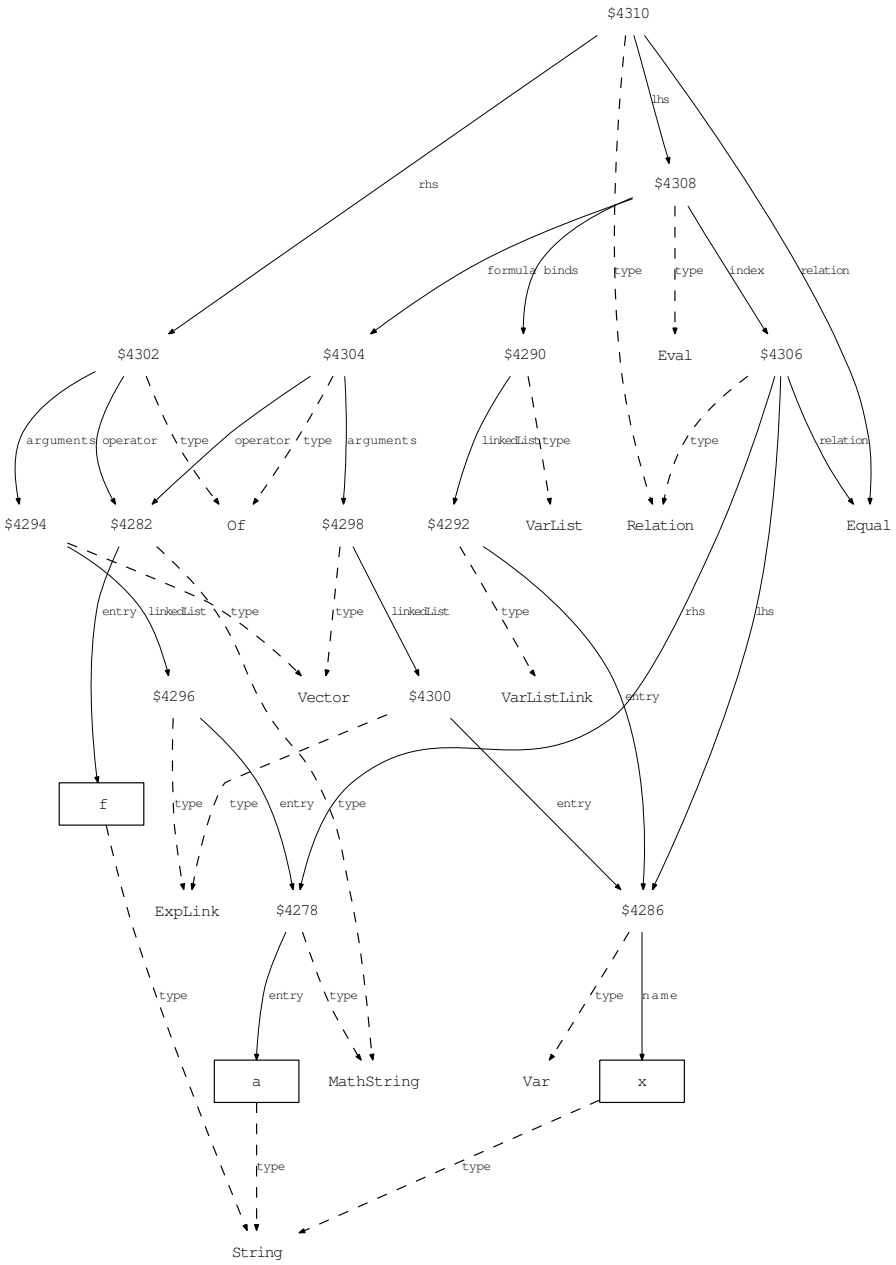

$3368
type
lhs
relation
rhs
above
Relation
$3354
Equal
$3358
$3366
name
type
type
name
type
entry
Var
Text
$3362
a
b
type
entry
type
type
MathString
Vor.
type
String

Example 11.

$$f\left(x\right)|_{x=a} = f\left(a\right)$$

```
\left.f \left(x\right) \right|_{x{{=}}a}{{=}}f \left
   (a\right)
```

```
$4310.type=Relation
$4310.lhs=$4308
$4310.relation=Equal
$4310.rhs=$4302
$4302.type=Of
$4302.operator=$4282
$4302.arguments=$4294
$4282.type=MathString
$4282.entry=$4284
$4284.type=String
$4294.type=Vector
$4294.linkedList=$4296
$4296.type=ExpLink
$4296.entry=$4278
$4278.type=MathString
$4278.entry=$4280
$4280.type=String
$4308.type=Eval
$4308.formula=$4304
$4308.binds=$4290
$4308.index=$4306
$4290.type=VarList
$4290.linkedList=$4292
$4292.type=VarListLink
$4292.entry=$4286
$4286.type=Var
$4286.name=$4288
$4288.type=String
$4304.type=Of
$4304.operator=$4282
$4304.arguments=$4298
$4298.type=Vector
$4298.linkedList=$4300
$4300.type=ExpLink
$4300.entry=$4286
$4306.type=Relation
$4306.lhs=$4286
$4306.relation=Equal
$4306.rhs=$4278
VALUE($4280) = a
VALUE($4284) = f
VALUE($4288) = x
```

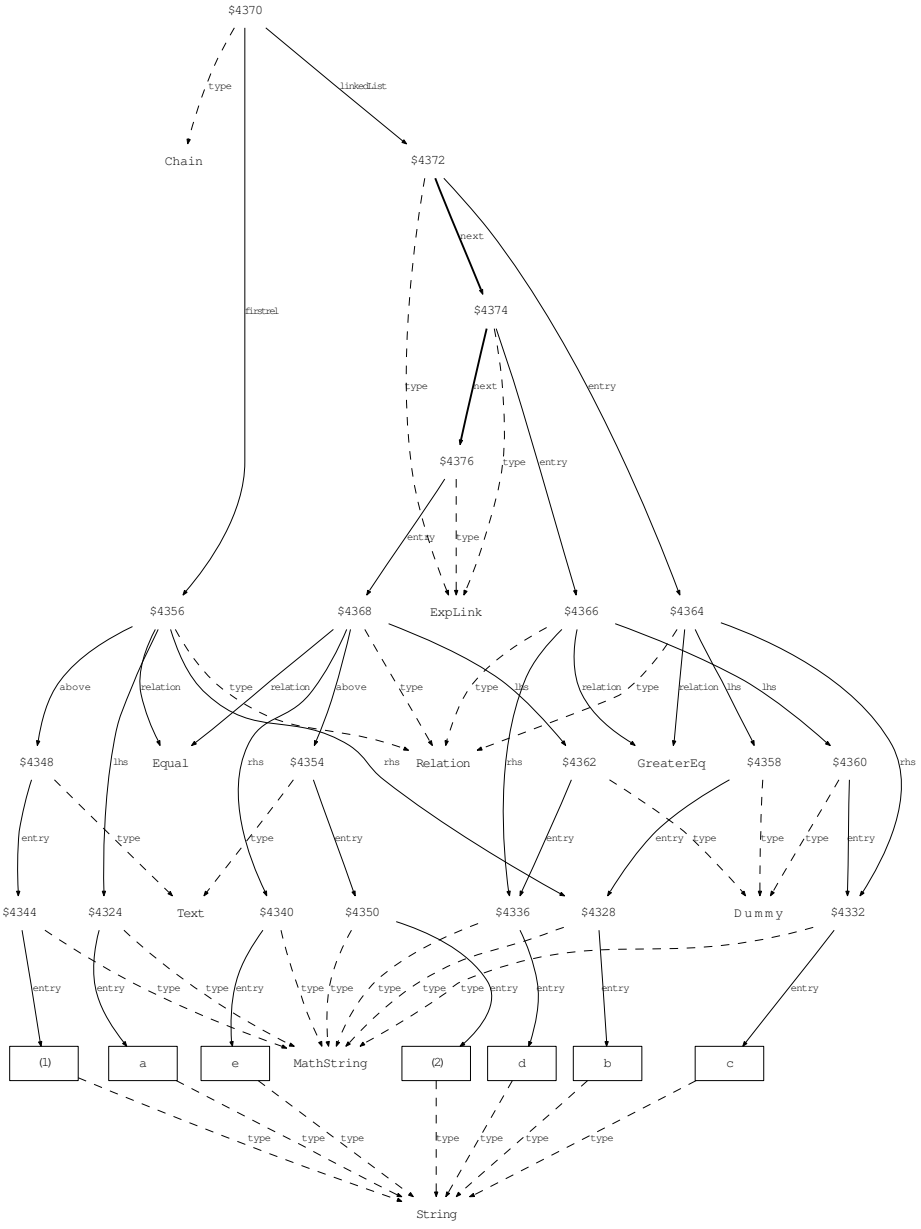

$4310
rhs
lhs
$4308
formula
binds
type
type
index
relation
$4302
$4304
$4290
Eval
$4306
arguments
operator
type
operator
type
arguments
linkedList
type
type
relation
$4294
$4282
Of
$4298
$4292
VarList
Relation
Equal
entry
linkedList
type
type
linkedList
type
rhs
lhs
$4296
Vector
$4300
VarListLink
entry
f
type
type
entry
type
entry
ExpLink
$4278
$4286
type
entry
type
type
name
a
MathString
Var
x
type
type
String

Example 12.

$$a \overset{(1)}{=} b \geq c \geq d \overset{(2)}{=} e$$

```
a\stackrel{\text{(1)}}{{=}}b{ \geq }c{ \geq }d\
   stackrel{\text{(2)}}{{=}}e
```

```
$4370.type=Chain
$4370.firstrel=$4356
$4370.linkedList=$4372
$4356.type=Relation
$4356.lhs=$4324
$4356.relation=Equal
$4356.rhs=$4328
$4356.above=$4348
$4324.type=MathString
$4324.entry=$4326
$4326.type=String
$4328.type=MathString
$4328.entry=$4330
$4330.type=String
$4348.type=Text
$4348.entry=$4344
$4344.type=MathString
$4344.entry=$4346
$4346.type=String
$4372.type=ExpLink
$4372.next=$4374
$4372.entry=$4364
$4364.type=Relation
$4364.lhs=$4358
$4364.relation=GreaterEq
$4364.rhs=$4332
$4332.type=MathString
$4332.entry=$4334
$4334.type=String
$4358.type=Dummy
$4358.entry=$4328
$4374.type=ExpLink
$4374.next=$4376
$4374.entry=$4366
$4366.type=Relation
$4366.lhs=$4360
$4366.relation=GreaterEq
$4366.rhs=$4336
$4336.type=MathString
$4336.entry=$4338
$4338.type=String
$4360.type=Dummy
$4360.entry=$4332
$4376.type=ExpLink
$4376.entry=$4368
$4368.type=Relation
$4368.lhs=$4362
$4368.relation=Equal
$4368.rhs=$4340
$4368.above=$4354
$4340.type=MathString
$4340.entry=$4342
$4342.type=String
$4354.type=Text
$4354.entry=$4350
$4350.type=MathString
$4350.entry=$4352
$4352.type=String
$4362.type=Dummy
$4362.entry=$4336
VALUE($4326) = a
VALUE($4330) = b
VALUE($4334) = c
VALUE($4338) = d
VALUE($4342) = e
VALUE($4346) = (1)
VALUE($4352) = (2)
```


$4370
type
linkedList
Chain
$4372
next
firstrel
$4374
type
next
entry
$4376
type
entry
entry
type
$4356
$4368
ExpLink
$4366
$4364
above
relation
type
relation
above
type
type
lhs
relation
type
relation
lhs
lhs
$4348
lhs
Equal
rhs
$4354
rhs
Relation
rhs
$4362
GreaterEq
$4358
$4360
rhs
entry
type
type
entry
entry
type
entry
type
type
type
entry
$4344
$4324
Text
$4340
$4350
$4336
$4328
Dummy
$4332
entry
entry
type
type
entry
type
type
type
type
type
entry
entry
entry
entry
(1)
a
e
MathString
(2)
d
b
c
type
type
type
type
type
type
type
String

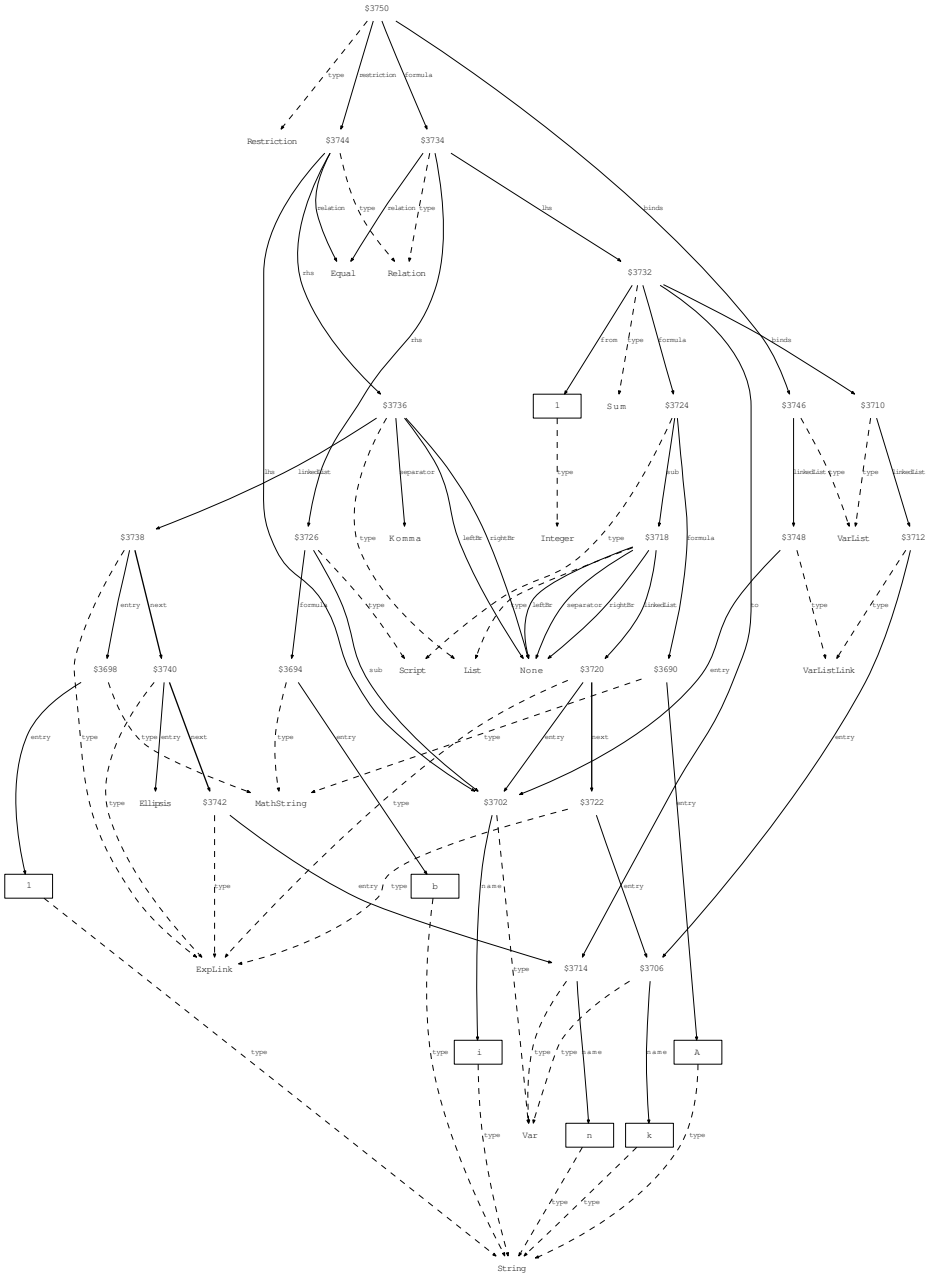
C.4 Sums and Integrals

Example 13.

$$\sum_{k=1}^{n} A_{ik}=b_i \qquad (i=1,\ldots,n)$$

```
\sum_{k = 1 }^{n} {}{A}_{ik} {{=}}{}{b}_{i} \qquad (
    i{{=}}1 , \ldots , n )
```

```
$3750.type=Restriction
$3750.formula=$3734
$3750.binds=$3746
$3750.restriction=$3744
$3734.type=Relation
$3734.lhs=$3732
$3734.relation=Equal
$3734.rhs=$3726
$3726.type=Script
$3726.formula=$3694
$3726.sub=$3702
$3694.type=MathString
$3694.entry=$3696
$3696.type=String
$3702.type=Var
$3702.name=$3704
$3704.type=String
$3732.type=Sum
$3732.formula=$3724
$3732.binds=$3710
$3732.from=$3688
$3732.to=$3714
$3688.type=Integer
$3710.type=VarList
$3710.linkedList=$3712
$3712.type=VarListLink
$3712.entry=$3706
$3706.type=Var
$3706.name=$3708
$3708.type=String
$3714.type=Var
$3714.name=$3716
$3716.type=String
$3724.type=Script
$3724.formula=$3690
$3724.sub=$3718
$3690.type=MathString
$3690.entry=$3692
$3692.type=String
$3718.type=List
$3718.leftBr=None
$3718.separator=None
$3718.rightBr=None
$3718.linkedList=$3720
$3720.type=ExpLink
$3720.next=$3722
$3720.entry=$3702
$3722.type=ExpLink
$3722.entry=$3706
$3744.type=Relation
$3744.lhs=$3702
$3744.relation=Equal
$3744.rhs=$3736
$3736.type=List
$3736.leftBr=None
$3736.separator=Komma
$3736.rightBr=None
$3736.linkedList=$3738
$3738.type=ExpLink
$3738.next=$3740
$3738.entry=$3698
$3698.type=MathString
$3698.entry=$3700
$3700.type=String
$3740.type=ExpLink
$3740.next=$3742
$3740.entry=Ellipsis
$3742.type=ExpLink
$3742.entry=$3714
$3746.type=VarList
$3746.linkedList=$3748
$3748.type=VarListLink
$3748.entry=$3702
VALUE($3688) = 1
VALUE($3692) = A
VALUE($3696) = b
VALUE($3700) = 1
VALUE($3704) = i
VALUE($3708) = k
VALUE($3716) = n
```

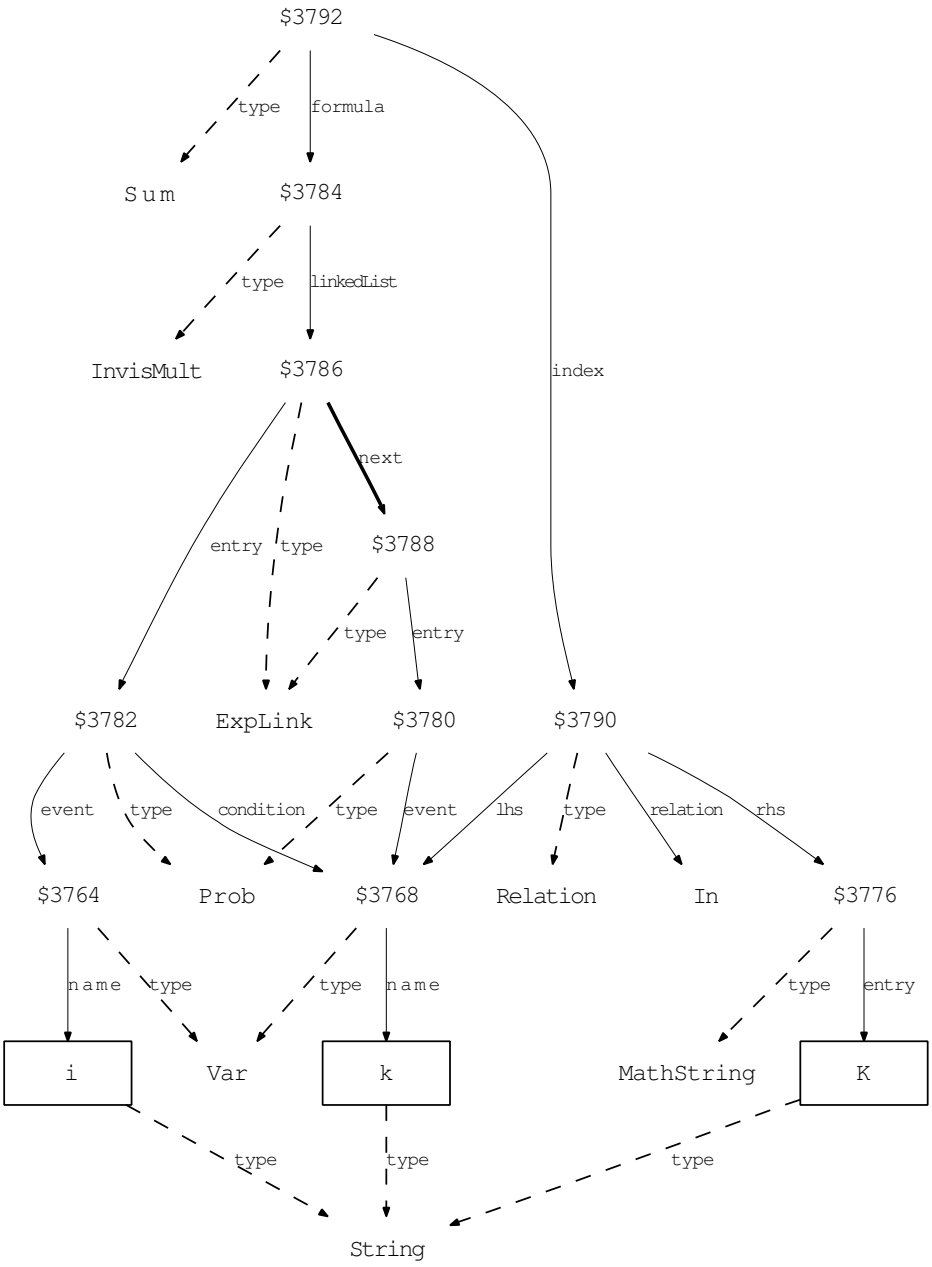

$3750
type
restriction
formula
Restriction
$3744
$3734
relation
type
relation
type
this
binds
this
Equal
Relation
$3732
this
from
type
formula
binds
$3736
1
Sum
$3724
$3746
$3710
this
linkedList
separator
type
sub
linkedList
type
type
linkedList
$3738
$3726
type
Komma
leftEx
rightEx
Integer
type
$3718
formula
$3748
VarList
$3712
entry
next
formula
type
type
leftEx
separator
rightEx
linkedList
to
type
type
$3698
$3740
$3694
sub
Script
List
None
$3720
$3690
entry
VarListLink
entry
type
type
type
entry
next
type
entry
type
entry
next
entry
type
Ellipsis
$3742
MathString
type
$3702
$3722
entry
1
type
entry
type
b
name
entry
ExpLink
type
$3714
$3706
type
i
type
type
name
name
A
type
Var
n
k
type
type
type
String

Example 14.

$$\sum_{k \in K} \Pr(i|k)\Pr(k)$$

```
\sum_{k{ \in }K} \text{Pr}(i|k) \text{Pr}(k)
```

```
$3792.type=Sum
$3792.formula=$3784
$3792.index=$3790
$3784.type=InvisMult
$3784.linkedList=$3786
$3786.type=ExpLink
$3786.next=$3788
$3786.entry=$3782
$3782.type=Prob
$3782.condition=$3768
$3782.event=$3764
$3764.type=Var
$3764.name=$3766
$3766.type=String
$3768.type=Var
$3768.name=$3770
$3770.type=String
$3788.type=ExpLink
$3788.entry=$3780
$3780.type=Prob
$3780.event=$3768
$3790.type=Relation
$3790.lhs=$3768
$3790.relation=In
$3790.rhs=$3776
$3776.type=MathString
$3776.entry=$3778
$3778.type=String
VALUE($3766) = i
VALUE($3770) = k
VALUE($3778) = K
```

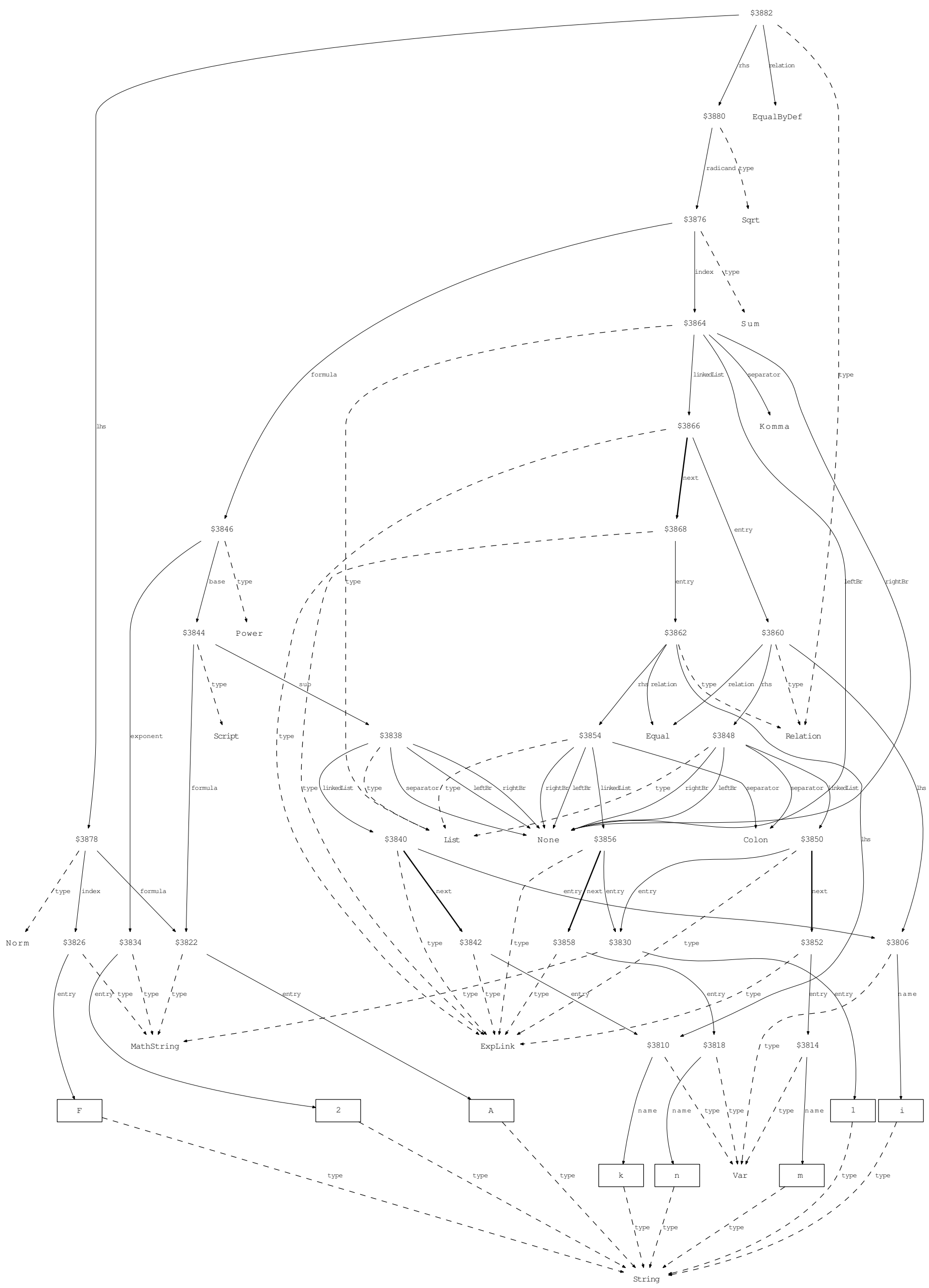

$3792
type
formula
Sum
$3784
type
linkedList
InvisMult
$3786
index
next
entry
type
$3788
type
entry
$3782
ExpLink
$3780
$3790
event
type
condition
type
event
lhs
type
relation
rhs
$3764
Prob
$3768
Relation
In
$3776
name
type
type
name
type
entry
i
Var
k
MathString
K
type
type
type
String

Example 15.

$$\|A\|_F := \sqrt{\sum_{i=1:m,k=1:n} {A_{ik}}^2}$$

```
\|A\|_{F}{ := }\sqrt{ \sum_{i{{=}}1 : m , k{{=}}1 :
   n} {{}{A}_{ik}}^{2} }
```

```
$3882.type=Relation
$3882.lhs=$3878
$3882.relation=EqualByDef
$3882.rhs=$3880
$3878.type=Norm
$3878.formula=$3822
$3878.index=$3826
$3822.type=MathString
$3822.entry=$3824
$3824.type=String
$3826.type=MathString
$3826.entry=$3828
$3828.type=String
$3880.type=Sqrt
$3880.radicand=$3876
$3876.type=Sum
$3876.formula=$3846
$3876.index=$3864
$3846.type=Power
$3846.base=$3844
$3846.exponent=$3834
$3834.type=MathString
$3834.entry=$3836
$3836.type=String
$3844.type=Script
$3844.formula=$3822
$3844.sub=$3838
$3838.type=List
$3838.leftBr=None
$3838.separator=None
$3838.rightBr=None
$3838.linkedList=$3840
$3840.type=ExpLink
$3840.next=$3842
$3840.entry=$3806
$3806.type=Var
$3806.name=$3808
$3808.type=String
$3842.type=ExpLink
$3842.entry=$3810
$3810.type=Var
$3810.name=$3812
$3812.type=String
$3864.type=List
$3864.leftBr=None
$3864.separator=Komma
$3864.rightBr=None
$3864.linkedList=$3866
$3866.type=ExpLink
$3866.next=$3868
$3866.entry=$3860
$3860.type=Relation
$3860.lhs=$3806
$3860.relation=Equal
$3860.rhs=$3848
$3848.type=List
$3848.leftBr=None
$3848.separator=Colon
$3848.rightBr=None
$3848.linkedList=$3850
$3850.type=ExpLink
$3850.next=$3852
$3850.entry=$3830
$3830.type=MathString
$3830.entry=$3832
$3832.type=String
$3852.type=ExpLink
$3852.entry=$3814
$3814.type=Var
$3814.name=$3816
$3816.type=String
$3868.type=ExpLink
$3868.entry=$3862
$3862.type=Relation
$3862.lhs=$3810
$3862.relation=Equal
$3862.rhs=$3854
$3854.type=List
$3854.leftBr=None
$3854.separator=Colon
$3854.rightBr=None
$3854.linkedList=$3856
$3856.type=ExpLink
$3856.next=$3858
$3856.entry=$3830
$3858.type=ExpLink
$3858.entry=$3818
$3818.type=Var
$3818.name=$3820
$3820.type=String
VALUE($3808) = i
VALUE($3812) = k
VALUE($3816) = m
VALUE($3820) = n
VALUE($3824) = A
VALUE($3828) = F
VALUE($3832) = 1
VALUE($3836) = 2
```

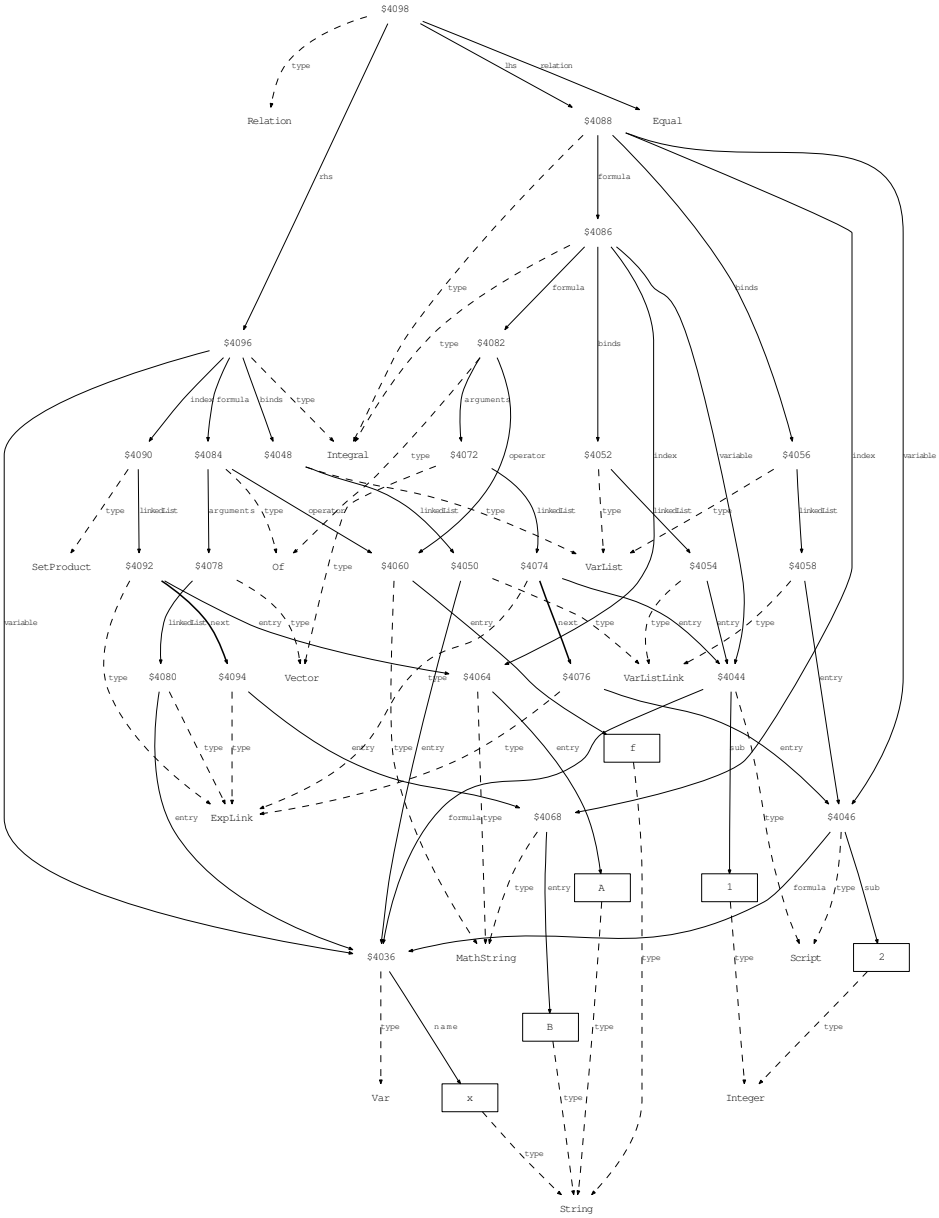

$3882
$3880
EqualByDef
$3876
Sqrt
$3864
Sum
$3866
Komma
$3846
$3868
$3844
Power
$3862
$3860
$3838
Script
$3854
Equal
$3848
Relation
$3878
$3840
List
None
$3856
Colon
$3850
Norm
$3826
$3834
$3822
$3842
$3858
$3830
$3852
$3806
MathString
ExpLink
$3810
$3818
$3814
F
2
A
1
i
k
n
Var
m
String

Example 16.

$$\int_B \int_A f\left(x_1, x_2\right)\ \mathrm{d}x_1\ \mathrm{d}x_2 = \int_{A \times B} f\left(x\right)\ \mathrm{d}x$$

```
\int_{B} \int_{A} f \left({}{x}_{ 1 } , {}{x}_{ 2 }\
   right) ~ \mathrm{d}{}{x}_{ 1 } ~ \mathrm{d}{}{x}_
   { 2 }{{=}}\int_{A \times B} f \left(x\right) ~ \
   mathrm{d}x
```

```
$4098.type=Relation
$4098.lhs=$4088
$4098.relation=Equal
$4098.rhs=$4096
$4088.type=Integral
$4088.formula=$4086
$4088.binds=$4056
$4088.variable=$4046
$4088.index=$4068
$4046.type=Script
$4046.formula=$4036
$4046.sub=$4042
$4036.type=Var
$4036.name=$4038
$4038.type=String
$4042.type=Integer
$4056.type=VarList
$4056.linkedList=$4058
$4058.type=VarListLink
$4058.entry=$4046
$4068.type=MathString
$4068.entry=$4070
$4070.type=String
$4086.type=Integral
$4086.formula=$4082
$4086.binds=$4052
$4086.variable=$4044
$4086.index=$4064
$4044.type=Script
$4044.formula=$4036
$4044.sub=$4040
$4040.type=Integer
$4052.type=VarList
$4052.linkedList=$4054
$4054.type=VarListLink
$4054.entry=$4044
$4064.type=MathString
$4064.entry=$4066
$4066.type=String
$4082.type=Of
$4082.operator=$4060
$4082.arguments=$4072
$4060.type=MathString
$4060.entry=$4062
$4062.type=String
$4072.type=Vector
$4072.linkedList=$4074
$4074.type=ExpLink
$4074.next=$4076
$4074.entry=$4044
$4076.type=ExpLink
$4076.entry=$4046
$4096.type=Integral
$4096.formula=$4084
$4096.binds=$4048
$4096.variable=$4036
$4096.index=$4090
$4048.type=VarList
$4048.linkedList=$4050
$4050.type=VarListLink
$4050.entry=$4036
$4084.type=Of
$4084.operator=$4060
$4084.arguments=$4078
$4078.type=Vector
$4078.linkedList=$4080
$4080.type=ExpLink
$4080.entry=$4036
$4090.type=SetProduct
$4090.linkedList=$4092
$4092.type=ExpLink
$4092.next=$4094
$4092.entry=$4064
$4094.type=ExpLink
$4094.entry=$4068
VALUE($4038) = x
VALUE($4040) = 1
VALUE($4042) = 2
VALUE($4062) = f
VALUE($4066) = A
VALUE($4070) = B
```

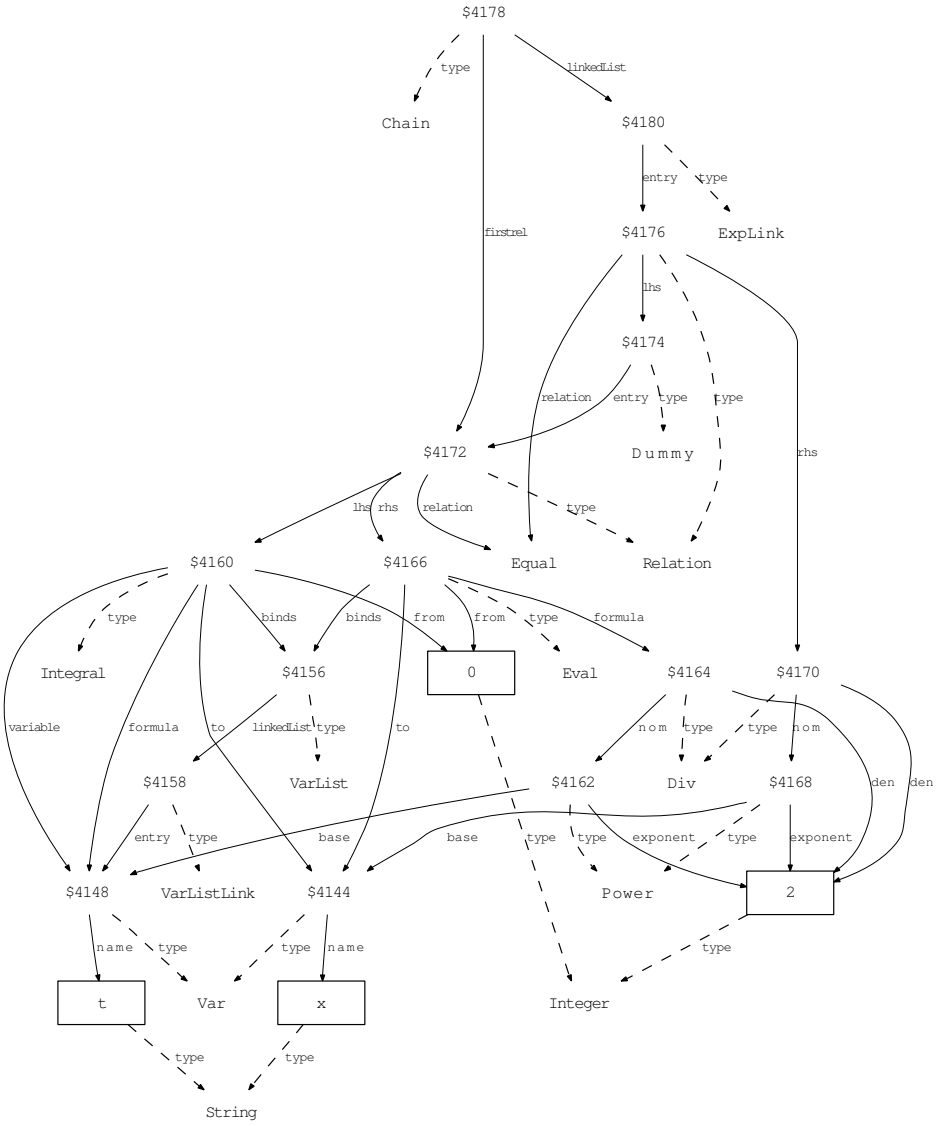

$4098
type
lhs
relation
Relation
$4088
Equal
rhs
formula
$4086
type
formula
binds
$4096
type
$4082
binds
index
formula
binds
type
arguments
$4090
$4084
$4048
Integral
type
$4072
operator
$4052
index
variable
$4056
index
variable
type
linkedList
arguments
type
operator
linkedList
type
linkedList
type
linkedList
type
linkedList
SetProduct
$4092
$4078
Of
type
$4060
$4050
$4074
VarList
$4054
$4058
variable
linkedList
next
entry
type
entry
next
type
type
entry
entry
type
type
$4080
$4094
Vector
type
$4064
$4076
VarListLink
$4044
entry
type
type
entry
type
entry
type
entry
f
sub
entry
entry
ExpLink
formula
type
$4068
type
$4046
type
entry
A
1
formula
type
sub
$4036
MathString
type
type
Script
2
type
name
B
type
type
Var
x
type
Integer
type
String

Example 17.

$$\int_0^x t \, \mathrm{d}t = \frac{t^2}{2}\Big|_0^x = \frac{x^2}{2}$$

```
\int_{ 0 }^{x} t ~ \mathrm{d}t{{=}}\left.\frac{{t}^{
    2 }}{ 2 }\right|_{ 0 }^{x}{{=}}\frac{{x}^{ 2 }}{
    2 }
```

```
$4178.type=Chain
$4178.firstrel=$4172
$4178.linkedList=$4180
$4172.type=Relation
$4172.lhs=$4160
$4172.relation=Equal
$4172.rhs=$4166
$4160.type=Integral
$4160.formula=$4148
$4160.binds=$4156
$4160.variable=$4148
$4160.from=$4152
$4160.to=$4144
$4144.type=Var
$4144.name=$4146
$4146.type=String
$4148.type=Var
$4148.name=$4150
$4150.type=String
$4152.type=Integer
$4156.type=VarList
$4156.linkedList=$4158
$4158.type=VarListLink
$4158.entry=$4148
$4166.type=Eval
$4166.formula=$4164
$4166.binds=$4156
$4166.from=$4152
$4166.to=$4144
$4164.type=Div
$4164.nom=$4162
$4164.den=$4154
$4154.type=Integer
$4162.type=Power
$4162.base=$4148
$4162.exponent=$4154
$4180.type=ExpLink
$4180.entry=$4176
$4176.type=Relation
$4176.lhs=$4174
$4176.relation=Equal
$4176.rhs=$4170
$4170.type=Div
$4170.nom=$4168
$4170.den=$4154
$4168.type=Power
$4168.base=$4144
$4168.exponent=$4154
$4174.type=Dummy
$4174.entry=$4172
VALUE($4146) = x
VALUE($4150) = t
VALUE($4152) = 0
VALUE($4154) = 2
```


$4178
type
linkedList
Chain
$4180
entry
type
firstrel
$4176
ExpLink
lhs
$4174
relation
entry
type
type
$4172
Dummy
rhs
lhs
rhs
relation
type
$4160
$4166
Equal
Relation
type
binds
binds
from
from
type
formula
Integral
$4156
0
Eval
$4164
$4170
variable
formula
to
linkedList
type
to
nom
type
type
nom
$4158
VarList
$4162
Div
$4168
den
den
entry
type
base
base
type
type
exponent
type
exponent
$4148
VarListLink
$4144
Power
2
name
type
type
name
type
t
Var
x
Integer
type
type
String

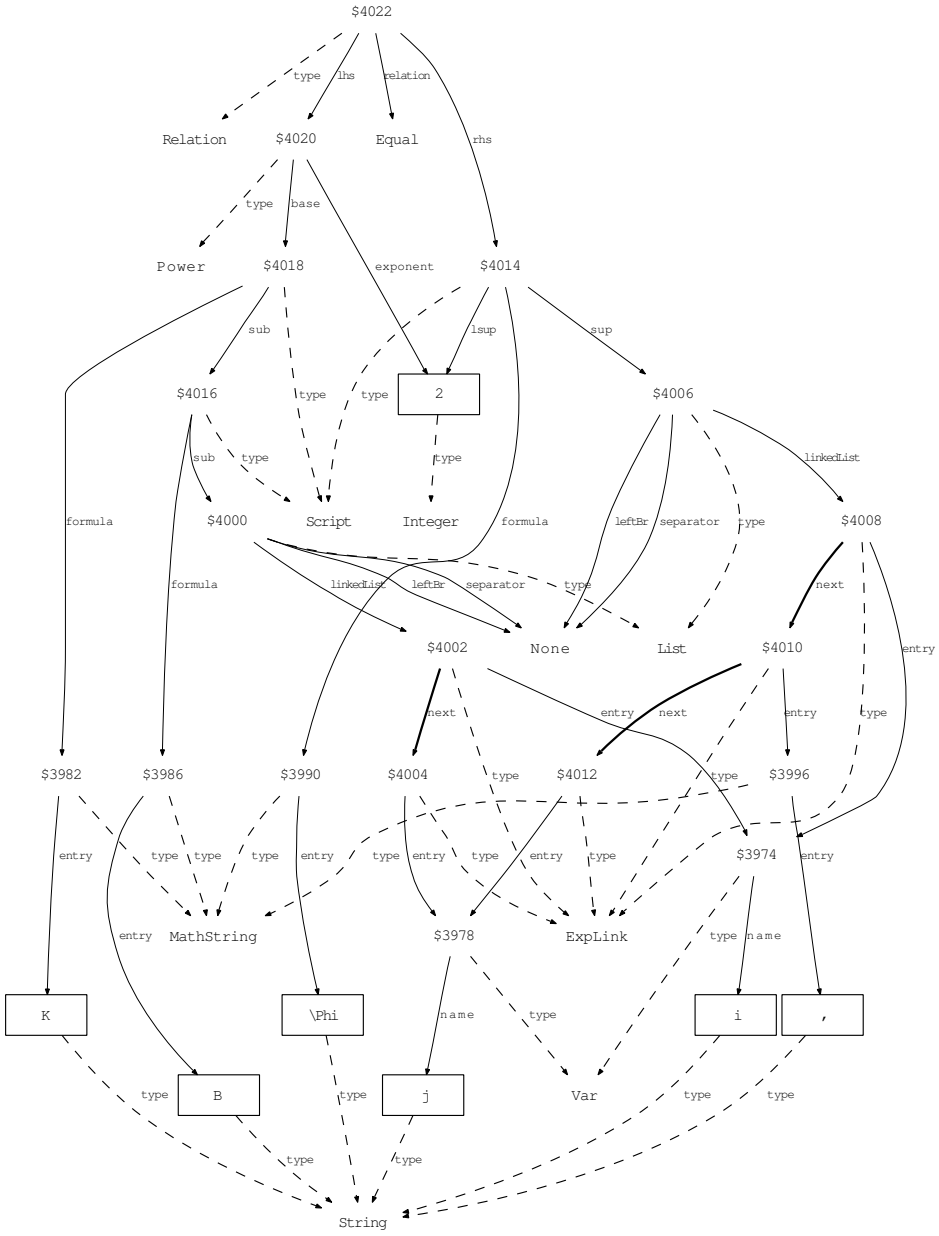
C.5 Subscripts and superscripts

Example 18.

$$K_{B_{ij}}{}^{2}{}_{=}{}^{2}\Phi^{i,j}$$

```
{{}{K}_{{}{B}_{ij}}}^{ 2 }{{=}}{}^{ 2 }{\Phi}^{i,j}
```

```
$4022.type=Relation
$4022.lhs=$4020
$4022.relation=Equal
$4022.rhs=$4014
$4014.type=Script
$4014.formula=$3990
$4014.sup=$4006
$4014.lsup=$3994
$3990.type=MathString
$3990.entry=$3992
$3992.type=String
$3994.type=Integer
$4006.type=List
$4006.leftBr=None
$4006.separator=None
$4006.linkedList=$4008
$4008.type=ExpLink
$4008.next=$4010
$4008.entry=$3974
$3974.type=Var
$3974.name=$3976
$3976.type=String
$4010.type=ExpLink
$4010.next=$4012
$4010.entry=$3996
$3996.type=MathString
$3996.entry=$3998
$3998.type=String
$4012.type=ExpLink
$4012.entry=$3978
$3978.type=Var
$3978.name=$3980
$3980.type=String
$4020.type=Power
$4020.base=$4018
$4020.exponent=$3994
$4018.type=Script
$4018.formula=$3982
$4018.sub=$4016
$3982.type=MathString
$3982.entry=$3984
$3984.type=String
$4016.type=Script
$4016.formula=$3986
$4016.sub=$4000
$3986.type=MathString
$3986.entry=$3988
$3988.type=String
$4000.type=List
$4000.leftBr=None
$4000.separator=None
$4000.linkedList=$4002
$4002.type=ExpLink
$4002.next=$4004
$4002.entry=$3974
$4004.type=ExpLink
$4004.entry=$3978
VALUE($3976) = i
VALUE($3980) = j
VALUE($3984) = K
VALUE($3988) = B
VALUE($3992) = \Phi
VALUE($3994) = 2
VALUE($3998) = ,
```

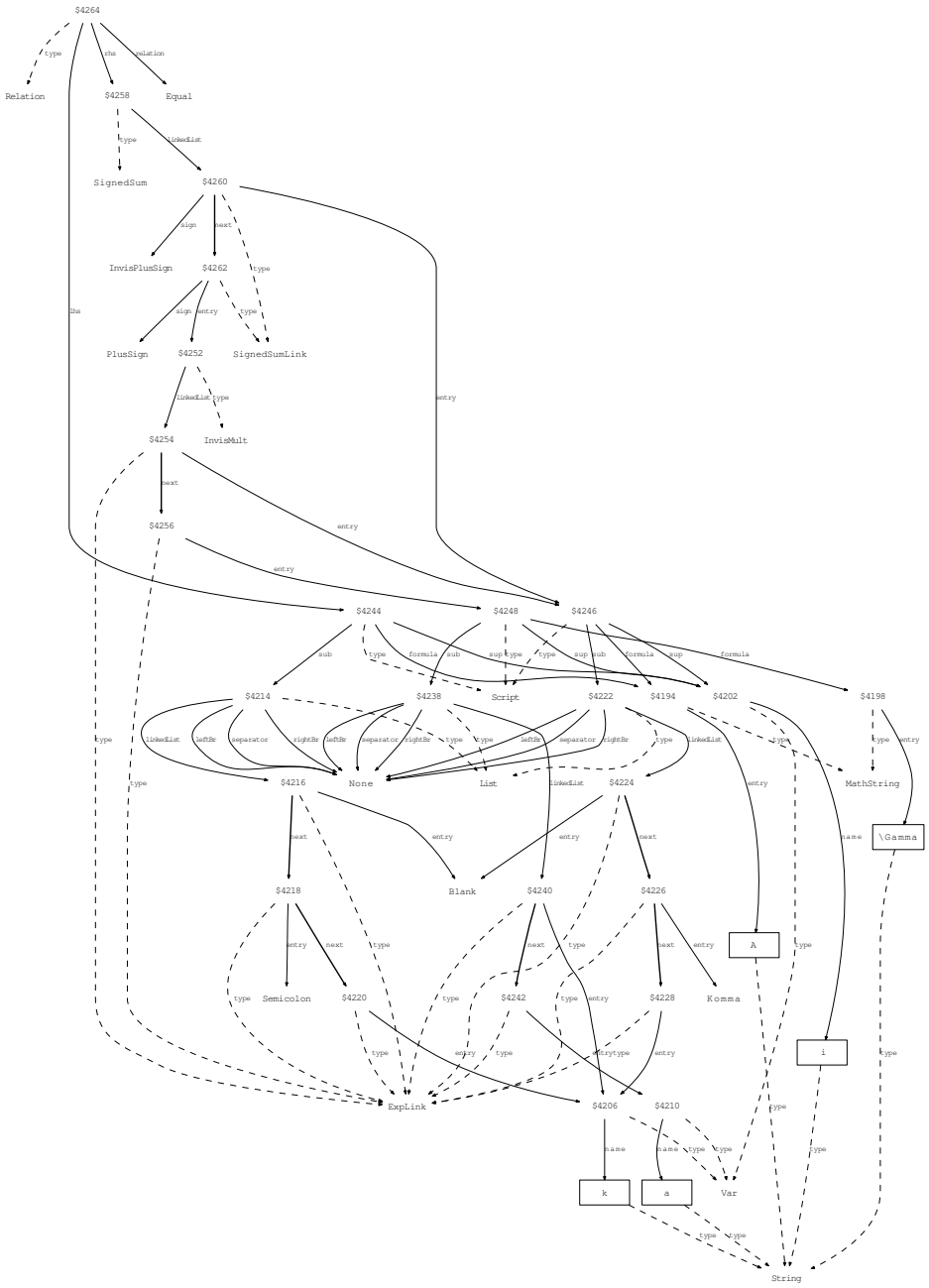

$4022
type
lhs
relation
Relation
$4020
Equal
rhs
type
base
Power
$4018
exponent
$4014
sub
lsup
sup
$4016
type
type
2
$4006
sub
type
type
linkedList
formula
$4000
Script
Integer
formula
leftBr
separator
type
$4008
formula
linkedList
leftBr
separator
type
next
$4002
None
List
$4010
entry
next
entry
type
$3982
$3986
$3990
$4004
type
$4012
type
$3996
entry
type
type
type
entry
type
entry
type
entry
type
$3974
entry
entry
MathString
$3978
ExpLink
type
name
K
\Phi
name
type
i
,
type
B
type
j
Var
type
type
type
type
String

Example 19.

$$A^{i}{}_{;k}=A^{i}{}_{,k}+A^{i}{}_{,k}\Gamma^{i}_{ka}$$

```
{}{A}_{ ~ ; k}^{i}{{=}} {}{A}_{ ~ , k}^{i} + {}{A}_{
    ~ , k}^{i} {}{\Gamma}_{ka}^{i}
```

```
$4264.type=Relation
$4264.lhs=$4244
$4264.relation=Equal
$4264.rhs=$4258
$4244.type=Script
$4244.formula=$4194
$4244.sub=$4214
$4244.sup=$4202
$4194.type=MathString
$4194.entry=$4196
$4196.type=String
$4202.type=Var
$4202.name=$4204
$4204.type=String
$4214.type=List
$4214.leftBr=None
$4214.separator=None
$4214.rightBr=None
$4214.linkedList=$4216
$4216.type=ExpLink
$4216.next=$4218
$4216.entry=Blank
$4218.type=ExpLink
$4218.next=$4220
$4218.entry=Semicolon
$4220.type=ExpLink
$4220.entry=$4206
$4206.type=Var
$4206.name=$4208
$4208.type=String
$4258.type=SignedSum
$4258.linkedList=$4260
$4260.type=SignedSumLink
$4260.next=$4262
$4260.sign=InvisPlusSign
$4260.entry=$4246
$4246.type=Script
$4246.formula=$4194
$4246.sub=$4222
$4246.sup=$4202
$4222.type=List
$4222.leftBr=None
$4222.separator=None
$4222.rightBr=None
$4222.linkedList=$4224
$4224.type=ExpLink
$4224.next=$4226
$4224.entry=Blank
$4226.type=ExpLink
$4226.next=$4228
$4226.entry=Komma
$4228.type=ExpLink
$4228.entry=$4206
$4262.type=SignedSumLink
$4262.sign=PlusSign
$4262.entry=$4252
$4252.type=InvisMult
$4252.linkedList=$4254
$4254.type=ExpLink
$4254.next=$4256
$4254.entry=$4246
$4256.type=ExpLink
$4256.entry=$4248
$4248.type=Script
$4248.formula=$4198
$4248.sub=$4238
$4248.sup=$4202
$4198.type=MathString
$4198.entry=$4200
$4200.type=String
$4238.type=List
$4238.leftBr=None
$4238.separator=None
$4238.rightBr=None
$4238.linkedList=$4240
$4240.type=ExpLink
$4240.next=$4242
$4240.entry=$4206
$4242.type=ExpLink
$4242.entry=$4210
$4210.type=Var
$4210.name=$4212
$4212.type=String
VALUE($4196) = A
VALUE($4200) = \Gamma
VALUE($4204) = i
VALUE($4208) = k
VALUE($4212) = a
```


$4264
type
lhs
relation
Relation
$4258
Equal
type
linkedList
SignedSum
$4260
sign
next
InvisPlusSign
$4262
type
lhs
sign
entry
type
PlusSign
$4252
SignedSumLink
linkedList
type
$4254
InvisMult
next
$4256
entry
entry
$4244
$4248
$4246
sub
type
formula
sub
sup
type
type
sup
sub
formula
sup
formula
$4214
$4238
Script
$4222
$4194
$4202
$4198
type
linkedList
leftBr
separator
rightBr
leftBr
separator
rightBr
type
type
leftBr
separator
rightBr
type
linkedList
type
type
entry
type
$4216
None
List
linkedList
$4224
entry
MathString
type
name
\Gamma
next
entry
entry
next
$4218
Blank
$4240
$4226
entry
next
type
next
type
next
entry
A
type
type
Semicolon
$4220
type
$4242
type
entry
$4228
Komma
type
entry
type
entry
type
entry
type
i
type
$4206
ExpLink
$4210
type
name
name
type
type
k
a
Var
type
type
type
String

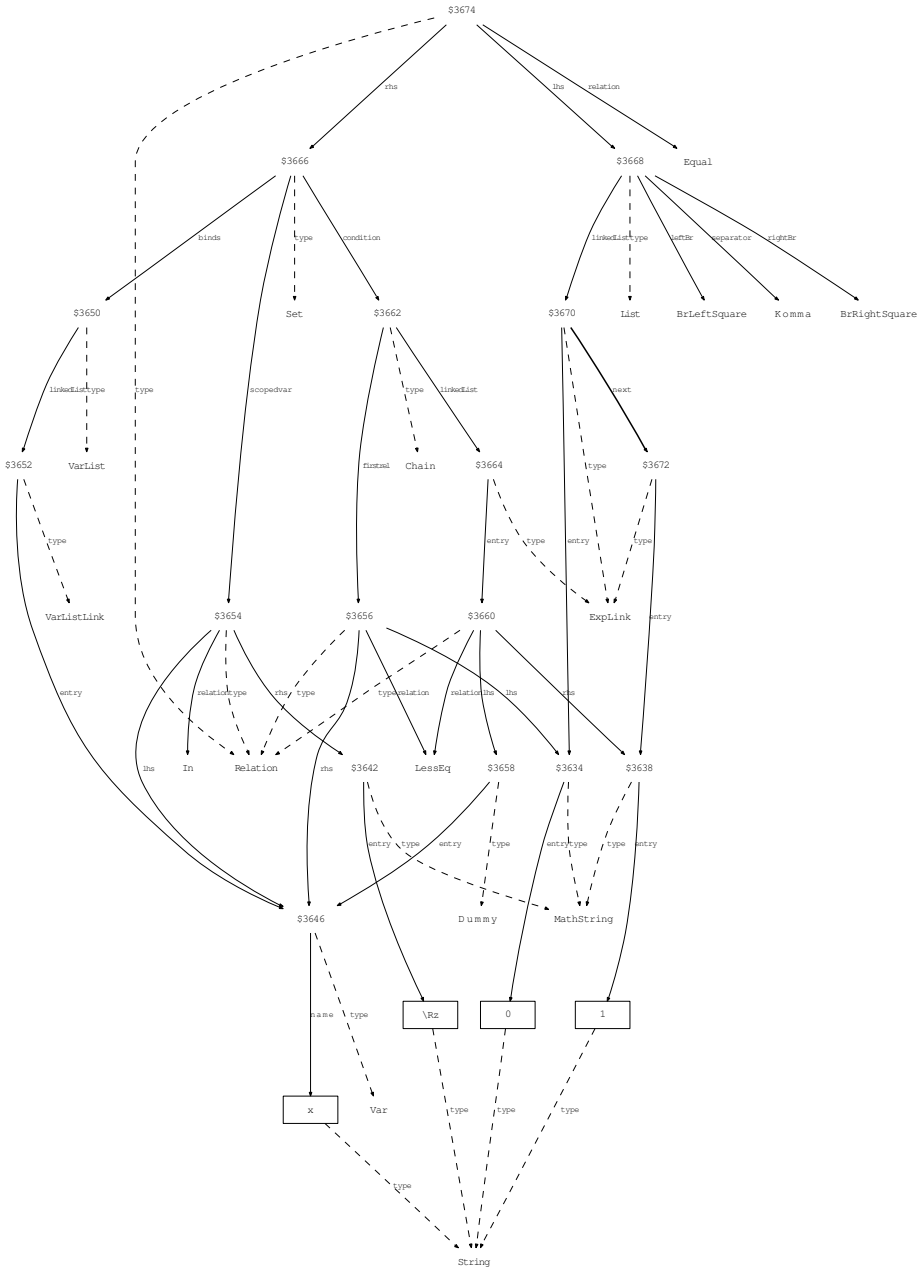
C.6 Intervals

Example 20. The interval on the LHS is a list with layout options for the parentheses.

$$[0,1]=\{x\in\mathbb{R} \mid 0\leq x\leq 1\}$$

```
[0 , 1]{{=}} \left\{x{ \in }\Rz \mid 0{ \leq }x{ \
   leq }1\right\}
```

```
$3674.type=Relation
$3674.lhs=$3668
$3674.relation=Equal
$3674.rhs=$3666
$3666.type=Set
$3666.scopedvar=$3654
$3666.binds=$3650
$3666.condition=$3662
$3650.type=VarList
$3650.linkedList=$3652
$3652.type=VarListLink
$3652.entry=$3646
$3646.type=Var
$3646.name=$3648
$3648.type=String
$3654.type=Relation
$3654.lhs=$3646
$3654.relation=In
$3654.rhs=$3642
$3642.type=MathString
$3642.entry=$3644
$3644.type=String
$3662.type=Chain
$3662.firstrel=$3656
$3662.linkedList=$3664
$3656.type=Relation
$3656.lhs=$3634
$3656.relation=LessEq
$3656.rhs=$3646
$3634.type=MathString
$3634.entry=$3636
$3636.type=String
$3664.type=ExpLink
$3664.entry=$3660
$3660.type=Relation
$3660.lhs=$3658
$3660.relation=LessEq
$3660.rhs=$3638
$3638.type=MathString
$3638.entry=$3640
$3640.type=String
$3658.type=Dummy
$3658.entry=$3646
$3668.type=List
$3668.leftBr=BrLeftSquare
$3668.separator=Komma
$3668.rightBr=BrRightSquare
$3668.linkedList=$3670
$3670.type=ExpLink
$3670.next=$3672
$3670.entry=$3634
$3672.type=ExpLink
$3672.entry=$3638
VALUE($3636) = 0
VALUE($3640) = 1
VALUE($3644) = \Rz
VALUE($3648) = x
```

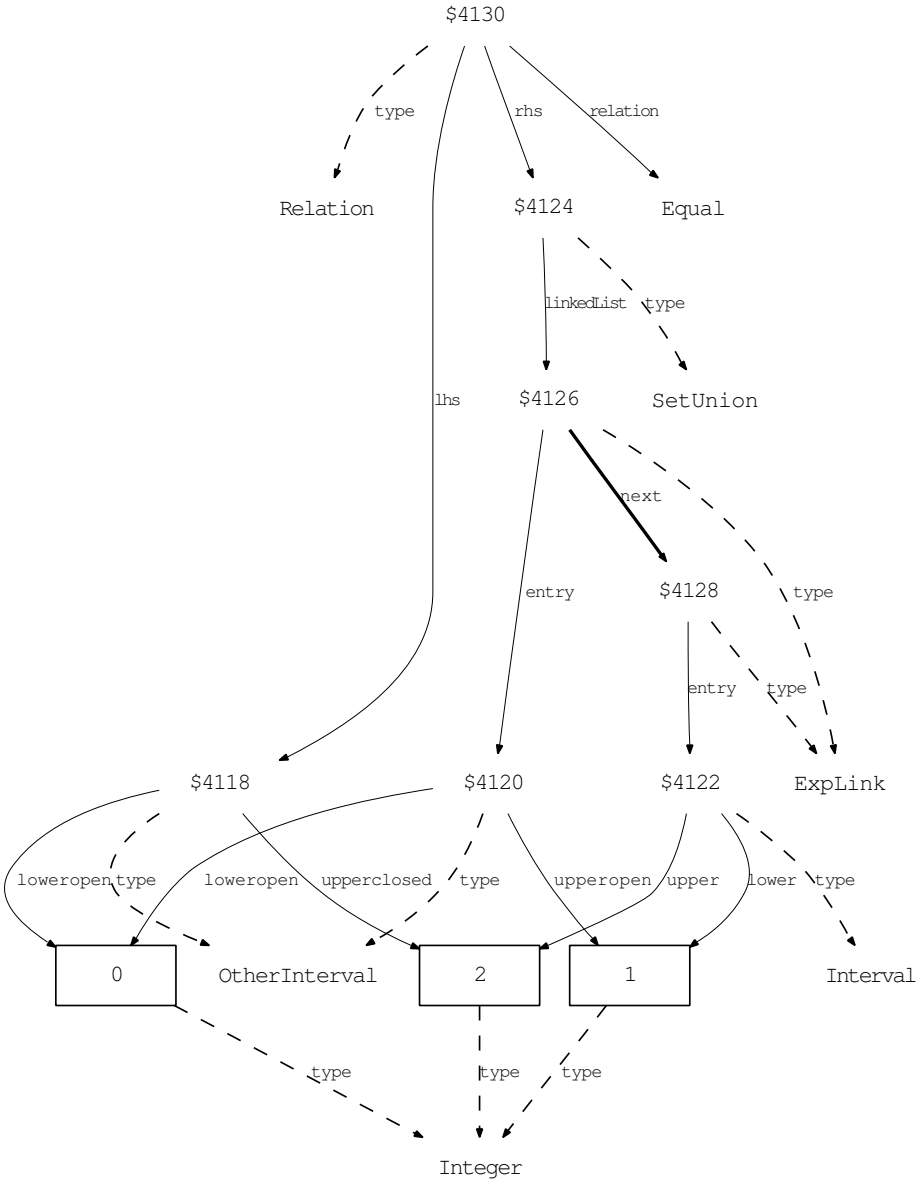

$3674
rhs
lhs
relation
$3666
$3668
Equal
binds
type
condition
linkedList
type
leftBr
separator
rightBr
$3650
Set
$3662
$3670
List
BrLeftSquare
Komma
BrRightSquare
linkedList
type
type
scopedvar
type
linkedList
next
$3652
VarList
firstel
Chain
$3664
type
$3672
type
entry
type
entry
type
type
VarListLink
$3654
$3656
$3660
ExpLink
entry
entry
relation
type
rhs
type
type
relation
relation
lhs
lhs
In
Relation
lhs
$3642
LessEq
$3658
$3634
$3638
entry
type
entry
type
entry
type
type
entry
$3646
Dummy
MathString
name
type
\Rz
0
1
x
Var
type
type
type
type
String

Example 21.

$$(0,2]{=}(0,1) \cup [1,2]$$

```
( 0 , 2 ]{{=}}( 0 , 1 ) \cup [ 1 , 2 ]
```

```
$4130.type=Relation
$4130.lhs=$4118
$4130.relation=Equal
$4130.rhs=$4124
$4118.type=OtherInterval
$4118.loweropen=$4112
$4118.upperclosed=$4116
$4112.type=Integer
$4116.type=Integer
$4124.type=SetUnion
$4124.linkedList=$4126
$4126.type=ExpLink
$4126.next=$4128
$4126.entry=$4120
$4120.type=OtherInterval
$4120.loweropen=$4112
$4120.upperopen=$4114
$4114.type=Integer
$4128.type=ExpLink
$4128.entry=$4122
$4122.type=Interval
$4122.lower=$4114
$4122.upper=$4116
VALUE($4112) = 0
VALUE($4114) = 1
VALUE($4116) = 2
```


$4130
type
rhs
relation
Relation
$4124
Equal
linkedList
type
lhs
$4126
SetUnion
next
entry
$4128
type
entry
type
$4118
$4120
$4122
ExpLink
loweropen
type
loweropen
upperclosed
type
upperopen
upper
lower
type
0
OtherInterval
2
1
Interval
type
type
type
Integer

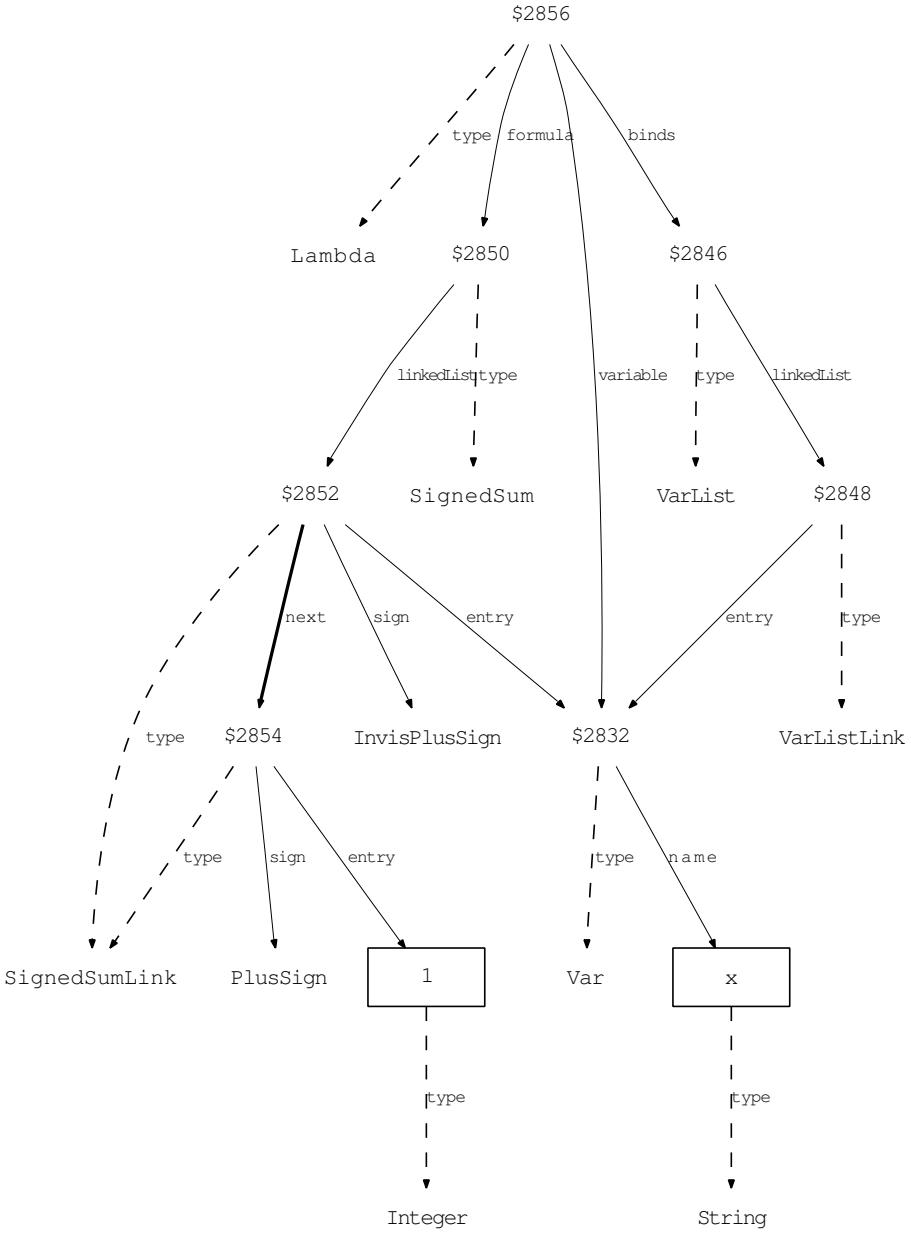
C.7 Quantification and lambda calculus

Example 22.

$$\lambda x.x + 1$$

```
\lambda x . x + 1
```

```
$2856.type=Lambda
$2856.formula=$2850
$2856.binds=$2846
$2856.variable=$2832
$2832.type=Var
$2832.name=$2834
$2834.type=String
$2846.type=VarList
$2846.linkedList=$2848
$2848.type=VarListLink
$2848.entry=$2832
$2850.type=SignedSum
$2850.linkedList=$2852
$2852.type=SignedSumLink
$2852.next=$2854
$2852.sign=InvisPlusSign
$2852.entry=$2832
$2854.type=SignedSumLink
$2854.sign=PlusSign
$2854.entry=$2844
$2844.type=Integer
VALUE($2834) = x
VALUE($2844) = 1
```

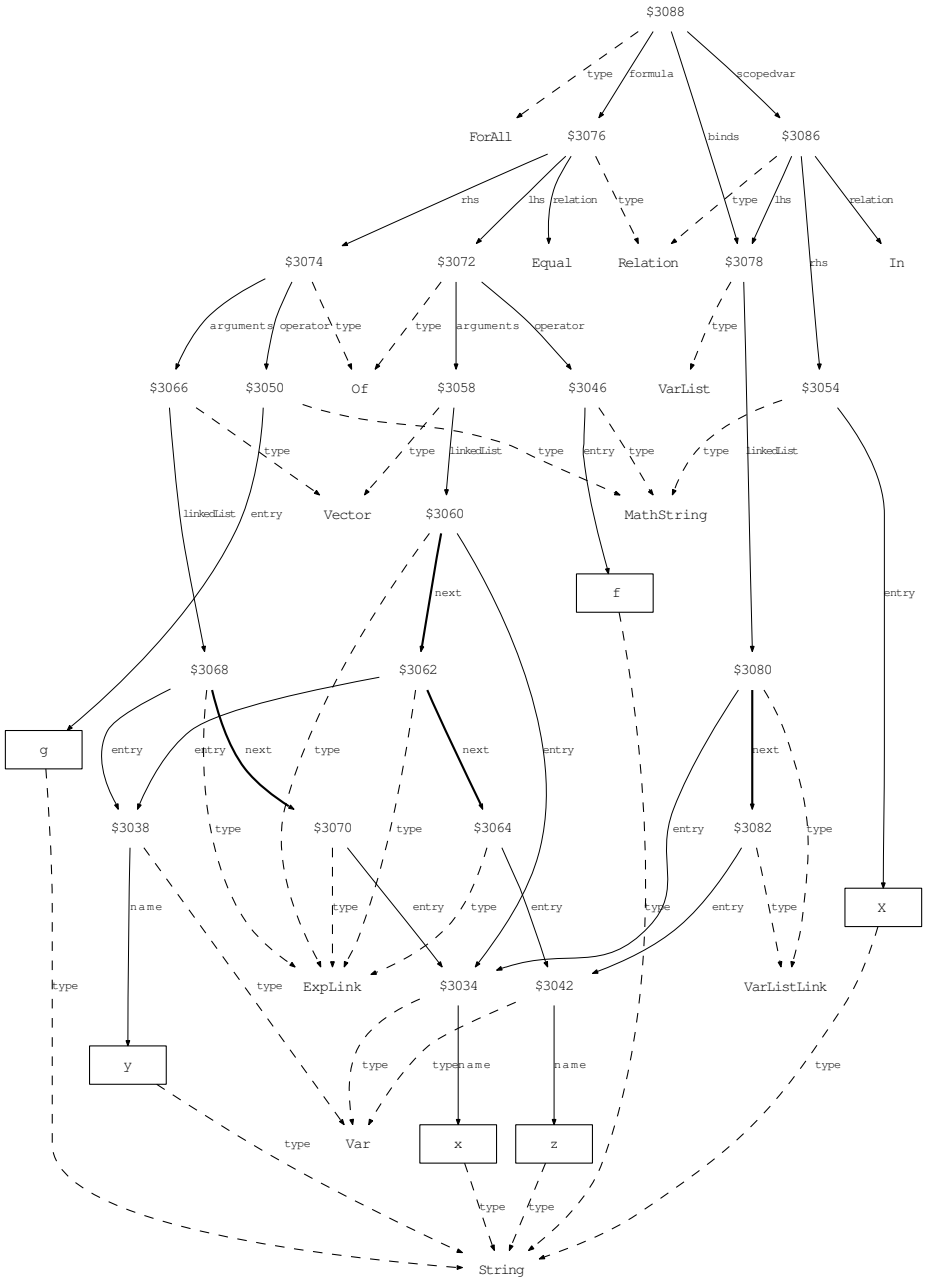

$2856
type
formula
binds
Lambda
$2850
$2846
linkedList
type
variable
type
linkedList
$2852
SignedSum
VarList
$2848
next
sign
entry
entry
type
type
$2854
InvisPlusSign
$2832
VarListLink
type
sign
entry
type
name
SignedSumLink
PlusSign
1
Var
x
type
type
Integer
String

Example 23.

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

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

```
$3088.type=ForAll
$3088.formula=$3076
$3088.scopedvar=$3086
$3088.binds=$3078
$3076.type=Relation
$3076.lhs=$3072
$3076.relation=Equal
$3076.rhs=$3074
$3072.type=Of
$3072.operator=$3046
$3072.arguments=$3058
$3046.type=MathString
$3046.entry=$3048
$3048.type=String
$3058.type=Vector
$3058.linkedList=$3060
$3060.type=ExpLink
$3060.next=$3062
$3060.entry=$3034
$3034.type=Var
$3034.name=$3036
$3036.type=String
$3062.type=ExpLink
$3062.next=$3064
$3062.entry=$3038
$3038.type=Var
$3038.name=$3040
$3040.type=String
$3064.type=ExpLink
$3064.entry=$3042
$3042.type=Var
$3042.name=$3044
$3044.type=String
$3074.type=Of
$3074.operator=$3050
$3074.arguments=$3066
$3050.type=MathString
$3050.entry=$3052
$3052.type=String
$3066.type=Vector
$3066.linkedList=$3068
$3068.type=ExpLink
$3068.next=$3070
$3068.entry=$3038
$3070.type=ExpLink
$3070.entry=$3034
$3078.type=VarList
$3078.linkedList=$3080
$3080.type=VarListLink
$3080.next=$3082
$3080.entry=$3034
$3082.type=VarListLink
$3082.entry=$3042
$3086.type=Relation
$3086.lhs=$3078
$3086.relation=In
$3086.rhs=$3054
$3054.type=MathString
$3054.entry=$3056
$3056.type=String
VALUE($3036) = x
VALUE($3040) = y
VALUE($3044) = z
VALUE($3048) = f
VALUE($3052) = g
VALUE($3056) = X
```

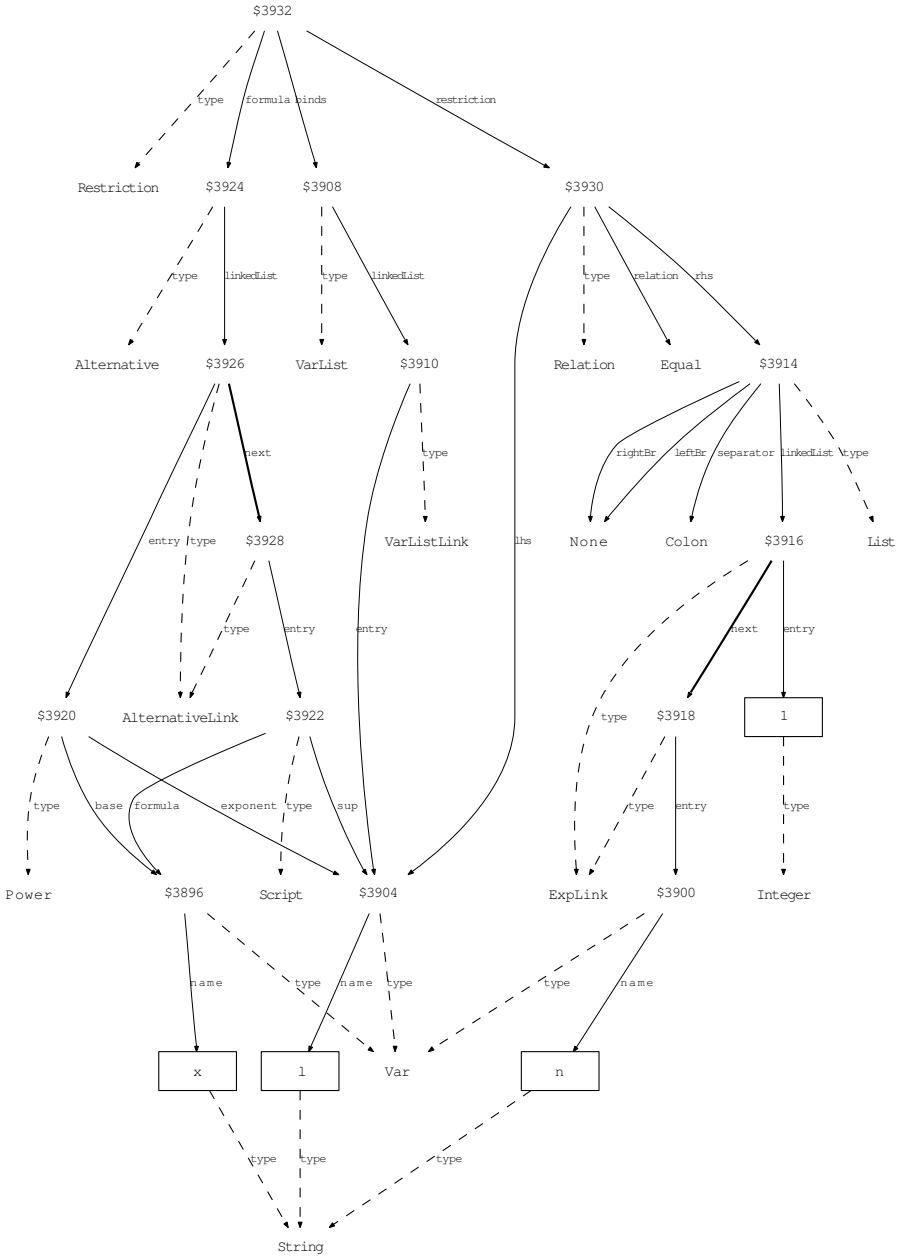

$3088
type
formula
scopedvar
ForAll
$3076
binds
$3086
rhs
lhs
relation
type
type
lhs
relation
$3074
$3072
Equal
Relation
$3078
rhs
In
arguments
operator
type
type
arguments
operator
type
$3066
$3050
Of
$3058
$3046
VarList
$3054
type
type
linkedList
type
entry
type
type
linkedList
linkedList
entry
Vector
$3060
MathString
next
f
entry
$3068
$3062
$3080
g
entry
entry
next
type
next
entry
next
$3038
type
$3070
type
$3064
entry
$3082
type
name
type
entry
type
entry
type
entry
type
X
type
type
ExpLink
$3034
$3042
VarListLink
y
type
type
name
name
type
type
Var
x
z
type
type
String

Example 24.

$$x^{l} \qquad (l{=}1 : n)$$

```
{x}^{l} \qquad ( l{{=}} 1 : n )
```

```
$3932.type=Restriction
$3932.formula=$3924
$3932.binds=$3908
$3932.restriction=$3930
$3908.type=VarList
$3908.linkedList=$3910
$3910.type=VarListLink
$3910.entry=$3904
$3904.type=Var
$3904.name=$3906
$3906.type=String
$3924.type=Alternative
$3924.linkedList=$3926
$3926.type=AlternativeLink
$3926.next=$3928
$3926.entry=$3920
$3920.type=Power
$3920.base=$3896
$3920.exponent=$3904
$3896.type=Var
$3896.name=$3898
$3898.type=String
$3928.type=AlternativeLink
$3928.entry=$3922
$3922.type=Script
$3922.formula=$3896
$3922.sup=$3904
$3930.type=Relation
$3930.lhs=$3904
$3930.relation=Equal
$3930.rhs=$3914
$3914.type=List
$3914.leftBr=None
$3914.separator=Colon
$3914.rightBr=None
$3914.linkedList=$3916
$3916.type=ExpLink
$3916.next=$3918
$3916.entry=$3912
$3912.type=Integer
$3918.type=ExpLink
$3918.entry=$3900
$3900.type=Var
$3900.name=$3902
$3902.type=String
VALUE($3898) = x
VALUE($3902) = n
VALUE($3906) = l
VALUE($3912) = 1
```

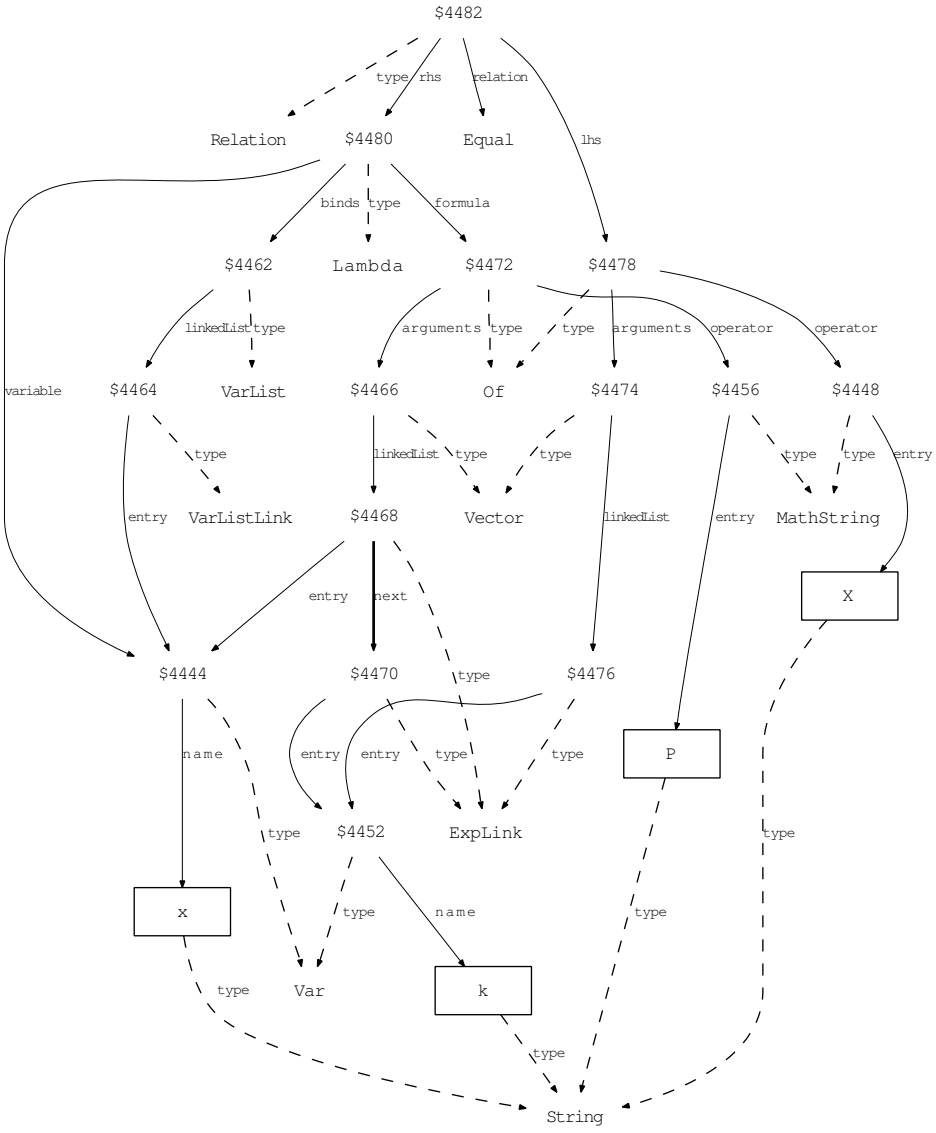

$3932
type
formula
binds
restriction
Restriction
$3924
$3908
$3930
type
linkedList
type
linkedList
type
relation
rhs
Alternative
$3926
VarList
$3910
Relation
Equal
$3914
next
type
rightBr
leftBr
separator
linkedList
type
entry
type
$3928
VarListLink
lhs
None
Colon
$3916
List
type
entry
entry
next
entry
$3920
AlternativeLink
$3922
type
$3918
1
type
base
formula
exponent
type
sup
type
entry
type
Power
$3896
Script
$3904
ExpLink
$3900
Integer
name
type
name
type
type
name
x
1
Var
n
type
type
type
String

Example 25.

$$X\left(k\right) {=} \lambda x . P\left(x , k\right)$$

```
X \left(k\right) {{=}} \lambda x . P \left(x , k\
   right)
```

```
$4482.type=Relation
$4482.lhs=$4478
$4482.relation=Equal
$4482.rhs=$4480
$4478.type=Of
$4478.operator=$4448
$4478.arguments=$4474
$4448.type=MathString
$4448.entry=$4450
$4450.type=String
$4474.type=Vector
$4474.linkedList=$4476
$4476.type=ExpLink
$4476.entry=$4452
$4452.type=Var
$4452.name=$4454
$4454.type=String
$4480.type=Lambda
$4480.formula=$4472
$4480.binds=$4462
$4480.variable=$4444
$4444.type=Var
$4444.name=$4446
$4446.type=String
$4462.type=VarList
$4462.linkedList=$4464
$4464.type=VarListLink
$4464.entry=$4444
$4472.type=Of
$4472.operator=$4456
$4472.arguments=$4466
$4456.type=MathString
$4456.entry=$4458
$4458.type=String
$4466.type=Vector
$4466.linkedList=$4468
$4468.type=ExpLink
$4468.next=$4470
$4468.entry=$4444
$4470.type=ExpLink
$4470.entry=$4452
VALUE($4446) = x
VALUE($4450) = X
VALUE($4454) = k
VALUE($4458) = P
```


$4482
type
rhs
relation
Relation
$4480
Equal
lhs
binds
type
formula
$4462
Lambda
$4472
$4478
linkedList
type
arguments
type
type
arguments
operator
operator
variable
$4464
VarList
$4466
Of
$4474
$4456
$4448
type
linkedList
type
type
type
type
entry
entry
VarListLink
$4468
Vector
linkedList
entry
MathString
X
entry
next
$4444
$4470
type
$4476
name
entry
entry
type
type
P
type
$4452
ExpLink
type
x
type
name
type
type
Var
k
type
String

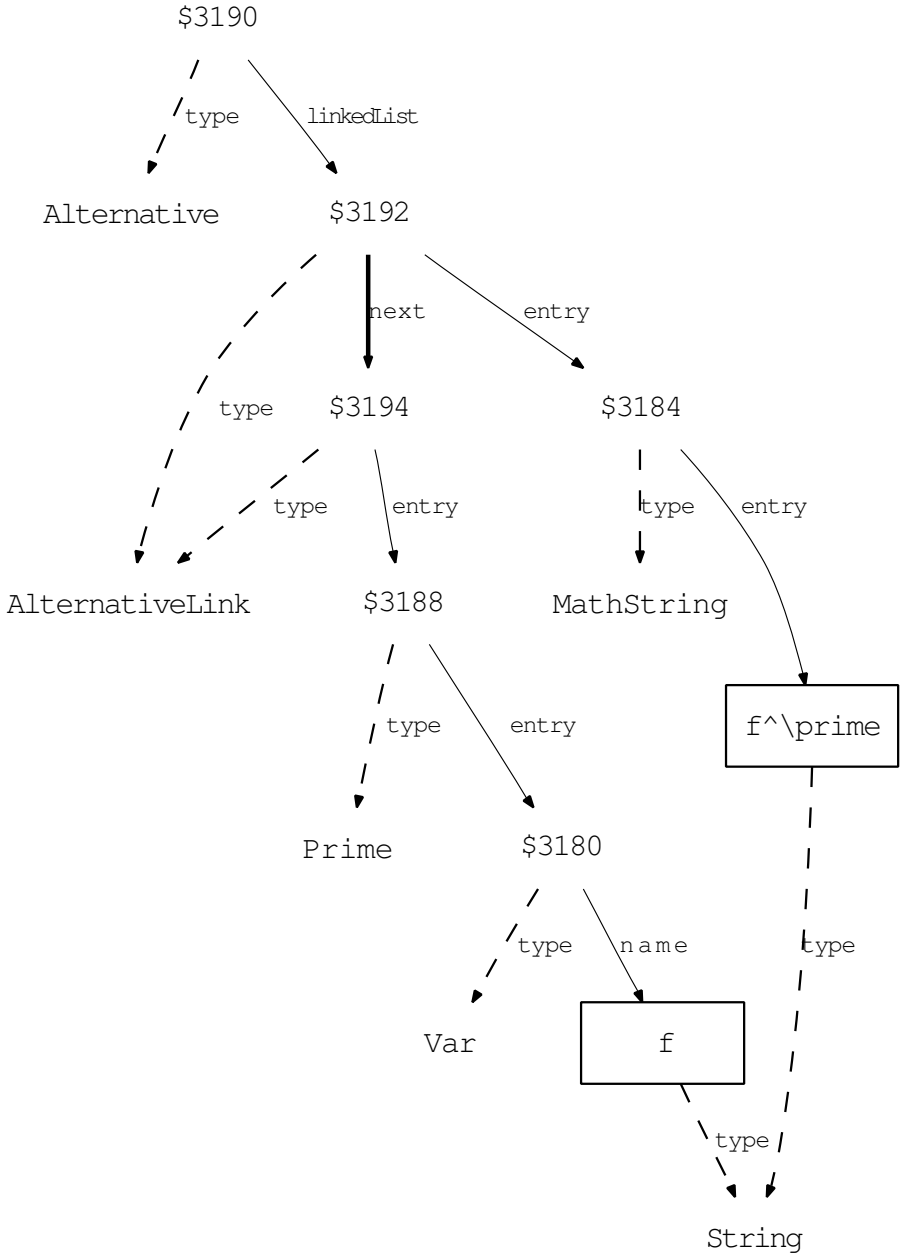
C.8 Ambiguous expressions

Example 26. f' is ambiguous: It could be the application of the operation ′ to the mapping f or the name of fuzzy numbers f, f', f''.

$$f'$$

```
f^\prime
```

```
$3190.type=Alternative
$3190.linkedList=$3192
$3192.type=AlternativeLink
$3192.next=$3194
$3192.entry=$3184
$3184.type=MathString
$3184.entry=$3186
$3186.type=String
$3194.type=AlternativeLink
$3194.entry=$3188
$3188.type=Prime
$3188.entry=$3180
$3180.type=Var
$3180.name=$3182
$3182.type=String
VALUE($3182) = f
VALUE($3186) = f^\prime
```


$3190
type
linkedList
Alternative
$3192
next
entry
type
$3194
$3184
type
entry
type
entry
AlternativeLink
$3188
MathString
f^\prime
type
entry
Prime
$3180
type
name
type
Var
f
type
String

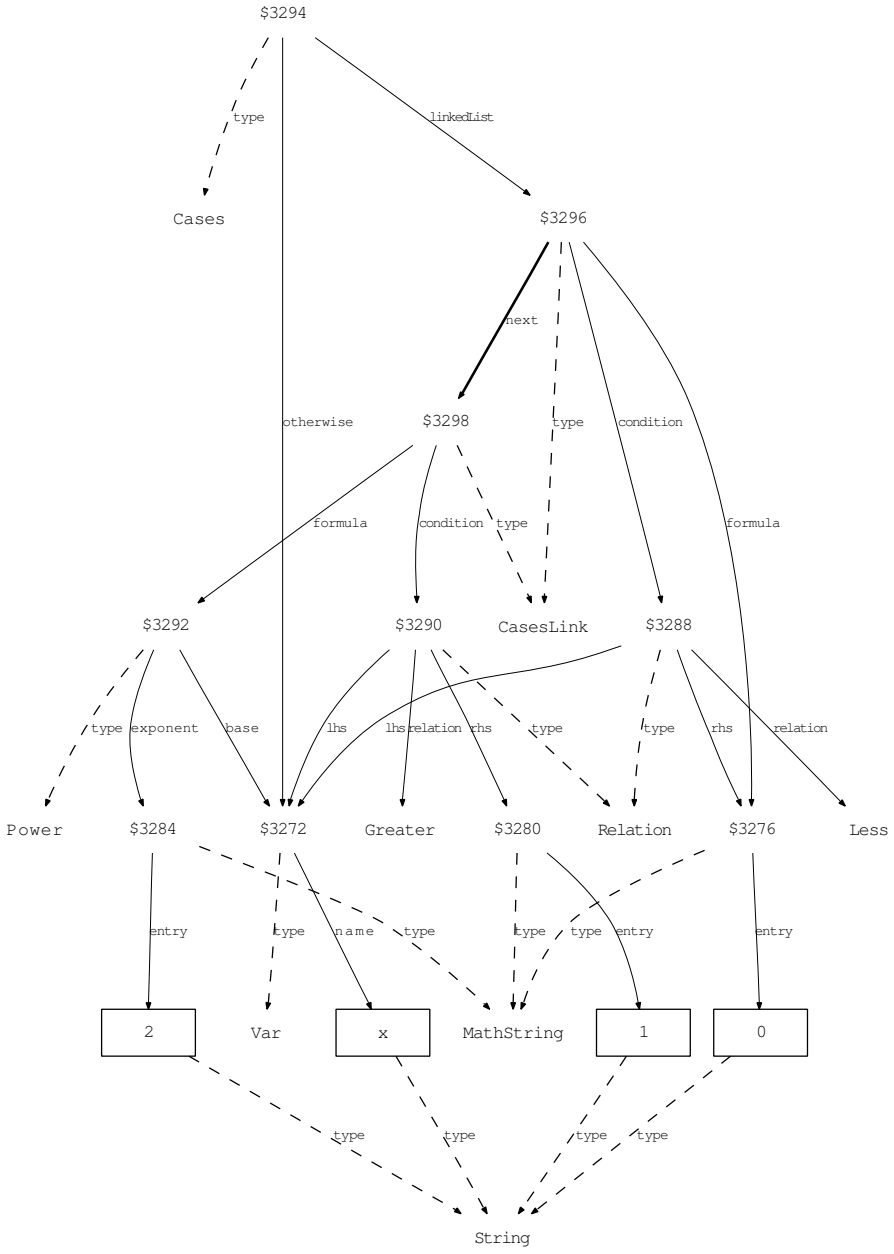
C.9 Case distinction

Example 27.

$$\begin{cases} 0 & \text{if } x<0 \\ x^2 & \text{if } x>1 \\ x & \text{otherwise} \end{cases}$$

```
\begin{cases} 0 & \text{if }x{ < }0 \\{x}^{2} & \
   text{if }x{ > }1 \\x & \text{otherwise}\end{cases
   }
```

```
$3294.type=Cases
$3294.otherwise=$3272
$3294.linkedList=$3296
$3272.type=Var
$3272.name=$3274
$3274.type=String
$3296.type=CasesLink
$3296.next=$3298
$3296.formula=$3276
$3296.condition=$3288
$3276.type=MathString
$3276.entry=$3278
$3278.type=String
$3288.type=Relation
$3288.lhs=$3272
$3288.relation=Less
$3288.rhs=$3276
$3298.type=CasesLink
$3298.formula=$3292
$3298.condition=$3290
$3290.type=Relation
$3290.lhs=$3272
$3290.relation=Greater
$3290.rhs=$3280
$3280.type=MathString
$3280.entry=$3282
$3282.type=String
$3292.type=Power
$3292.base=$3272
$3292.exponent=$3284
$3284.type=MathString
$3284.entry=$3286
$3286.type=String
VALUE($3274) = x
VALUE($3278) = 0
VALUE($3282) = 1
VALUE($3286) = 2
```


$3294
type
linkedList
Cases
$3296
next
otherwise
$3298
type
condition
formula
condition
type
formula
$3292
$3290
CasesLink
$3288
type
exponent
base
lhs
lhs
relation
rhs
type
type
rhs
relation
Power
$3284
$3272
Greater
$3280
Relation
$3276
Less
entry
type
name
type
type
type
entry
entry
2
Var
x
MathString
1
0
type
type
type
type
String

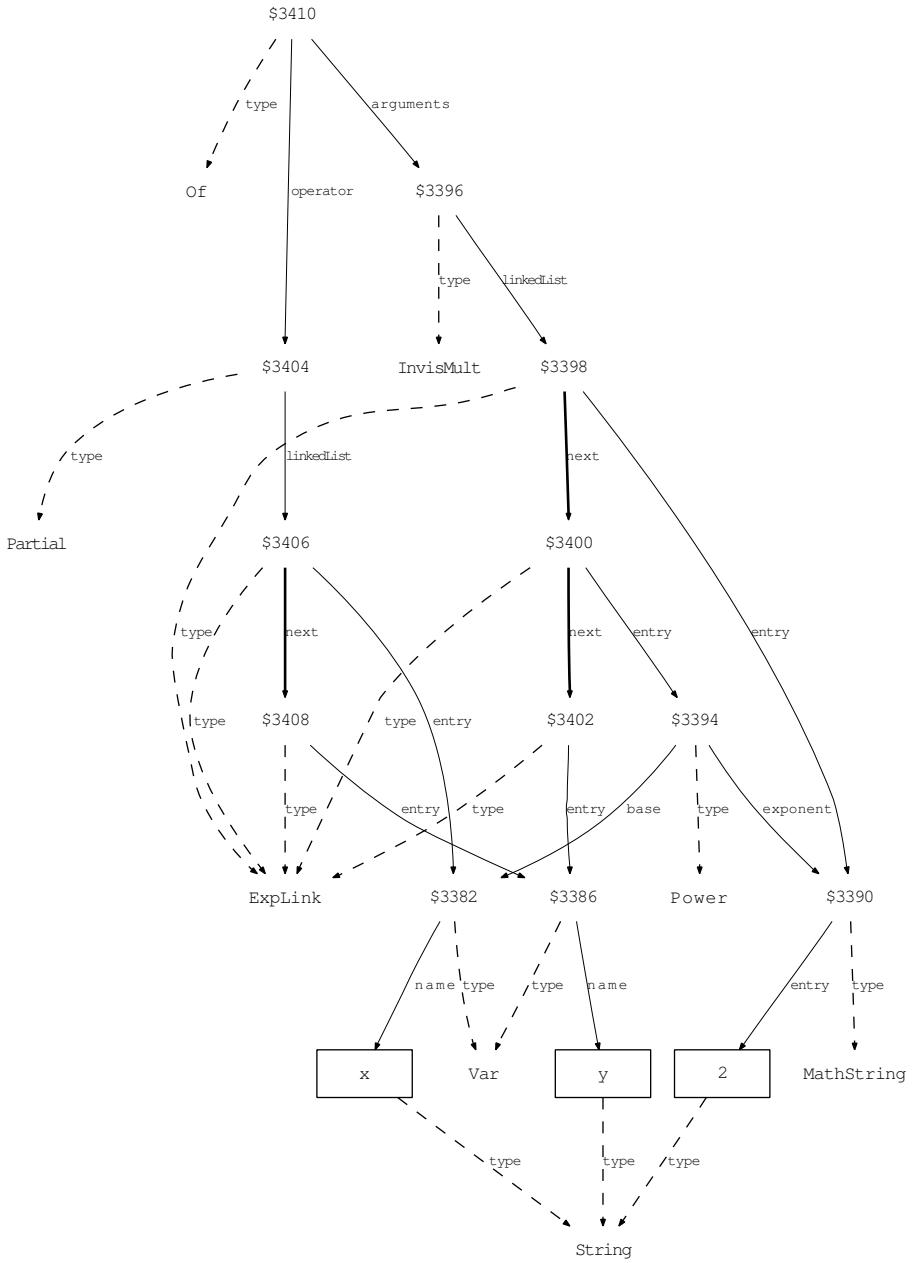
C.10 Partial derivatives

Example 28.

$$\frac{\partial^2}{\partial x \partial y} 2x^2 y$$

```
\frac{\partial^{2}}{ \partial x \partial y}2 {x}^{2}
    y
```

```
$3410.type=Of
$3410.operator=$3404
$3410.arguments=$3396
$3396.type=InvisMult
$3396.linkedList=$3398
$3398.type=ExpLink
$3398.next=$3400
$3398.entry=$3390
$3390.type=MathString
$3390.entry=$3392
$3392.type=String
$3400.type=ExpLink
$3400.next=$3402
$3400.entry=$3394
$3394.type=Power
$3394.base=$3382
$3394.exponent=$3390
$3382.type=Var
$3382.name=$3384
$3384.type=String
$3402.type=ExpLink
$3402.entry=$3386
$3386.type=Var
$3386.name=$3388
$3388.type=String
$3404.type=Partial
$3404.linkedList=$3406
$3406.type=ExpLink
$3406.next=$3408
$3406.entry=$3382
$3408.type=ExpLink
$3408.entry=$3386
VALUE($3384) = x
VALUE($3388) = y
VALUE($3392) = 2
```


$3410
type
arguments
Of
operator
$3396
type
linkedList
$3404
InvisMult
$3398
type
linkedList
next
Partial
$3406
$3400
type
next
next
entry
entry
type
$3408
type
entry
$3402
$3394
type
entry
type
entry
base
type
exponent
ExpLink
$3382
$3386
Power
$3390
name
type
type
name
entry
type
x
Var
y
2
MathString
type
type
type
String

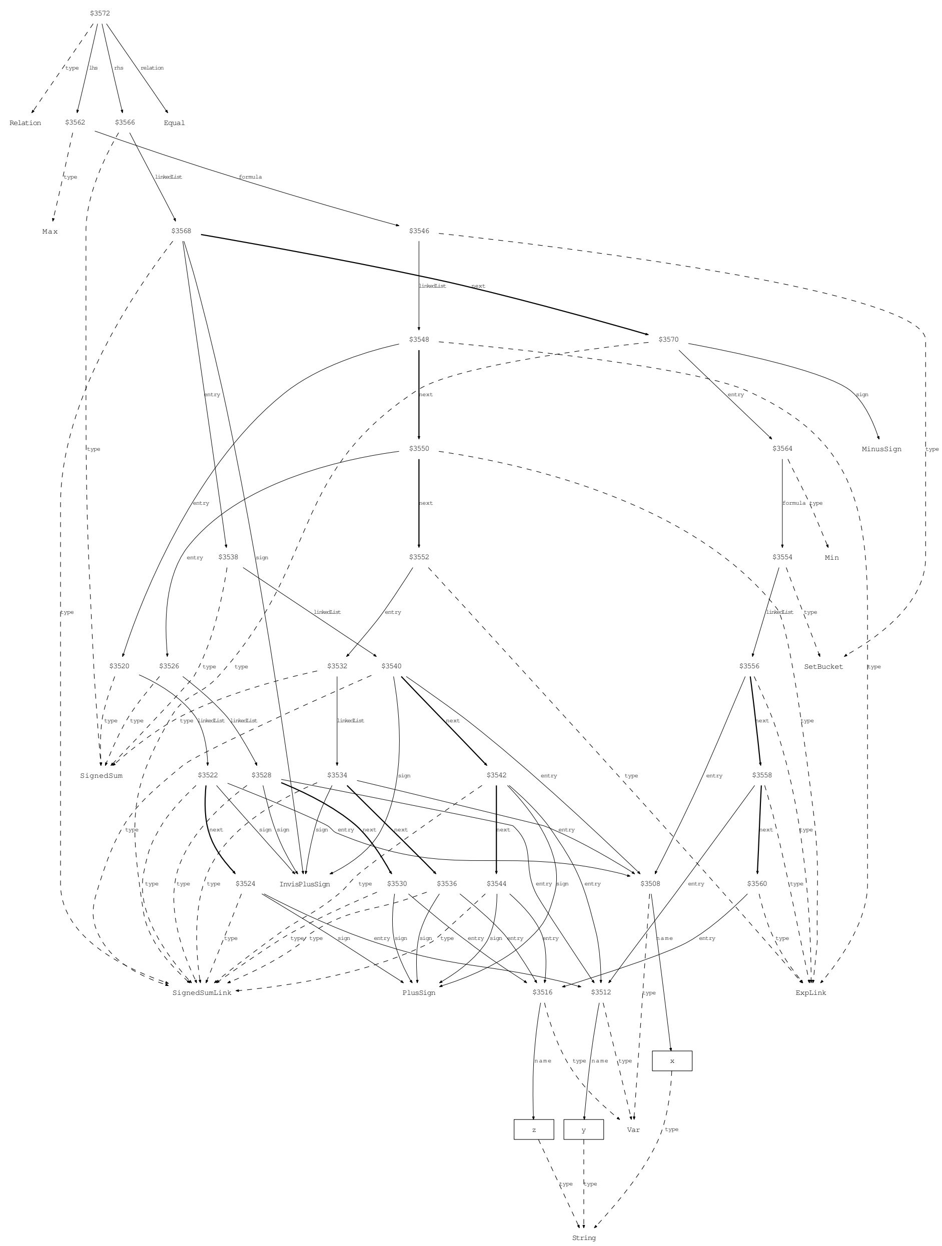
C.11 Minimum and maximum

Example 29.

$$\max\{x+y, y+z, x+z\}{=}x+y+z-\min\{x,y,z\}$$

```
\max{ \left\{ x + y , y + z , x + z \right\} }{{=}}
   x + y + z - \min{ \left\{ x , y , z \right\} }
```

```
$3572.type=Relation
$3572.lhs=$3562
$3572.relation=Equal
$3572.rhs=$3566
$3562.type=Max
$3562.formula=$3546
$3546.type=SetBucket
$3546.linkedList=$3548
$3548.type=ExpLink
$3548.next=$3550
$3548.entry=$3520
$3520.type=SignedSum
$3520.linkedList=$3522
$3522.type=SignedSumLink
$3522.next=$3524
$3522.sign=InvisPlusSign
$3522.entry=$3508
$3508.type=Var
$3508.name=$3510
$3510.type=String
$3524.type=SignedSumLink
$3524.sign=PlusSign
$3524.entry=$3512
$3512.type=Var
$3512.name=$3514
$3514.type=String
$3550.type=ExpLink
$3550.next=$3552
$3550.entry=$3526
$3526.type=SignedSum
$3526.linkedList=$3528
$3528.type=SignedSumLink
$3528.next=$3530
$3528.sign=InvisPlusSign
$3528.entry=$3512
$3530.type=SignedSumLink
$3530.sign=PlusSign
$3530.entry=$3516
$3516.type=Var
$3516.name=$3518
$3518.type=String
$3552.type=ExpLink
$3552.entry=$3532
$3532.type=SignedSum
$3532.linkedList=$3534
$3534.type=SignedSumLink
$3534.next=$3536
$3534.sign=InvisPlusSign
$3534.entry=$3508
$3536.type=SignedSumLink
$3536.sign=PlusSign
$3536.entry=$3516
$3566.type=SignedSum
$3566.linkedList=$3568
$3568.type=SignedSumLink
$3568.next=$3570
$3568.sign=InvisPlusSign
$3568.entry=$3538
$3538.type=SignedSum
$3538.linkedList=$3540
$3540.type=SignedSumLink
$3540.next=$3542
$3540.sign=InvisPlusSign
$3540.entry=$3508
$3542.type=SignedSumLink
$3542.next=$3544
$3542.sign=PlusSign
$3542.entry=$3512
$3544.type=SignedSumLink
$3544.sign=PlusSign
$3544.entry=$3516
$3570.type=SignedSumLink
$3570.sign=MinusSign
$3570.entry=$3564
$3564.type=Min
$3564.formula=$3554
$3554.type=SetBucket
$3554.linkedList=$3556
$3556.type=ExpLink
$3556.next=$3558
$3556.entry=$3508
$3558.type=ExpLink
$3558.next=$3560
$3558.entry=$3512
$3560.type=ExpLink
$3560.entry=$3516
VALUE($3510) = x
VALUE($3514) = y
VALUE($3518) = z
```

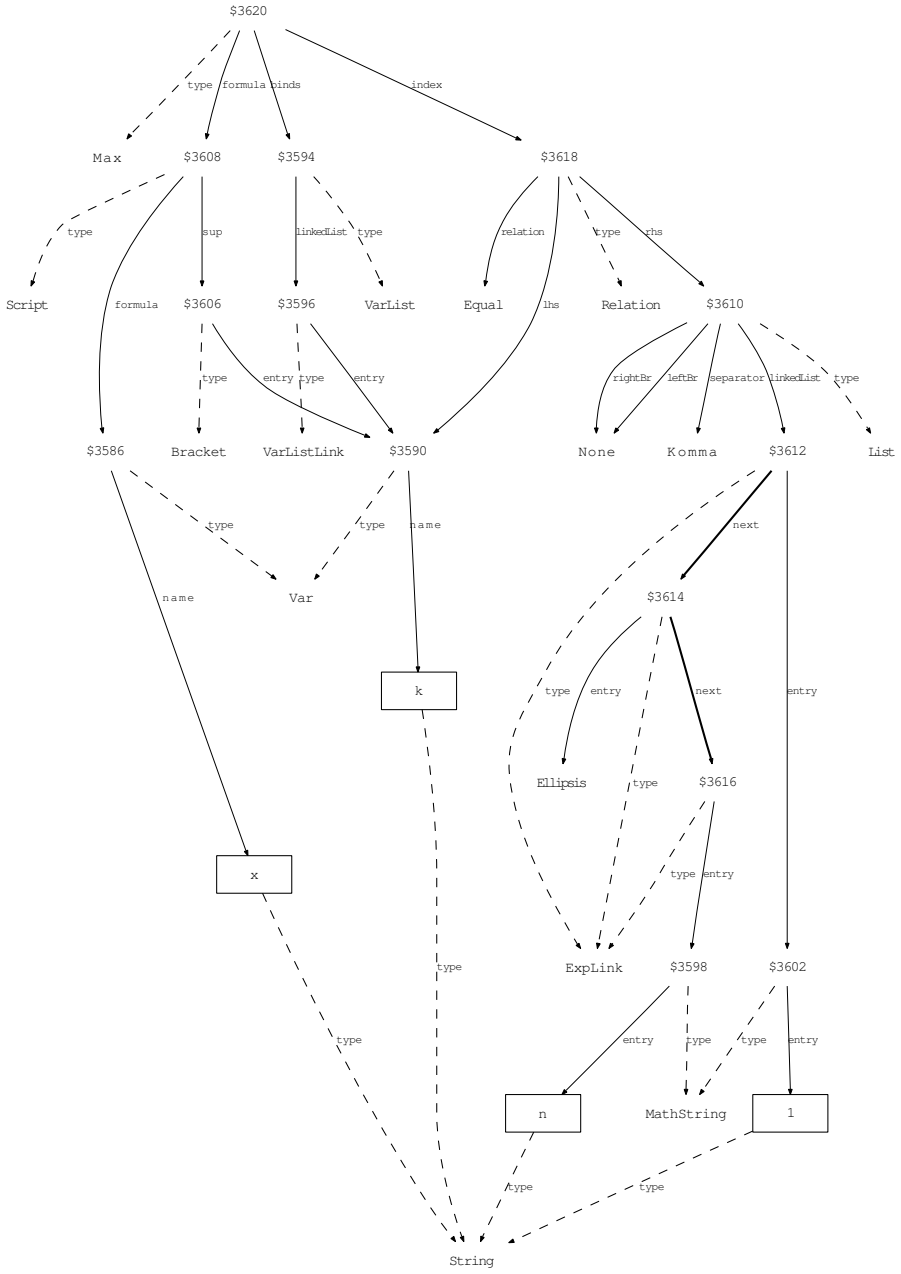

$3572
Relation
$3562
$3566
Equal
Max
$3568
$3546
$3548
$3570
$3550
$3564
MinusSign
$3538
$3552
$3554
Min
$3520
$3526
$3532
$3540
$3556
SetBucket
SignedSum
$3522
$3528
$3534
$3542
$3558
$3524
InvisPlusSign
$3530
$3536
$3544
$3508
$3560
SignedSumLink
PlusSign
$3516
$3512
ExpLink
x
z
y
Var
String

Example 30.

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

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

```
$3620.type=Max
$3620.formula=$3608
$3620.binds=$3594
$3620.index=$3618
$3594.type=VarList
$3594.linkedList=$3596
$3596.type=VarListLink
$3596.entry=$3590
$3590.type=Var
$3590.name=$3592
$3592.type=String
$3608.type=Script
$3608.formula=$3586
$3608.sup=$3606
$3586.type=Var
$3586.name=$3588
$3588.type=String
$3606.type=Bracket
$3606.entry=$3590
$3618.type=Relation
$3618.lhs=$3590
$3618.relation=Equal
$3618.rhs=$3610
$3610.type=List
$3610.leftBr=None
$3610.separator=Komma
$3610.rightBr=None
$3610.linkedList=$3612
$3612.type=ExpLink
$3612.next=$3614
$3612.entry=$3602
$3602.type=MathString
$3602.entry=$3604
$3604.type=String
$3614.type=ExpLink
$3614.next=$3616
$3614.entry=Ellipsis
$3616.type=ExpLink
$3616.entry=$3598
$3598.type=MathString
$3598.entry=$3600
$3600.type=String
VALUE($3588) = x
VALUE($3592) = k
VALUE($3600) = n
VALUE($3604) = 1
```


$3620
type
formula
binds
index
Max
$3608
$3594
$3618
type
sup
linkedList
type
relation
type
rhs
Script
formula
$3606
$3596
VarList
Equal
lhs
Relation
$3610
type
entry
type
entry
rightBr
leftBr
separator
linkedList
type
$3586
Bracket
VarListLink
$3590
None
Komma
$3612
List
type
type
name
next
name
Var
$3614
k
type
entry
next
entry
Ellipsis
type
$3616
x
type
entry
type
ExpLink
$3598
$3602
type
entry
type
type
entry
n
MathString
1
type
type
String

Appendix D

Examples of problems from the OR-Library

As with representation of informal mathematical text (Section 5.2), the automatically created output contains grammatical errors, which we plan to overcome by interfacing the Grammatical Framework [37].

D.1 Multi-dimensional knapsack.

Let N that is integer be cardinality of contract and let M that is integer be cardinality of budget. Let c_j for $j \in \{1, \ldots, N\}$ be contract volume of project j. Let $A_{i,j}$ for $i \in \{1, \ldots, M\}$ and $j \in \{1, \ldots, N\}$ be estimated cost of project j and budget i. Let B_i for $i \in \{1, \ldots, M\}$ be available amount of budget i. Let $x_j = 1$ if project j is selected, and $x_j = 0$ otherwise for $j \in \{1, \ldots, N\}$.

Problem: Given N that is integer , M that is integer , N-dimensional vector c, $M \times N$-matrix A and M-dimensional vector B, find N-dimensional vector x that is binary 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 $i \in \{1, \ldots, M\}$.
The OR-Lib contains the files :
`mknap1.txt`,
`mknapcb<i>.txt~~(i=1:9)`
to be read with:

```
read nr_of_instances
newline
for 1:nr_of_instances
  read M
  read N
  skip S
  newline
  for i = 1:M
    read B_i
  end
  newline
  for j = 1:N
    for i = 1:M
      read A_{i,j}
    end
    newline
  end
  for j = 1:N
    read c_j
  end
  newline
end
```

Output generated for AMPL:

```
param N integer ;
param M integer ;
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
 constraint1{i in 1..M} : sum{j in 1..N}(A[i , j] *
    x[j]) <= B[i];
```

D.2 Multi-demand multi-dimensional knapsack.

Problem: Given m that is integer , n that is integer , q that is integer , $m + q \times n$-matrix a that is nonnegative , $m + q$-dimensional vector b that is nonnegative and n-dimensional vector c, find n-dimensional vector x that is

binary such that

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

is maximal under the constraint
$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$ for $i \in \{1, \ldots, m\}$
and
$\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ for $i \in \{m+1, \ldots, m+q\}$. The OR-Lib contains the files :
`mdmkp_test.txt`
to be read with:

```
read nr_of_instances
newline
for 1:nr_of_instances
  read n
  read m
  newline
  for i = 1:m
    for j = 1:n
      read a_ij
    end
    newline
  end
  for i = 1:m
    read b_i
  end
  newline
  for i = m + 1:m + m
    for j = 1:n
      read a_ij
    end
    newline
  end
  for i = m + 1:m + m
    read b_i
  end
  newline
  for j = 1:n
    read c1_j
  end
  newline
  for j = 1:n
    read c2_j
  end
  newline
  for j = 1:n
    read c3_j
  end
  newline
  for j = 1:n
    read c4_j
  end
  newline
  for j = 1:n
    read c5_j
  end
  newline
  for j = 1:n
    read c6_j
  end
  newline
end
```

. Choose q=1 and c=$c1$ or q=$\frac{m}{2}$ and c=$c2$ or q=m and c=$c3$ or q=1 and

c=$c4$ or q=$\frac{m}{2}$ and c=$c5$ or q=m and c=$c6$.

Output generated for AMPL:

```
param m integer ;
param n integer ;
param q integer ;
param a{i in 1..m+q , j in 1..n} >= 0 ;
param b{i in 1..m+q} >= 0 ;
param c{j in 1..n} ;
var x{j in 1..n} binary ;
maximize
 target : sum{j in 1..n}(c[j] * x[j]);
subject to
 constraint0{i in 1..m} : sum{j in 1..n}(a[i , j] *
    x[j]) <= b[i];
subject to
 constraint1{i in m+1..m+q} : sum{j in 1..n}(a[i , j
    ] * x[j]) >= b[i];
```

D.3 Portfolio optimization.

Let N that is integer be cardinality of available asset. For i=$\{1,\ldots,N\}$ let r_i be expected return of asset i. For i=$\{1,\ldots,N\}$ and j∈$\{1,\ldots,N\}$ let c_{ij} be covariance of asset i and asset j. Let real number R be desired expected return. For i=$\{1,\ldots,N\}$ let real number w_i be held proportion of asset i.

Given N that is integer , N-dimensional vector r, N×N-matrix c and real number R, find N-dimensional vector w such that

$$\sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j c_{ij}$$

is minimal under the constraint
$\sum_{i=1}^{N} w_i r_i$=R
and
$\sum_{i=1}^{N} w_i$=1
and
0≤w_i for i=$\{1,\ldots,N\}$
and
w_i≤1 for i=$\{1,\ldots,N\}$.

The OR-Lib contains the files :
`port<i>~~(i=1:5)`
to be read with:

```
read N
newline
for i = 1:N
  read r_i
  skip d_i
  newline
end
until EOF
  read i
  read j
  read c_ij
  newline
end
```

.

Let $V := \sum_{i=1}^{N} \sum_{j=1}^{N} w_i w_j c_{ij}$. The OR-Lib contains solutions in the files :
`portef<i> (i=1:5)`
to be read with:

```
until EOF
 read R
 read V
 newline
end
```

.

Output generated for AMPL:

```
param N integer ;
param r{i in 1..N} ;
param c{i in 1..N , j in 1..N} ;
param R ;
var w{i in 1..N} ;
minimize
 target : sum{i in 1..N}(sum{j in 1..N}(w[i] * w[j]
    * c[i , j]));
subject to
 constraint0 : sum{i in 1..N}(w[i] * r[i]) = R;
subject to
 constraint1 : sum{i in 1..N}(w[i]) = 1;
subject to
 constraint2{i = 1..N} : 0 <= w[i];
subject to
 constraint3{i = 1..N} : w[i] <= 1;
```

D.4 Set partitioning problem.

For element i and set j let $a_{i,j}$=1 if $i \in j$, and 0 otherwise. Let c_j be cost of column j. Let x_j=1 if set j is selected, and x_j=0 otherwise.

Given m that is integer , n that is integer , $m \times n$-matrix a that is binary and n-dimensional vector c, find n-dimensional vector x that is binary such that

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

is minimal under the constraint $\sum_{j=1}^{n} a_{i,j} x_j = 1$ for $i \in \{1, \ldots, m\}$.

The OR-Lib contains the files :
`sppnw<i> (i=01:43)`,
`sppaa0<i> (i=1:6)`,
`sppus0<i> (i=1:4)`,
`sppkl0<i> (i=1:2)`
to be read with:

```
read m
read n
for j = 1:n
   read c_j
   read K
   for k = 1:K
      read i
      set a_{i,j} := 1
   end
end
```

.

Output generated for AMPL:

```
param m integer ;
param n integer ;
param a{i in 1..m , j in 1..n} binary ;
param c{j in 1..n} ;
var x{j in 1..n} binary ;
minimize
 target : sum{j in 1..n}(c[j] * x[j]);
subject to
 constraint1{i in 1..m} : sum{j in 1..n}(a[i , j] *
    x[j]) = 1;
```

D.5 Set covering problem.

For element i and set j let $a_{i,j}$=1 if $i \in j$, and 0 otherwise. Let c_j be cost of column j. Let x_j=1 if set j is selected, and x_j=0 otherwise.

Given m that is integer , n that is integer , $m \times n$-matrix a that is binary and n-dimensional vector c, find n-dimensional vector x that is binary such that

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

is minimal under the constraint $\sum_{j=1}^{n} a_{i,j} x_j \geq 1$ for $i \in \{1, \ldots, m\}$.

The OR-Lib contains the files :
`scp4<i> (i=1:10)`,
`scp5<i> (i=1:10)`,
`scp6<i> (i=1:5)`,
`scp<i><j> (i=a,b,c,d,e j=1:5)`,
`scpr<i><j> (i=e,f,g,h j=1:5)`,
`scpcyc<i> (i=06:11)`,
`scpclr<i> (i=10:13)`
to be read with:

```
read m
read n
for j = 1:n
  read c_j
  read K
  for k = 1:K
    read i
    set a_{i,j} := 1
  end
end
```

.

Output generated for AMPL:

```
param m integer ;
param n integer ;
param a{i in 1..m , j in 1..n} binary ;
param c{j in 1..n} ;
var x{j in 1..n} binary ;
minimize
 target : sum{j in 1..n}(c[j] * x[j]);
subject to
 constraint1{i in 1..m} : sum{j in 1..n}(a[i , j] *
    x[j]) >= 1;
```

D.6 Equitable partitioning.

Given set of n student , set of m group and set of p attribute , we define that $a_{j,k}$:=1 if student j has attribute k and $a_{j,k}$:=0 otherwise. Given group i, we define that $X_{i,j}$:=1 if group i has student j and $X_{i,j}$:=0 otherwise. Let $d_{i,k}$ that is nonnegative be overallocation of group i and attribute k. Let $e_{i,k}$ that is nonnegative be underallocation of group i and attribute k.

Given m that is integer , n that is integer , p that is integer and $n\times p$-matrix a that is binary , find $m\times p$-matrix d that is nonnegative , $m\times p$-matrix e that is nonnegative and $m\times n$-matrix X that is binary such that

$$\sum_{k=1}^{p}\sum_{i=1}^{m} d_{i,k} + e_{i,k}$$

is minimal under the constraint
$\sum_{i=1}^{m} X_{i,j}$=1 for $j\in\{1,\ldots,n\}$
and
$\sum_{j=1}^{n} X_{i,j}a_{j,k} - d_{i,k} + e_{i,k} = \sum_{j=1}^{n} \frac{a_{j,k}}{m}$ for $i\in\{1,\ldots,m\}, k\in\{1,\ldots,p\}$.
The OR-Lib contains the files :
`epprandom<i> (i=1:5)`,
`eppperf<i> (i=1:5)`
to be read with:

```
set j := 1
until EOF
   set k := 1
   until EOL
      read a_{j,k}
      k++
   end
   j++
   newline
end
set n := j - 1
set p := k - 1
```

. Choose m.

Output generated for AMPL:

```
param m integer ;
param n integer ;
param p integer ;
param a{j in 1..n , k in 1..p} binary ;
var d{i in 1..m , k in 1..p} >= 0 ;
var e{i in 1..m , k in 1..p} >= 0 ;
var X{i in 1..m , j in 1..n} binary ;
```

```
minimize
 target : sum{k in 1..p}(sum{i in 1..m}(d[i , k]+e[i
     , k]));
subject to
 constraint0{j in 1..n} : sum{i in 1..m}(X[i , j]) =
     1;
subject to
 constraint1{i in 1..m , k in 1..p} : sum{j in 1..n
    }(X[i , j] * a[j , k]-d[i , k]+e[i , k]) = sum{j
     in 1..n}((a[j , k])/(m));
```

D.7 Data envelopment problem.

Let s be cardinality of output measure, let t be cardinality of input measure and let n be cardinality of decision making unit. Let $y_{i,k}$ be value of output measure i for decision making unit k. Let $x_{j,k}$ be value of input measure j for decision making unit k. Let u_i be weight for output measure i and let v_j be weight for input measure j. We define $S(k)$ as $\sum_{i=1}^{s} u_i y_{i,k}$ and we define $T(k)$ as $\sum_{j=1}^{t} v_j x_{j,k}$.

Given s that is integer , t that is integer , n that is integer , $t \times n$-matrix x and $s \times n$-matrix y, find s-dimensional vector u that is nonnegative and t-dimensional vector v that is nonnegative such that $\frac{S(k)}{T(k)}$ is maximal for $k \in \{1, \ldots, n\}$.

Output generated for AMPL:

```
param s integer ;
param t integer ;
param n integer ;
param x{i in 1..t , k in 1..n} ;
param y{j in 1..s , k in 1..n} ;
var u{i in 1..s} >= 0 ;
var v{j in 1..t} >= 0 ;
maximize
 target{k in 1..n} : (sum{i in 1..s}(u[i] * y[i , k
    ]))/(sum{j in 1..t}(v[j] * x[j , k]));
```

Appendix E

Examples from Naproche

Again, as with representation of informal mathematical text (Section 5.2), the automatically created output contains grammatical errors, which we plan to overcome by interfacing the Grammatical Framework [37].

E.1 Burali-Forti paradox

```
Axiom.
There is no $y$ such that $y \in \emptyset$.
Axiom.
For all $x$ it is not the case that $x \in x$.
Define $x$ to be transitive if and only if for all
   $u$ , $v$ it is the case that if $u \in v$ and $v
    \in x$ then $u \in x$. Define $x$ to be a
   ordinal if and only if $x$ is transitive and for
   all $y$ it is the case that if $y \in x$ then $y$
    is transitive.
Theorem.
$\emptyset$ is a ordinal.
Proof.
Assume$u \in v$ and $v \in \emptyset$. Hence there
   is a $x$ such that $x \in \emptyset$.
   Contradiction. Thus $\emptyset$ is transitive.
   Assume$y \in \emptyset$. Hence there is a $x$
   such that $x \in \emptyset$. Contradiction. Hence
    for all $y$ it is the case that if $y \in \
   emptyset$ then $y$ is transitive. Thus $\
   emptyset$ is a ordinal. Qed.
Theorem.
```

```
For all $x$ , $y$ it is the case that if $x \in y$
   and $y$ is a ordinal then $x$ is a ordinal.
Proof.
Assume$x \in y$ and $y$ is a ordinal. Hence for all
   $v$ it is the case that if $v \in y$ then $v$ is
   transitive. Hence $x$ is transitive. Assume$u \in
    x$. Hence $u \in y$ and $u$ is transitive. Thus
   $x$ is a ordinal. Qed.
Theorem.
There is no $x$ such that for all $u$ it is the case
    that $u \in x$if and only if$u$ is a ordinal.
Proof.
Assume there is a $x$ such that for all $u$ it is
   the case that $u \in x$if and only if$u$ is a
   ordinal. Lemma: $x$ is a ordinal.
Proof.
Assume$u \in v$ and $v \in x$. Hence $v$ is a
   ordinal and $u$ is a ordinal and $u \in x$. Thus
   $x$ is transitive. Assume$v \in x$. Hence $v$ is
   a ordinal and $v$ is transitive. Thus $x$ is a
   ordinal. Qed.
Hence $x \in x$. Contradiction. Qed.
```

E.2 An example from elementary group

```
Axiom 1.
For all $x$ , $y$ , $z$ it is the case that $ ( ( x
   * y ) * z ) = ( x * ( y * z ) ) $.
Axiom 2.
For all $x$ it is the case that $ ( 1 * x ) = x$ and
    $ ( x * 1 ) = x$.
Axiom 3.
For all $x$ it is the case that $ ( x * f ( x ) ) =
   1$ and $ ( f ( x ) * x ) = 1$.
Lemma 1: If $ ( u * x ) = x$ then $u = 1$.
Proof.
Assume$ ( u * x ) = x$. Hence $ ( ( u * x ) * f ( x
   ) ) = ( x * f ( x ) ) $. By axiom 1, $ ( u * ( x
   * f ( x ) ) ) = ( x * f ( x ) ) $. Hence by axiom
    3, $ ( u * 1 ) = 1$. Hence by axiom 2, $u = 1$.
   Qed.
Lemma 2: If $ ( x * y ) = 1$ then $y = f ( x ) $.
Proof.
```

```
Assume$ ( x * y ) = 1$. Hence $ ( f ( x ) * ( x * y
   ) ) = ( f ( x ) * 1 ) $ and $ ( ( f ( x ) * x ) *
    y ) = f ( x ) $. Hence $ ( 1 * y ) = f ( x ) $
   and $y = f ( x ) $. Qed.
Theorem 1.
$f ( x * y ) = ( f ( y ) * f ( x ) ) $.
Proof.
Let $u = ( ( x * y ) * ( f ( y ) * f ( x ) ) ) $.
   Hence by axiom 1, $u = ( x * ( ( y * f ( y ) ) *
   f ( x ) ) ) $. Hence $u = ( x * ( 1 * f ( x ) ) )
    = ( x * f ( x ) ) = 1$. Hence $ ( ( x * y ) * (
   f ( y ) * f ( x ) ) ) = 1$. Hence by lemma 2, $ (
    f ( y ) * f ( x ) ) = f ( x * y ) $. Qed.
```

Acknowledgements.

Support by the Austrian Science Foundation (FWF) under contract number P20631 is gratefully acknowledged.
I wish to thank Prof. Arnold Neumaier and the members of the FMathL seminar for their support and input.

Bibliography

[1] P.B. Andrews. A Universal Automated Information System for Science and Technology. In *First Workshop on Challenges and Novel Applications for Automated Reasoning*, pages 13–18, 2003.

[2] J.E. Beasley. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[3] R. Boyer et al. The QED Manifesto. *Automated Deduction–CADE*, 12:238–251, 1994.

[4] O. Caprotti and D. Carlisle. OpenMath and MathML: semantic markup for mathematics. *Crossroads*, 6(2):14, 1999.

[5] J. Clark, M. Murata, et al. Relax NG specification – Committee Specification 3 December 2001. *Web document*. `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`.

[6] Daniel I. Cohen. *Introduction to computer theory*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[7] M.A. Covington. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th annual ACM southeast conference*, pages 95–102. Citeseer, 2001.

[8] F. Domes, K. Kofler, A. Neumaier, and P. Schodl. CONCISE – The FMathL programming system. *Manuscript*, 2010.

[9] R. Fourer, D.M. Gay, and B.W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, 1990.

[10] M. Ganesalingam. *The Language of Mathematics*. PhD thesis, University of Cambridge, 2009.

[11] B. Ganter and R. Wille. *Formale Begriffsanalyse: Mathematische Grundlagen*. Springer-Verlag Berlin Heidelberg New York, 1996.

[12] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *Knowledge and Data Engineering, IEEE Transactions on*, 6(4):572–586, 1994.

[13] J.E. Hopcroft, J.D. Ullman, and A.V. Aho. *The design and analysis of computer algorithms*. Addison-Wesley, Boston, MA, USA, 1975.

[14] M. Humayoun and C. Raffalli. MathNat – Mathematical Text in a Controlled Natural Language. *Special issue: Natural Language Processing and its Applications*, page 293, 2010.

[15] S. Jefferson and D.P. Friedman. A simple reflective interpreter. *LISP and symbolic computation*, 9(2):181–202, 1996.

[16] J. Kallrath. *Modeling languages in mathematical optimization (Applied Optimization Vol. 88)*. Kluwer Academic Publisghers, Boston, Dordrecht, London, 2004. accepted for publication.

[17] F. Kamareddine and JB Wells. Computerizing mathematical text with mathlang. *Electronic Notes in Theoretical Computer Science*, 205:5–30, 2008.

[18] R. M. Karp. Reducibility among combinatorial problems. *50 Years of Integer Programming 1958-2008*, pages 219–241, 1972.

[19] K. Kofler. A Dynamic Generalized Parser for Common Mathematical Language. *PhD thesis*, 2011. In preparation.

[20] K. Kofler, P. Schodl, and A. Neumaier. Limitations in Content MathML. *Technical report*, 2009. `http://www.mat.univie.ac.at/~neum/FMathL.html#Related`.

[21] K. Kofler, P. Schodl, and A. Neumaier. Limitations in OpenMath. *Technical report*, 2009. `http://www.mat.univie.ac.at/~neum/FMathL.html#Related`.

[22] M. Kohlhase. OMDoc: Towards an internet standard for the administration, distribution, and teaching of mathematical knowledge. In *Artificial Intelligence and Symbolic Computation*, pages 32–52. Springer, 2001.

[23] M. Kohlhase. Using LATEX as a semantic markup format. *Mathematics in Computer Science*, 2(2):279–304, 2008.

[24] D. Kühlwein, M. Cramer, P. Koepke, and B. Schröder. The naproche system. *Intelligent Computer Mathematics, Springer LNCS*, 2009.

[25] O. Lassila, R.R. Swick, et al. Resource Description Framework (RDF) Model and Syntax Specification. 1999.

[26] D. Lee and W.W. Chu. Comparative analysis of six XML schema languages. *ACM SIGMOD Record*, 29(3):76–87, 2000.

[27] T.B. Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[28] F. Manola, E. Miller, et al. RDF Primer. *Web document*, 2004. W3C Recommendation.

[29] B. Miller. Latexml the manual. *Web document*, 2011. `http://dlmf.nist.gov/LaTeXML/manual.pdf`.

[30] Marvin Minsky. Size and structure of universal Turing machines using tag systems. *Proceedings of Symposia in Pure Mathematics*, 5.

[31] T. Neary and D. Woods. Small fast universal Turing machines. *Theoretical Computer Science*, 362(1–3):171–195, 2006.

[32] A. Neumaier. Analysis und lineare Algebra. *Lecture notes*, 2008. `http://www.mat.univie.ac.at/~neum/FMathL.html#ALA`.

[33] A. Neumaier. The FMathL mathematical framework. *Draft version*, 2009. `http://www.mat.univie.ac.at/~neum/FMathL.html#foundations`.

[34] A. Neumaier and P. Schodl. A Framework for Representing and Processing Arbitrary Mathematics. *Proceedings of the International Conference on Knowledge Engineering and Ontology Development*, pages 476–479, 2010. An ealier version is available at `http://www.mat.univie.ac.at/~schodl/pdfs/IC3K_10.pdf`.

[35] A. Neumaier and P. Schodl. A semantic virtual machine. 2011. submitted for publication.

[36] Piergiorgio Odifreddi. *Classical Recursion Theory*. North Holland, Amsterdam, New York, Oxford, 1999.

[37] A. Ranta. Grammatical framework. *Journal of Functional Programming*, 14(02):145–189, 2004.

[38] R. H. Richens. Preprogramming for mechanical translation. *Mechanical Translation*, 3(1):20–28, 1956.

[39] Raphael M. Robinson. Minsky's Small Universal Turing Machine. *International Journal of Mathematics*, 2(5):551–562.

[40] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

[41] P. Schodl and A. Neumaier. An experimental grammar for German mathematical text. *Manuscript*, 2009.
`http://www.mat.univie.ac.at/~neum/FMathL.html#ALA`.

[42] P. Schodl and A. Neumaier. The FMathL type system. 2011. submitted for publication.

[43] S. Shapiro. An introduction to SNePS 3. *Conceptual Structures: Logical, Linguistic, and Computational Issues*, pages 510–524, 2000.

[44] J.R. Shoenfield. *Recursion theory*. Springer-Verlag New York, 1993.

[45] J.F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. MIT Press, 2000.

[46] D. E. Stevenson. Fire Truck: A Tutorial Example of Problem Solving. *Working paper*, 2010.

[47] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[48] G. Sutcliffe and C. Suttner. The TPTP problem library. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[49] A. Trybulec and H. Blair. Computer assisted reasoning with Mizar. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28. Citeseer, 1985.

[50] J. Trzeciak. *Writing mathematical papers in English: a practical guide*. Gdańsk Teacher's Press, Gdańsk, 1995.

[51] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42(2):230–265.

[52] T. Walsh. A Grand Challenge for Computing Research: a mathematical assistant. In *First Workshop on Challenges and Novel Applications for Automated Reasoning*, pages 33–34, 2003.

[53] C. Zinn. *Understanding informal mathematical discourse*. PhD thesis, University of Erlangen-Nürnberg, 2004.

Index

Zusammenfassung

Das Projekt "a modeling system for mathematics" (MoSMath), das zur Zeit and der Universität Wien durchgeführt wird, hat die Erstellung eines Systems zur Spezifikation von numerischen Modellen zum Ziel, in einer Form wie sie für Mathematiker natürlich ist. Das spezifizierte Modell soll innerhalb des Systems repräsentiert und bearbeitet werden, und dann zu numerischen Solvern, die nicht Teil des System sind, übermittelt werden können.

Als ein erster Schritt zu einer universal einsetzbaren Software für die Repräsentation und bearbeitung von Mathematik auf dem Computer (das FMathL Projekt) entwickeln wir eine Repräsentation von Mathematik in einem Semantischen Netz (das "semantic memory"), zusammen mit einem Typsystem das die Gültigkeit der Repräsentation prüft, und einer virtuellen Maschine, die Algorithmen ausführen kann.

Der Benutzer profitiert von so einem System auf mehrfache Weise: Der offensichtlichste Vorteil ist dass der Benutzer nicht gezwungen ist eine Modellierungssprache zu erlernen und kann stattdessen die natürliche Sprache der Mathematik verwenden, welche von jedem Mathematiker, Informatiker, Physker, etc. erlernt und praktiziert wird.

Zusätzlich ist diese Art der Spezifizierung eines Modells am wenigsten Fehleranfällig, und die natürlichste Art ein Modell zu kommunizieren. Einmal in dem System repräsentiert, können ohne zusätzlichen Aufwand Ausgaben des Modells in verschiedenen Modellierungssprachen, und verschiednenen natürlichen Sprachen erzeugt werden, vorausgesetzt dass passende Transformationsroutinen verfügbar sind.

Curriculum Vitae

Name:	Peter Schodl
Date of birth:	24 August 1978
Place of birth:	Mödling, Austria
Nationality:	Austrian

Education and occupations:

1992 – 1997	HTL Mödling
Oct. 1997 – Oct. 1998	compulsory community service
1999 – 2004	Master Philosophy, University of Vienna
1999 – 2005	Master Mathematics, University of Vienna
Aug. 2005 – Feb. 2006	Voluntary social work in Sucre, Bolivia
2006 – 2008	Austrian statistical office
2008 – 2011	PhD Mathematics, University of Vienna