Exploring the Core of MathLang

Internship Report

Paul van Tilburg
paulvt@macs.hw.ac.uk

February 24, 2006

ULTRA group
School of Mathematics and Computer Science
Heriot-Watt University, Edinburgh

Under supervision of:
Prof. Fairouz Kamareddine (Heriot-Watt University)
Dr. Rob Nederpelt (Eindhoven University of Technology)

Abstract

MathLang is a language for mathematics on computers. It allows computerisation of existing and new mathematical texts written in the Common Mathematical Language, and checking the grammatical correctness of this computerisation. The framework also allows the user to take incremental steps towards the generation of a fully formalised document in such a way that the result can be checked by a proof checker.

This report describes the language itself, its grammar, elements, characteristics and one of the concrete syntaxes: the plain syntax. An encoding of a large example is presented and explained.

The main focus of the report lies on the implementation of the heart of the framework: MathLang-Core. While the architecture has changed to a more XML-centred design during the implementation, both the old and proposed new architectures are discussed. Four components can be distinguished in the framework: the parser, the abstract syntax tree, the checker and the printer. The latter has many different instances for many formats.

Finally the report considers the implemented components and their role in the new architecture. Some components can be used directly, some have to be bridged or modified.

Contents

Preface & Acknowledgements

During the period of December 2005 to February 2006 I have been working in the ULTRA group of the School of Mathematics and Computer Science at the Heriot-Watt University in Edinburgh. The work involved doing research and implementing parts of the MathLang framework that has been under construction by this group for a few years now.

I want to thank professor Fairouz Kamareddine[1] for her guidance, help, enthusiasme and support during this project and Rob Nederpelt[2] for his help in arranging this internship and preparing me well for this assignment. I also want to thank Joe Wells[3] for his insights and views on MathLang that made me constantly change, adapt and refine my own views on the language. Last but not least I want to thank my two colleagues, Manual Maarek[4] and Krzysztof Retel[5] without whom the result of my assignment wouldn't have been as it is now. Their advise, discussions and comments were essential for the quality of my implementation and this report.

[1] http://www.macs.hw.ac.uk/~fairouz/
[2] http://www.win.tue.nl/~wsinrpn/
[3] http://www.macs.hw.ac.uk/~jbw/
[4] http://www.macs.hw.ac.uk/~mm20/
[5] http://www.macs.hw.ac.uk/~retel/

Chapter 1

Introduction

My work on the MathLang project mainly concerned implementing parts of the core (see Section 2.2) and reworking some of this implementation when at a later stage the architecture was changed to achieve greater flexibility. I want to mention beforehand that not all rework has been completed. This report will mainly discuss what parts have been created (Chapter 3), what has changed (Section 3.1), what has been reworked and what still needs to be reworked (see Section 5.1).

1.1 What is MathLang?

MathLang is next to what the name suggests (a mathematical language) also a framework for writing mathematical texts. It allows for more formalisation than the normal Common Mathematical Language (CML) does in such a way that one can check the correctness of grammar of the text (on some given level). It also allows for conversion to more stringent forms at a later stage so that it can be checked by proof checkers such as (Mizar[1], Coq[2], PVS[3], etc.).

1.1.1 Vernacular

MathLang has it roots in both Mathematical Vernacular (MV) [2] and the Weak-Type Theory (WTT) [11], which also has its roots in MV.

MathLang tries to capture the "mathematical vernacular". The best way to explain what that is, is by citing The Mathematical Vernacular by N.G. de Bruin [2]:

> The word "vernacular" means the native language of the people, in contrast to the official, or the literary language (in older days in

[1] See `http://mizar.org/`.
[2] See `http://coq.inria.fr/`.
[3] See `http://pvs.csl.sri.com/`.

> contrast to the latin of the church). In combination with the word "mathematical", the vernacular is taken to mean the very precise mixture of words and formulas used by mathematicians in their better moments, whereas the "official" mathematical language is taken to be some formal system that uses formulas only.
>
> We shall use MV as abbreviation for "mathematical vernacular".
>
> This MV obeys rules of grammar which are sometimes different from those of the "natural" languages, and, on the other hand, by no means contained in current formal systems.

All "mathematical vernaculars" that are used around the world for any arbitrary kind of mathematical foundation are captured by CML: the Common Mathematical Language (CML). MathLang lives in the space between CML and the language of any formal system that uses formulas only.

1.1.2 Language on Computers

MathLang is more than just a mathematical vernacular:

> **MathLang is a language for mathematics on computers.**
>
> **MathLang is a language.** It is meant to be used for communication and as a concrete support for human mind constructions. MathLang is a constructed language aimed to synthesise the common mathematical language.
>
> **MathLang is for mathematics.** It is meant to be open to any branch of mathematics and to any topic that uses mathematics as base language. MathLang mimics mathematics in its incremental construction of a body of knowledge.
>
> **MathLang is on computers.** It is meant to be a communication medium between human and computer systems, between humans via a digital support and between computer systems. MathLang is a computerised language and therefore offers automation possibilities.
>
> *(from The MathLang Manual [7])*

History. The first version of MathLang (version 1.x) was developed as a small adaption of the WTT [13] and later refined [12]. The WTT itself was a refinement of de Bruin's MV [11]. Late in the development of the WTT-based MathLang some limitations appeared during the translation of *Euclid's Elements* [10]. At this point the language of the MathLang framework, MathLang-Core (MathLang-Core), was revised [14] and object-orientedness was added. This meant that the development of MathLang implementation had to be restarted.

Versioning. Due to the many versions and revisions available of MathLang a versioning scheme was introduced. This scheme is described in detail in [6]. A short summary:

> This MathLang grammar (and related feature grammars) is versioned using the following scheme: $x.y.z$, where each number has its own meaning:
>
> x is the major number. It is incremented when MathLang as a whole changes. [...]
>
> y is the minor number. This number is incremented when the grammar "interface" changes [...]
>
> z is the micro number. This is incremented when small changes are made to the grammar [...]

Syntaxes. Since MathLang is targeted at computerisation[4], there are several syntaxes available at the moment:

1. The XML syntax defined by a Document Type Definition (DTD) given in Appendix B. The XML documents can be generated by editors or other future applications that MathLang can be embedded in.
2. The plain syntax defined by the grammar given in Appendix A. This syntax is mainly used to create a MathLang document in a very easy way. It is close to the abstract syntax and structure of MathLang as well as the WTT syntax [11]. This syntax is more oriented towards experimenting and testing purposes rather than use in a future application.

While the Extensible Markup Language (XML) syntax is a very important aspect of the MathLang system, this document will mainly use the plain syntax for its clarity and compactness.

1.2 Notations and Conventions

As stated in the previous subsection the plain syntax is the most used in this document. The notation of the plain syntax is given by the grammar in Appendix A and will be is elaborated on in Section 2.4.

Acronyms

The following acronyms and abbreviations are used throughout this document:

AST Abstract Syntax Tree

[4]The term *computerisation* refers to storing in a computer. However, when a text is translated using a specific syntax, information will be lost. To keep this in mind we call the translating process: *encoding*.

CML Common Mathematical Language

DTD Document Type Definition

DCf Document Context feature

DOM Document Object Model

IJf Informal Justification feature

MLf Meta Logic feature

MV Mathematical Vernacular

OO Object-Oriented

XML Extensible Markup Language

WTT Weak-Type Theory

N.B. All MathLang features will have an abbreviation that starts with a normal acronym describing the function and than end with a small "f".

Glossary

Below is a list of technical terms that are used often in this report but not explained:

Abstract Syntax Tree A data structure representing something that has been parsed in a tree form. The nodes of the tree represent elements of the abstract syntax corresponding to the concrete syntax that has been parsed.

Common Mathematical Language The mixture of natural languages and formulae used in mathematics and other fields of science to express mathematical notions.

Document Type Definition The definition of an SGML or XML document that specifies which elements there are, which attributes they can have and how they can be nested (structured) in each other.

Object-Oriented A programming paradigm that organizes data structures into class definitions which can be instantiated as objects. These objects can have procedures that effect the object's internal state or allows for interaction with other objects.

1.3 Outline

The next chapter will explain MathLang as a language. It will discuss the goals, grammar and one of the concrete syntaxes of the language. The chapter after that will elaborate on the architecture of the framework and the parts that have been implemented. Thereafter an example encoding of a real text[5] will be presented and explained. Finally a conclusion will be drawn together with a description of future work in this area.

[5] That is, a text written by a mathematician about a mathematical topic.

Chapter 2

The MathLang Language

This chapter will elaborate on what MathLang exactly is, what its goals and characteristics are and how to use it. The first section will state the goals of MathLang after which the next section will give an overview of the design and extensibility. The last section will present the plain syntax of MathLang showing how to write or encode a text in MathLang.

2.1 Goals

It is easy to imagine that there are infinite ways in which CML can be implicit, informal or ambiguous. MathLang could help with this. However, having every mathematician learning a new syntax is not ideal either. So, MathLang should stay close to the CML and do its work in the background as a layer under the text that can be checked for correctness grammar-wise. Next to that should it not enforce one specific logic system, set/type theory or way of deriving truth but be as flexible as possible.

Considering the original goals of MV [2], WTT [11] and what is written above as being the most important aspects of MathLang, we can identify the following four main goals:

1. **Computerisation of mathematical texts:** The computerisation should capture the text in a precise way. This means that MathLang should not just split the CML up into chunks and store them.
2. **Forcing no specific set/type theory:** Maximal flexibility is required so that encoding of all possible mathematical text is an attainable goal.
3. **Guiding the author:** MathLang should aid the mathematician during the writing of text. It should identify problems, such as ambiguities and missing information, be uniform and consistent.
4. **Allow for stepwise path to formalisation:** Although not compulsory, MathLang should provide ways to incrementally formalise a text so that

in the end it can be verified by a proof system. At the same time the starting point should be very close to the CML text.

For example historical documents are hard to formalise, but can be computerised. A new theorem can be written and steps can be taken to convert it to FPS [18] and then to the Mizar system/language.

2.2 Core & Features

As mentioned in the previous subsection we want MathLang to allow for stepwise formalisation. This requires a design of the framework that has a layered structure.

At the heart of the MathLang framework we find MathLang-Core. It defines the mathematical grammar elements and basic structure (see Section 2.3). It is obvious that although a text can be encoded into a MathLang-Core, information loss is still inevitable. To be able to capture more of the original text and allow for example for incremental steps towards formalisation, features were introduced.

A feature is defined on top of MathLang-Core and other features and extends the capability of parsing and checking of the input text. In essence it refines the language it is built upon. Below are some descriptions listed of some of the features that were defined at the time of writing:

- **DCf:** The Document Context feature defines textual structures and entities that are present in most mathematical CML texts. It defines annotations for MathLang-Core texts for sectioning it into documents, chapters, sections, etc. The feature also allows for identifying more semantical things than chunks of text that sometimes are left implicit like assertions, lemmas, proofs, theorems, hints, their dependencies and links. With the feature enabled errors in the structure (a chapter in a section) or missing entities (a proof for a theorem) can be found.

- **IJf:** The Informal Justification feature attempts to capture words in mathematical texts like: “hence”, “by”, “since” and different kind of ways that local scopings can be used. For example introductions, hypotheses and considerations can be distinguished. With this feature these logical annotations can be checked and holes in proofs or (missing) proof obligations can be found.

- **MLf:** The Meta Logic feature attempts to fill the holes that are left behind when a text is written in Core + DCf + IJf. It is used to describe the logical framework the author needs to be able to fulfil the proof requirements so that the text can reach logical completeness next to the already available semantical and logical information.

2.3 Grammar, Elements and Object-Oriented

When writing a mathematical text in plain CML certain types of elements and structures can be identified. It has been proven useful to draw an analogy with natural language grammar (its elements and structure). MathLang tries to capture this grammar seen through "the mathematical glasses" in a CML text as done before by its "ancestors": MV and WTT. It identifies the element types and makes sure that the grammar of the text is correct and consistent.

Type	**Example**
Adjective	even, odd, trilateral
Declaration	$x \in \mathbb{R}$, $G : graph$
Definition	$ceil(x)$ is the smallest integer larger than x, $O := (0, 0)$
Noun	a natural number, circle
Set	the natural numbers, $\mathbb{Z}$
Statement	$3 > 0$, $true \Rightarrow false$
Term	the graph G, 3

Table 2.1: Basic Elements in MathLang

Table 2.1 contains a list of elements that MathLang identifies. The following two subsections will go into more detail about what these elements represent.

2.3.1 Nouns, Adjectives, Sets & Terms

A noun is a generic description that captures the characteristics of some class of objects, whereas an adjective can refine this description. For example: *vector* or *a vector* is a noun. When saying that $\overrightarrow{v}$ is *a vector*, one does not say specifically which vector but just that $\overrightarrow{v}$ has the characteristics of one: $\overrightarrow{v}$ is an object that has a *length* and *angle*.

An adjective for a vector could be that it is a *unit* vector. This fixes the *length* of the class of *unit vectors* to the value 1.

A term is a specific element of the class that satisfies these noun characteristics. Considering the previous example $\overrightarrow{w}$ could be a *vector* term, e.g. (2 2), (-2 0) or (0 1). The last vector also being a *unit vector* term.

The set of a noun can be seen as a (possibly infinite) collection of terms satisfying the noun characteristics. For example, the set of vectors $\{(x\ y) \mid x, y \in \mathbb{N}\}$ is called the vector space over the field $\mathbb{R}$.

In plain syntax (see also Section 2.4) this example would be written as:

```
1  vector := Noun { length : N;
                    rho : angle; };

   unit := Adj (vector) { length := 1; };

6  v : vector;        (* vector (-2 0) *)
   v.length := 2;
   v.rho := 180;
```

```
w : unit vector; (* unit vector (0 1) *)
w.length := 1;
w.rho := 90;
```

N.B. A comment in the plain syntax is expressed by: *(* ... *)*.

Object-oriented. A previous implementation of MathLang was based on the WTT [11] with some small adaptations. However, for some encodings it appears to be very hard to capture the essence of the text. Some object-orientedness was added. This paradigm, used in many modern program languages, contributes to the expressiveness of the language [14].

Consider the explanation of nouns, adjectives, terms and sets in the previous paragraph in a more Object-Oriented (OO) setting:

A noun is a class definition: it defines the attributes and characteristics of some group of elements.

An adjectives is a mixin. When mixed into a noun it refines the class by adding new attributes or constraints, thus creating a new class.

A term is an object instance of class defined by the noun.

A set of a noun is a collection of object instances, i.e. all object instances that are possible.

An example listing of some code in a OO language[1] that could be one of many possible implementations of the vector example:

```
class Vector
  attr :length
  attr :direction

  def initialize(len, dir)
    @length = len; @direction = dir
  end
end

module Unit
  def initialize
    super                    # call Vector.initialize first
    assert(self.length = 1)
  end
end

class UnitVector < Vector
  include Unit
end

v = Vector.new(2, 180)       # vector (-2 0)
w = UnitVector.new(1, 90)    # unit vector (0 1)
```

[1]This listing uses the Ruby programming language: `http://ruby-lang.org`.

2.3.2 Declarations, Definitions & Statements

Declarations, definitions and statements form the main lines of a text. A statement aims at stating a property or a truth, for example: $orthogonal(v, w)$. The declaration denotes the type of some identifier (see also Section 2.3.3). It also can express that some object is a member of some class (a noun, e.g. $v : vector$) or a member of a collection (set) of objects for some class (e.g. $n \in \mathbb{N}$).

Though maybe the most important elements are the definitions. A definition is more than just a mathematical definition: it binds identifiers to nouns, adjectives, terms, sets or statements and types them in the process.

Identifiers have multiple roles in MathLang. They are:

- Names for defined nouns, adjectives and sets (*vector*)
- Variables referring to objects ($\overrightarrow{v}$)
- Functions (*length*)
- Binders ($\forall_x \dots$)

The role is determined by the definition (or declaration) of the identifier.

2.3.3 Types

In MathLang all identifiers have a type[2] that consists of a list of parameter types and one result type. This type is represented with the 2-tuple:

$$T_{id} := (T_a \times \cdots \times T_a) \times T_a$$

When an identifier refers to a constant and thus has no parameters, the empty parameter type list will be denoted as ().

Element	**Type**
Adjective	$Adj(T_{map}, T_{map})$
Declaration	$Dec(T_{id})$
Definition	$Def(T_{id})$
Noun	$Noun(T_{map})$
Set	$Set(T_{map})$
Statement	$Stat$
Term	$Term(T_{map})$
Step	$Step$

Table 2.2: Basic Elements in MathLang

The parameter and result types are atomic types T_a (see Table 2.2 for the defined types). For example, the equality (=) identifier can have the type

[2] Assuming the identifier in question can be typed correctly.

$((Term(\langle\rangle),\ Term(\langle\rangle)),\ Stat)$. This can be simplified to $((Term,\ Term),\ Stat)$ and even syntax-sugared as $(Term, Term) \to Stat$.

Some atomic types wrap a type mapping T_{map}, which is a partial function of identifiers (the set I) to atomic types:

$$T_{map} := I \to T_{id}$$

This mapping captures the identifiers that were defined during a noun (class) definition, adjective (mixin) definition, set (set of object) or simple instantiation (object). Some examples:

- The noun representing a graph which has a set of vertices and set of edges could have the type: $Noun(\langle V \mapsto (() \to Set),\ E \mapsto (() \to Set)\rangle)$.
- The adjective for a vector that restricts the length:
  $Adj(\langle rho \mapsto (() \to Term),\ length \mapsto (() \to Term)\rangle)$.

If there are no identifiers declared or defined, the empty mapping is denoted by $\langle\rangle$ and usually left out of the specification of the atomic type.

The atomic types *Def* and *Dec* wrap the type of the identifier that was defined or declared respectively. For example, the type of the definition of equality (=) would result in: $Def((Term, Term) \to Stat)$.

The *Step* type is different from the other types. This type is only used for structure. All phrases, blocks and local scopings are typed *Step* if they contain one or more elements that has a valid type.

2.4 Plain Syntax

This section will explain one of MathLang's concrete syntaxes: the plain syntax. It will show how each of the elements given in previous subsections can be represented and used in the language. The first sections will show how to create expressions and the last two sections will show how to structure the text.

2.4.1 Declarations & Elementhood

A declaration defines the type of an identifier (see also Section 2.3.3). A declared identifier can then be used in a definition or an instantiation that we will see later on.

Constants: Constant declarations are usually not mentioned explicitly in the text, they are just assumed to be known. However for MathLang this means that they will have to be made explicit:

```
true : term;
nat : noun;
N : set(nat);
```

After these declarations *true* can be used as a term, *nat* could refer to a natural number and N to the set of natural numbers.

Functions, variables: Functions and variables (and predicates) can be declared in a similar way as constants. The difference is that these identifiers carry parameters.

```
S(term) : term;
not(term) : term;
=(term, term) : stat;
```

Because only a few mathematical symbols can be encoded directly in the plain syntax, some symbols have to be encoded by their name. For example, $\neg(x)$ is encoded as `not(term) : term;`.

Binders: In MathLang binders are considered to be parameterised identifiers as well. They carry a specific parameter: a declaration of some (dummy) variable. The declaration of a declaration of some variable is encoded with **dec**(‘x), where ‘x means “some variable x”. So the declarations of $\forall_x$ and $\exists_x$ are:

```
forall(dec(‘x), stat) : stat;
exists(dec(‘x), stat) : stat;
```

Elementhood: When the right-hand side of the declaration is an expression and not some abstract type the declaration is called a declaration of elementhood. This is typically used for encodings of “0 is a natural number” or $n \in \mathbb{N}$:

```
0 : nat;
n : N;
```

2.4.2 Definitions

Definitions are used to define the meaning of some identifier. Note that not all definitions are mentioned explicitly in mathematical texts, some are hidden.

Simple definitions: This definition binds an expression to an identifier. The identifier to-be-defined may have named constant[3] parameters. For example the definition of the term *false*:

```
false := not(true);
is_nat(n) := exists(m : N, =(n, m));
```

[3] A function or predicate as parameter is not allowed. These “definition parameters” will have parameters by themselves.

Definition by case: The by-case definition is slightly different from the simple definition. It does not completely define the identifier but just a specific case. Because of this, the parameters of the defined identifier can be expressions (mainly instantiations) and a different definition symbol is used (˜=).

Consider the following inductive definition of the *positive* predicate:

```
positive(0) ˜= false;
positive(S(n)) ˜= true;
```

2.4.3 Instantiation

An instantiation is just a constant, variable or binder with arguments that comply with the declared or defined type. In the previous sections some instantiations were already given using *not*, *true*, *exists*, =, n and m. A repetition of the two definitions containing these instantiations:

```
false := not(true);
is_nat(n) := exists(m : N, =(n, m));
```

2.4.4 Nouns & Adjectives

As explained in Section 2.3.1 there are also nouns and adjectives like in natural languages. There are many uses for these two element types but first they have to be defined.

Simple nouns & adjectives: The definition of simple nouns and adjectives are equivalent to their declaration. That is, to define an identifier x to be a *Noun* is to declare it to be a *noun*. The *Noun* and *Adj* expressions are considered to be more like a constructor, they define a specific class and mixin respectively. For example:

```
nat := Noun;
odd := Adj (nat);
even := Adj (nat);
```

This will make *nat* a noun and *odd* and *even* adjectives for this noun.

More contrived nouns & adjectives: As mentioned in the previous paragraph the *Noun* and *Adj* expressions can define specific classes and mixins. Consider the following definitions:

```
line := Noun { length : term; };
finite := Adj (line) { exists(l : N, =(self.length, l)); };
```

A *line* is a noun which has a length and *finite* is an adjective for lines (that is, classes or objects which have a *line* attribute) that constraints the line to have a specific length.

Note that in a noun definition **self** refers to the noun itself as if it was instantiated. In an adjective definition **self** returns to the new, refined noun and **super** to the original noun.

2.4.5 Refinement & Sub-nouns

It seems obvious that the nouns and adjectives are combinable like in natural language.

Refinement: When a noun is preceded by an adjective, a new noun is formed:

```
line_segment := finite line;
two_fold := even nat;
```

The *line_segment* could also have been defined in one go:

```
line_segment := Noun { length : term;
                       exist(l : N, =(self.length, l)); };
```

However that would mean that if the *line* definition is changed it will have no effect on the *line_segment*. This shows the reusability of adjectives.

Sub-noun: The sub-noun states for A <− B that every A is a B[4]. For MathLang this means that the noun A should at least have everything that noun B has. Given the previous definitions of *line* and *line_segment* we can write:

```
line_segment <− line;
```

Adjective statement: This statement is similar to the sub-noun statement, except that it involves a term and adjective:

```
l : line_segment;
l <− finite;
```

Here l is said to be a *line_segment*, so l is a line which is *finite* (has a finite length).

2.4.6 Phrases & blocks

The previous sections described ways to create expressions. Next to these expressions MathLang also has elements to structure the text: phrases, blocks and local scopings. The local scopings will be described in the next section. All these elements are considered to be *steps* and are ended by a semi-colon (;). A MathLang document is simply said a sequence of steps.

[4]This is similar to N.G. de Bruin's $A \ll B$ given in [2, Section 12].

Phrases: The phrase is the atomic step. It wraps one singular expression (a definition, a declaration or an instantiation). All previous examples have been phrases. Another phrase example:

```
forall(k : N,
  forall(l : N, =( f(k), f(m) )));
```

Blocks: To bundle a series of phrases (mixed with nested blocks and local scopings), one uses the block. This is a sequence of steps enclosed in curly braces ({...}):

```
{ is_zero(0) ~= true;
  is_zero(S(n)) ~= false; };
```

Note that if the whole document seems to be a list of phrases, an implicit outer block is present.

2.4.7 Local Scoping

The local scoping is a combination of two steps combined with the "local" operator (|>). It denotes that identifiers defined or declared in the first step can only be used in the second step called the body, but not further on in the text. Thus they are kept *local* to the body. For example:

```
{ n : N; or(term, term) : stat; } |>
   or( not(is_zero(n)), is_zero(n) );
```

The identifiers n and *or* are used in the body but will not be declared nor defined further on in the text until redeclared.

The local scoping can be used to encode flag notations for assumptions and introductions. For example, the snippet above could be an encoding of:

var $n \in \mathbb{N}$

declaration $\vee$

$n = 0 \vee n \neq 0$

2.4.8 Labels & References

In MathLang it is possible to label a step and refer to it later. The following encoding labels the declaration step `1 : nat;` with the label "L1" and uses it later in a local scope. Labels are preceded with a pound (#) character.

```
label #L1 1 : nat;

ref #L1 |> 1 : odd nat;
```

Chapter 3

Implementation

This chapter will discuss the status of the implementation at the time of writing. The first section will discuss the architecture of the framework and the shift of focus that occurred during implementation. The following sections will deal with the four stages that can be distinguished in the processing of a MathLang text: parsing the text, constructing the Abstract Syntax Tree (AST), checking and typing the AST, printing the AST (to several formats).

Consider the example that will be used in the following subsections to show each stage of the process:

```
(* Some terms and variables *)
0 : term;
s(term) : term;

x : term;
y : term;

(* Definition by cases of addition *)
a(x, 0)    ~= x;
a(x, s(y)) ~= s(a(x, y));
```

For the implementation the programming language OCaml [19] was used. This language was chosen mainly because of the experience of the team with the language, its type system, object-orientedness and the fact that previous implementations also had been done in this language. The code was modularised conform the four stages: AST, parser, printers, checker. These modules were used by the two resulting programs described in the last section (Section 3.7).

3.1 Framework Architecture

The implementation of the components followed two related architectures: the original one that only kept an XML-centred design in mind and the new proposed architecture that was a result of a design.

The original architecture. The original architecture is shown in Figure 3.1. The solid boxes and ellipses represent components that have been implemented, whereas the dotted shapes represent planned components.

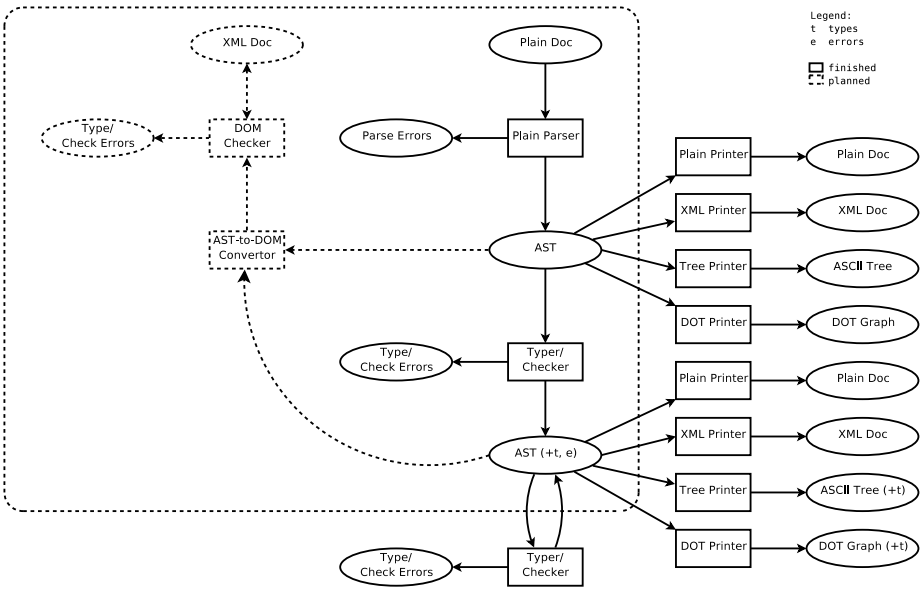


Figure 3.1: First Architecture of the Implemented Components

The concept of the architecture is quite simple. First the system parses the plain syntax document and generates an AST. During parsing it prints the errors if there are any and aborts the process. If there are no parse errors, a complete parse tree will have been constructed. On that tree the type inference/checker can do its work. It can process the tree recursively, infer the types and during inference make sure that there are no typing problems or errors. Afterwards all errors found during the inference/checking can be printed. At any moment the AST can be fed to a printer, thus converting it into another format.

The proposed architecture. Half-way during the implementation a decision was made to move to a more XML-centred architecture. Specifically a move to the Document Object Model (DOM):

> The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. [16]

Instead of the AST having a specific interface of our own design, the central data structure –the DOM graph– would have a generic and specified (see [16]) interface. This change required a revision of the first architecture. The new architecture is shown in Figure 3.2.

To be able to reuse the implemented components in this architecture, the following actions were or will have to be taken:

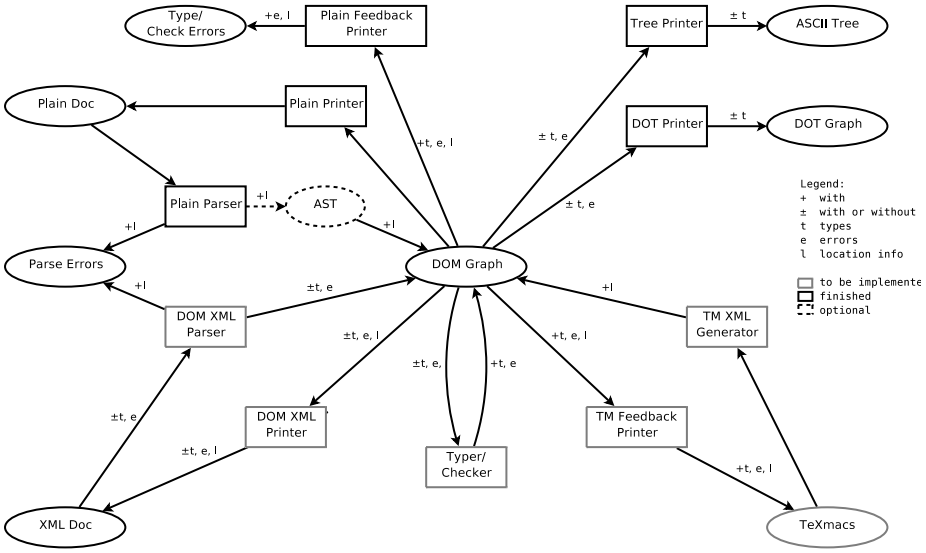


Figure 3.2: Proposed Architecture

1. The plain document, tree, and DOT printers must be modified to use the DOM graph as input.

2. The plain parser must either construct the DOM graph directly or use a component to convert the original AST into the the DOM graph.

3. The XML printer can be dropped since the DOM graph can be converted to an XML document directly.

4. Instead of the type inference/check error printer, the plain feedback printer has to be created.

At the time of writing (1), (3) and (4) have not been realised. For item (2) the implementation of this converter turned out to be the easiest solution. The converter is described in Section 3.6.

3.2 Parser

The plain syntax parser that has been implemented is described in [9]. Its job is to parse the grammar[1] described in [8, 6] and produce an Abstract Syntax Tree given that there are no errors in the syntax.

The implementation was done in OCaml [19] and Camlp4 [3] in particular. There were several problems implementing the parser due to overlapping rules in the original grammar (see [14, Section 4]). However these were all dealt with using indirect descent parsing as described in [9].

[1] The grammar is included in this document, see Appendix A.

Consider the parsing of the example text given at the beginning of Section 3. The parse sequence that is executed for this example is given in Table 3.1. The notation that is used in this table is explained in [9, Section 2]. Besides the notation the right arrow (⇒) is used to remark that something is concluded by the parser and the vertical lines (|) are used to signify nesting levels (i.e. recursive calls).

Line	Parsing Steps
1–10	`document`; *seq* = `step_seq`;
1	⇒ comment line: ignored.
2	*s*2 = `step`;
	*e*1 = `exp`;
	id = `ident`;
	'**:**';
	cat = `cat`;
	⇒ *e*1 [`0` : **term**] is a declaration of 0.
	⇒ *s*2 [`0` : **term**;] is a phrase.
⋮	⋮
9	*s*8 = `step`;
	*e*1′ = `exp`;
	id′ = `ident`;
	'**(**'; *e*′ = `exp`; *id* = `ident`;
	⇒ *e*′ [`x`] is an instantiation without parameters.
	'**,**'; *e*″ = `exp`; *id* = `ident`;
	⇒ *e*″ [`0`] is an instantiation without parameters.
	'**)**';
	⇒ *e*1′ [`a(x, 0)`] is an instantiation with two parameters.
	'**∼=**';
	*e*2′ = `exp`; *id* = `ident`;
	⇒ *e*2′ [`x`] is an instantiation without parameters.
	⇒ *s*8 [`a(x, 0) ~= x;`] is a definition-by-case.
10	*s*9 = `step`; ... etc.
1–10	⇒ *seq* is a sequence of steps (block)

Table 3.1: Parsing Sequence of the Example Text

At the end of the parsing of lines 1 to 10, the parser will have understood that 0 is declared as term, s as a function on terms, x and y as term variables. Also it will have parsed the definition of a consisting of two cases. The AST that is eventually created will be shown and explained in the next subsection.

3.3 AST

The AST plays a central role in the original architecture. This data structure captures the essence of the MathLang text (and in case of a good encoding, the CML text). The node names used in the AST correspond to rules in the grammar. This means that there are *Noun* nodes, *De*claration nodes, *Step* nodes etcetera. The plain syntax AST is described in [5].

This is the AST that is constructed after the example text has been parsed:

Line	Tree
1–10	$Block(loc, meta, "",$
2	$\langle Phrase(loc, meta, "",$
	$Dec(loc, meta, Id(loc, meta, "0"), \langle\rangle,$
	$Cterm(loc, meta, "")))$,
3	$Phrase(loc, meta, "",$
	$Dec(loc, meta, Id(loc, meta, "s"), \langle Cterm(loc, meta, "")\rangle,$
	$Cterm(loc, meta, "")))$;
5	$Phrase(loc, meta, "",$
	$Dec(loc, meta, Id(loc, meta, "x"), \langle\rangle,$
	$Cterm(loc, meta, None)))$,
6	$Phrase(loc, meta, None,$
	$Dec(loc, meta, Id(loc, meta, "y"), \langle\rangle,$
	$Cterm(loc, meta, None)))$,
9	$DefCase(loc, meta, None, Id(loc, meta, "a"),$
	$\langle Inst(loc, meta, CId(loc, meta, \langle\rangle, "x"), \langle\rangle),$
	$Inst(loc, meta, CId(loc, meta, \langle\rangle, "0"), \langle\rangle)\rangle,$
	$Inst(loc, meta, CId(loc, meta, \langle\rangle, "x"), \langle\rangle));$
10	$DefCase(loc, meta, None, Id(loc, meta, "a"),$
	$\langle Inst(loc, meta, CId(loc, meta, \langle\rangle, "x"), \langle\rangle),$
	$Inst(loc, meta, CId(loc, meta, \langle\rangle, "s"),$
	$\langle Inst(loc, meta, CId(loc, meta, \langle\rangle, "y"), \langle\rangle)\rangle)\rangle,$
	$Inst(loc, meta, CId(loc, meta, \langle\rangle, "s"),$
	$\langle Inst(loc, meta, CId(loc, meta, \langle\rangle, "a"),$
	$\langle Inst(loc, meta, CId(loc, meta, \langle\rangle, "x"), \langle\rangle),$
	$Inst(loc, meta, CId(loc, meta, \langle\rangle, "y"), \langle\rangle)\rangle\rangle\rangle\rangle\rangle\rangle)$

Extensible Markup Language (XML). Because an eventual move to XML was clear from the beginning, all elements of the AST received a mapping to an XML element or a combination of elements [5]. This is the AST of the example mapped to XML (see also Appendix B) in an XML-pseudo format:

```
mathlang---step-+-step---dec-+-id---name---"0"
                |            `-cterm
                |-step---dec-+-id---name---"s"
                |            |-cterm
                |            `-cterm
                |-step---dec-+-id---name---"x"
                |            `-cterm
                |-step---dec-+-id---name---"y"
                |            `-cterm
                |-defcase-+-id---name---"a"
                |         |-inst---cid---name---"x"
                |         |-inst---cid---name---"0"
                |         `-inst---cid---name---"x"
                `-defcase-+-id---name---"a"
                          |-inst---cid---name---"x"
                          |-inst-+-cid---name---"s"
                          |      `-inst---cid---name---"y"
                          `-inst-+-cid---name---"s"
                                 `-inst-+-cid---name---"a"
                                        |-inst---cid---name---"x"
                                        `-inst---cid---name---"y"
```

3.4 Checker

After parsing is done and the AST is constructed, the checker routine can be called to infer and check the types of all constructions and elements in the tree. The checker that has been implemented follows the rules given in [14, pages 12–14]. In most cases, the rules directly correspond to a node, however the IDENT-* rules are used throughout node checking.

Due to the shift to the Document Object Model (DOM) during implementation, the completion of this checker was abandoned and therefore only implements a part of the rules. Not (fully) implemented rules are: NOUN, ADJ, REFINEMENT, ADJ-REFINEMENT, SUBNOUN and ADJ-TERM.

An excerpt from the checking and type inference steps taken when processing the AST of the example:

Line	Checking and Type Inference Steps
1–10	Found a *Block*, check contained steps.
2	Found a *Phrase*, check contained expression.
	Found a *Dec*, check arguments.
	Found an *Id* “0”, check if already defined.
	⇒ The *Id* “0” is undeclared and thus can be declared.
	Found no category arguments for the *Id*.
	⇒ The argument list type of the declaration is ().
	Found a *CTerm* category.
	⇒ The result type of the declaration is *Term*.
	⇒ The declared *Id* “0” has type () → *Term*.
	⇒ The *Dec* has type *Dec*(() → *Term*).
	⇒ The *Phrase* has type *Step*.
⋮	⋮
9	Found a *DefCase*, check arguments.
	Found an *Id* “a”, check if already defined.
	Found an *Inst*, check arguments.
	Found an *Id* “x”, check if already defined.
	⇒ *Id* “x” is declared as () → *Term*.
	Check for arguments.
	⇒ The empty argument list type is ().
	⇒ The *Inst* has type *Term*.
	⋮
	⇒ The by-case defined *Id* “a” has type (*Term*, *Term*) → *Term*.
	⇒ The *DefCase* has type *Def*((*Term*, *Term*) → *Term*).
10	...
1–10	⇒ The *Block* has type *Step*.

Table 3.2: Checking and Type Inference of the Example Text

The feedback of the checker can notify the author of inconsistencies, ambiguities or missing parts. Also, now that the tree is typed, it can be show in a different way by the ASCII tree and DOT printers (see the next subsection).

3.5 Printers

A printer takes the AST and generates output from it in a given format. In the implementation four output printers are available:

1. **Plain printer:** attempts to regenerate the original text. This output can be used to verify that the parser has understood what the author has written.

2. **XML printer:** outputs a mapping of the AST into XML elements as described by the DTD in Appendix B. This printer has been obsoleted by the AST-to-DOM converter described in the next section.

3. **Tree printer:** outputs the AST as it can be found in the memory in a comprehensible text (ASCII) format.

   There are two types of trees that the printer can generate: the *untyped* and the *typed* tree. The typed tree variant can obviously only be printed once the checker has processed the tree.

4. **DOT printer:** a printer similar to the Tree printer, except that it generates the tree in the format of the DOT language [15]. This language describes graphs (trees) that can for example be rendered graphically to several formats (e.g. the PostScript format). The DOT printer will annotate nodes in the tree with types if they are available (i.e. the checker has processed the tree).

All printers are implemented using a recursing function on the tree that makes a case distinction on each node. The output documents of each printer when run on the AST of the example of this chapter can be found in Appendix D.

3.6 AST-to-DOM XML Converter

In Figure 3.2 on page 22 there is a bridge between the Plain Parser and the DOM Graph. This bridge would allow for the plain syntax to be used in the XML-centred framework. For this the DOM-to-DOM XML Converter was implemented.

In general its implementation is almost equal to the DOT Printer (also generating a graph). Instead of printing nodes and edges to a file it uses calls to a new library abstracting functionality[2] to create MathLang-Core DOM graphs:

- `create_element_with_children` to create a node.
- `create_element` to create an empty leaf.
- `create_element_with_text` to create a leaf containing a string.

[2] Please refer to [7, Section 6.1] for more information about the MathLang DOM libraries.

3.7 Programs

Currently, the MathLang source generates two programs: `mathlang` and `mathlang-convert`.

- `mathlang`: A program similar to the old MathLang implementation. It parses the document and runs the checker on it, both runs will give feedback if something is wrong with the document.

  Usage: `mathlang -plain <input file>`

- `matchlang-convert`: A program that will parse a MathLang-Core document in the plain syntax and convert it to other output types such as an ASCII tree (typed and untyped), DOT (typed and untyped), plain syntax, XML (normal and abbreviated). Checking and type-inference is only done when required by the output type.

  Usage:

  ```
  mathlang-convert [-t <output type> ]
                   [-o <output file>] <input file>
  ```

  The output type defaults to XML, the output file to STDOUT and the input file to STDIN. Valid values for output types: `dot`, `dot-typed`, `plain`, `tree`, `tree-typed`, `xml`, `xml-dom` and `xml-abbr`. The `xml-dom` output type will activate the DOM-to-XML converter and uses the new DOM implementation.

Chapter 4

Example

This chapter will explain the encoding of a part of the Chapter 1 (the introduction and Section 1.1) of Graph Theory book written by Reinhard Diestel [4]. The encoding presented below only captures mathematical notions from the text, i.e. no side remarks about notation or use. Also not every part of the encoding is discussed, just the more complicated parts. The original text and the complete encoding is available in Appendix C.2.

The Preamble

During any encoding you will notice that a lot of variables, meta-variables, terms, sets and functions are assumed to be known by the reader (i.e. defined elsewhere or even outside the text). However, since the checking of a document starts with a clean slate you will have to tell MathLang at least what the types of these identifiers are.

It is a custom to place the declarations that are not mentioned in the text and have no direct correlation to what is said in the text, at the top of the document. The list of declarations at the start of an encoding is called the *preamble*. The preamble is enclosed in a block so that it may be annotated by features at a later stage.

Note that the complete encoding is given in the order of the original text except for the preamble where lines are added when encountered during the encoding process.

Constructors: MathLang has no set constructor ($\{x \in S \mid \dots\}$), but it is easy to define them:

```
8  Set(dec('a), term) : set;
   Set'(dec('a), stat) : set;
```

Note that there are two variants: the first collecting terms, the second collecting set elements for which a statement is true.

Binders: Some binders are used in the text as symbols $\forall$, $\exists$, $\bigcup$, ι, or in natural language form: "for all", "there exists", "the union of", and "the *object* for which".

```
12  forall(dec(`i), stat) : stat;
    exists(dec(`i), stat) : stat;
14  iota(dec(`x), term) : term;
    Union(dec(`s), set) : term;
```

Nouns, terms, sets: A lot of (hidden) nouns are used in the text as well as terms and sets that may seem obvious, but still have to be made explicit for MathLang:

```
18  natural_number : noun;
    real_number : noun;
20  integer : noun;

22  empty_set : set;

24  0 : term;
    1 : term;
26  2 : term;
    3 : term;
28  e : term;
```

As you can see, the numbers 0, 1, 2, etc. really have to be given a type before they can be used later on.

Functions & predicates: Like hidden nouns there are also a lot of hidden functions and predicates in the text like "and", "if ... then ...", "the number of elements of" some set S.

```
    and(stat, stat) : stat;
32  disjoint(set, set) : stat;
    eq(term, term) : stat;
34  eq_set(set, set) : stat;
    element_of(term, term) : term;
36  geq(term, term) : stat;
    iff(term, term) : stat;
38  in(term, set) : stat;
    impl(stat, stat) : stat;
40  intersect(set, set) : set;
```

Sometimes the same name is used for different types. In MathLang identifiers can not be polymorphic therefore a different name has to be chosen. Compare equality (=) for terms and sets. The identifiers are encoded using the names *eq* and *eq_set* respectively.

Chapter 1: The Basics

The chapter 1 introduction starts with a few paragraphs that do not concern mathematics and are therefore ignored. We start at the following sentence: "By $\mathbb{N}$ we denote the set of natural numbers, including zero." Here *natural number* is a noun, already defined in the preamble. We introduce a set for this noun (N) and explicitly mention that 0 is a member of this set:

```
N : set(natural_number);
0 : N;
```

"The set $\mathbb{Z}/n\mathbb{Z}$ of integers modulo n is denoted by $\mathbb{Z}_n$". This set definition requires the set of integers (Z) similar to N (line 53). Since MathLang can't handle mathematical symbols we need to encode the notation $\mathbb{Z}_n$. In this notation n is a hidden natural number argument (the module factor), so we define Z_mod(n) as a function producing a set of integers modulo the argument n. The set is constructed using the *Set* constructor predefined in the preamble. This binder takes a declaration, in this case the declaration of the dummy integer variable z, and collects all integer values module n.

```
Z : set(integer);

n : natural_number |>
  { Z_mod(n) := Set(z : Z, mod(z, n));
```

(Here the natural number n is kept local to the definition of the set function.)

The next part of the sentence, "its elements are written as $\bar{i} := i + n\mathbb{Z}$.", is not encoded since it is purely about notation.

"For a real number x we denote by $\lfloor x \rfloor$ the greatest integer $\leq x$, and by $\lceil x \rceil$ the least integer $\geq x$." This sentence talks about two functions for some real number x. We put the fact that x is a real number in the local scope and define the functions in the body.

Since these functions return a specific real number, we use the ι-binder (iota). This binder is used to return a specific object for which something holds. In the case of $\lfloor x \rfloor$ this is *the specific integer* that is smaller than x but that is greater than any ($\forall$) other integer smaller than x. In other words, if an integer smaller than this specific x is found, then ($\Rightarrow$) it is smaller then this $\lfloor x \rfloor$:

```
x : real_number |>
  { floor(x) :=
      iota(i : integer, and(leq(i, x),
                            forall(j : integer, impl(leq(j, x), leq(j, i))))),
    ceil(x) :=
      iota(i : integer, and(geq(i, x)
                            forall(j : integer, impl(geq(j, x), geq(j, i)))));
```

The following sentence uses some hidden notion of a *logarithm with a base*: "Logarithms written as 'log' are taken at base 2; the natural logarithm will be denoted by 'ln'.". We declare the *logarithm-with-base* as some function and

instantiate it for the definitions of $\log x$ and $\ln x$. Note that this also ends the need for the local scope of x, so the body is closed is after the definitions:

```
    log_with_base(term, term) : term;
    log(x) := log_with_base(2, x);
    ln(x) := log_with_base(e, x); };
```

Now we encounter a more complex sentence: "A set $\mathcal{A} = \{A_1, \ldots, A_k\}$ of disjoint subsets of a set A is a *partition* of A if the union $\bigcup \mathcal{A}$ of all the sets $A_i \in A$ is A and $A_i \neq \emptyset$ for every i.". This sentence establishes several things: some set of disjoint subsets of A ($\mathcal{A}$), what a partition of a set is, and that this set $\mathcal{A}$ is a subset of A. The first two are definitions, while the latter is a statement.

For the definitions we need a local scope to declare A and the elements of the set of subsets of A. These subsets are denoted by indexes which can be seen as a function from an index to a set ($A\_ : \mathbb{N} \to \mathcal{P}(A)$). Besides that we need the number of subsets k (some natural number) and the statement that all A_i are subsets of A and disjoint with each other:

```
{ A : set; A_(term) : set; k : natural_number;
  forall(i : range(1, k),
    and(subset(A_(i), A),
        forall(j : range(1, k), disjoint(A_(i), A_(j))))); } |>
```

The use of the function range in this definition is to signify a range of (natural) numbers: range (1, k) is an encoding of $\{1, \ldots, k\}$.

Now, for the body we need the partition of A (part_A) to be this set of disjoint subsets:

```
  { part_A := Set(i : range(1, k), A_(i));
```

We also need to define the *partition_of* function, however we can not do this for $\mathcal{A}$. This would result in the definition of *partition_of* being tied directly to this specific partition of A, instead of any partition of the set A. For this purpose we introduce set B and subsets B_i with similar properties to the set A:

```
    { B : set; B_(term) : set;
      forall(i : range(1, k),
        and(subset(B_(i), B),
            forall(j : range(1, k), disjoint(B_(i), B_(j))))); } |>
      { partition_of(B, A) :=
          and(eq(A, Union(i : range(1, k), B_(i))),
              forall(i : range(1, k), not(eq(B_(i), empty_set)))); };
```

The last part is the statement that $\mathcal{A}$ (part_A) is a partition of A:

```
    partition_of(part_A, A);
```

The encoding of the sentence: "Another partition $\{A'_1, \ldots, A'_l\}$ of A *refine* the partition $\mathcal{A}$ if each A'_i is contained in some A_j." is now very trivial since it is

similar to the encoding above. We introduce the set A', the subsets A'_i in a local scoping and give the partition $\mathcal{A}'$ and the definition of *refines* in the body.

The fact that some subset is contained in the other is encoded with the mathematical notion of $\subseteq$ (subset). Also 'some A_j' is not encoded by putting the declaration of A_j in the local scope, as has been done before, but by using $\exists$ (exists).

```
    { A' : set; A'_(term) : set; } |>
87    { part_A' := Set(i : range(1, k) A'_(i));
        refines(part_A', part_A) :=
89        forall(i : range(1, k),
            exists(j : range(1, k), subset(A'_(i), A_(j)))); };
```

The last few lines of the introduction are not relevant for the rest of the encoding and are left out of the explanation. The encoding is given in lines 92–95 of Appendix C.2.2.

Section 1.1: Graphs

This section starts with the definition of the notion (noun) *graph*: "A *graph* is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$". Most of the time a graph is instantiated with some given set of vertices V and set of edges E. Because of this we will give the noun two parameters V' and E', being declared as arbitrary sets for the purpose of this definition. A piece of a sentence later in the text is also added to this definition: "we shall always assume tacitly that $V \cap E = \emptyset$.".

```
{ V' : set; E' : set; } |>
100   graph(V', E') :=
        Noun { V := V';
102            E := E';
               subset(self.E, k_set_of_subsets(2, self.V));
104            eq_set(intersection(self.V, self.E), empty_set); };
```

The prime (') is used in the parameters of the graph definition to prevent a clash with the identifiers of the noun specification.

During the entire section the identifiers V, E and G are constantly used, so we put them in a local scoping and the rest of the section in the body:

```
{ V : set; E : set; G : graph(V, E); } |>
```

Next, the text specifies what vertices and edges are: "The elements of V are the *vertices* (or *nodes*, or *points*) of the graph G, the elements of E are its *edges* (or *lines*).". The encoding can be done quite directly. It defines vertex and edge as nouns of which no properties are given and introduces some aliases.

```
{ vertex : noun;
  node := vertex;
  point := vertex;
  v : G.V |> v : vertex;

  edge : noun;
  line := edge;
  e : G.E |> e : edge;
```

N.B. When a vertex v is to be a member of the set V of the graph V, note that we use G.V and not V, since the latter has no relation to the graph. It is just a set.

The relation between V and G.V is mentioned later in the text: "A graph with vertex set V is said to be a graph *on* V. The vertex set of a graph G is referred to as $V(G)$, its edge set as $E(G)$.". Except for a small problem the translation is easy: the text mentions $V(G)$ as some function on G, however V is already declared as an identifier without arguments (i.e. type: $() \to Term$), so a different name (V') is used:

```
on(V, G) := eq_set(V, G.V);
vertex_set(G) := G.V;
V'(G) := vertex_set(G);
edge_set(G) := G.E;
E'(G) := edge_set(G);
```

The last thing said about the connection between V and G.V is related to the time I used V' as a parameter of the graph definition. When a graph is instantiated, the value of the vertex set parameter (V') is set to the V property of the noun. So, even if a graph is instantiated as H = graph(W, F) for some set W, the vertex set of that graph is still referred to with H.V. This is also mentioned in the text: "These conventions are independent of any actual names of these two sets: the vertex set W of a graph $H = (W, F)$ is still referred to as $V(H)$, not as $W(H)$.".

The following part of the text gives more definitions in a similar fashion, so we'll skip ahead to line 193 where there is more complex encoding.

Consider the following paragraph about isomorphic and isomorphism: "Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. We call G and G' *isomorphic*, and write $G \simeq G'$, if there exists a bijection $\varphi : V \to V'$ with $xy \in E \Leftrightarrow \varphi(x)\varphi(y) \in E'$ for all $x, y \in V$. Such a map φ is called an *isomorphism*;".

First, for the definition we need an addition graph G' and a bijection. Then we need to define *isomorphism* as a special bijection on vertex sets of two graphs before we can say what being isomorphic means. In this definition we need the if-and-only-if (iff) and edge (ed) statements. The latter specifies that for two vertices v_1 and v_2 of a graph the edge (v_1, v_2) is in the set of edges (see lines 160–161 of Appendix C.2.2).

```
{ V' : set; E' : set; G' : graph(V', E'); phi : bijection(V, V'); } |>
  { isomorphism(G, G') :=
    Noun { forall(x : G.V,
             forall(y : G.V,
               iff(in(ed(x, y), G.E),
                   in(ed(phi(x), phi(y)), G'.E)))); };
```

Now that the isomorphism is defined we can say what it is for the graphs G and G' to be *isomorphic*:

```
    exists(phi : term, phi : isomorphism(G, G')) |>
      isomorphic(G, G'); };
```

Consequently, the encoding of the definition of *automorphism* is simple now. It is defined in the text by: "if $G = G'$, it is called an *automorphism*.". So this is just a simpler form of the definition of isomorphism. The author means to say that we should reread the definition of isomorphism with G substituted for G' (and also implicitly V for V').

```
{ phi : bijection(V, V); } |>
  automorphism(G) :=
    Noun { forall(x : G.V,
             forall(y : G.V,
               iff(in(ed(x, y), G.E),
                   in(ed(phi(x), phi(y)), G.E)))); };
```

The rest of the encoding of the section is left as an exercise for the reader. Note that the encoding stops on page 3 at the end of the sentence: "[...] then G' is a *proper subgraph* of G.". For this encoded and a transcription of the original text, please refer to Appendix C.2.

Chapter 5

Conclusion

Hopefully this report has shown that MathLang is a simple, flexible and clear language. It has many possibilities and I am sure that it will be able to do more things than we can think of now. The current implementation described in this report is just a start.

The current implementation can: parse a MathLang-Core text, partially check this text, convert it to the XML format, show the types of the elements in the tree, provide feedback during parsing and checking. However, because the checker is not finished, complex nouns and adjectives are not checked and typed. More complex expressions like refinements and sub-nouns are not possible either.

Due to the fact that MathLang keeps changing because it has to be adapted to deal with new mathematical texts, a new implementation needed to take place. This is the reason that, while earlier checkers implemented for earlier versions of the language were completed, this checker is not finished. In the new architecture a new checker will do its work that is more resilient to language changes, as is the whole architecture.

Due to the AST-to-DOM converter, the components dealing with the plain syntax can be reused in the new architecture. This means that MathLang-Core documents can be authored using both an XML or a plain syntax editor.

5.1 Future Work

Some things are still missing from the plain syntax part of the framework:

- There is no component to convert the DOM graph back to the AST tree. If this would be possible, an author using the XML syntax could see a "plain view" on his/her text.
- A new checker for the DOM graph should be written that provides plain syntax feedback to the author. This will make the system backward compatible with what is possible now.

- Some printers that originally used the AST as input should be adapted to use the DOM graph.

Besides the plain syntax there are more areas in which work can continue:

- A TeXmacs interface could be constructed so that the mathematician doesn't have to deal with both the plain syntax or the XML syntax. The mathematician uses it tow write texts and receives feedback about the grammatical correctness and/or types inside the TeXmacs editor.
- More features should be implemented to annotate parts of the MathLang-Core document so that eventually output can be generated that interacts with a proof system. However, one can also think of other kind of features. For example a feature generating (or reconstructing) a "CML view" of the MathLang text.

Appendix A

Plain Syntax Grammar

This appendix contains a verbatim copy of the plain syntax grammar of MathLang-Core as given in [8] (with notes in [6]).

Document

⟨*document*⟩→ ⟨*step-seq*⟩

Sequences

⟨*step-seq*⟩→ { ⟨*step*⟩ **';'** }

⟨*exp-seq*⟩→ [⟨*exp*⟩ { **','** ⟨*exp*⟩ }]

⟨*cat-seq*⟩→ [⟨*cat*⟩ { **','** ⟨*cat*⟩ }]

⟨*ident-seq*⟩→ [⟨*ident*⟩ { **','** ⟨*ident*⟩ }]

Steps

⟨*step*⟩→ ⟨*exp*⟩
| ⟨*label-step*⟩
| ⟨*block-step*⟩
| ⟨*definition-step*⟩
| ⟨*local-scope-step*⟩
| ⟨*referencing-step*⟩

⟨*label-step*⟩→ ⟨*labelling*⟩ ⟨*step*⟩

⟨*block-step*⟩→ **'{'** ⟨*step-seq*⟩ **'}'**

⟨*local-scope-step*⟩→ ⟨*local-scope-cont*⟩ **'|>'** ⟨*local-scope-concl*⟩

⟨*definition-step*⟩→ ⟨*def*⟩ | ⟨*def-match-case*⟩
⟨*referencing-step*⟩→ **'ref'** ⟨*label*⟩

Labelling

⟨*labelling*⟩→ **'label'** ⟨*label*⟩

Definitions

⟨*def*⟩→ ⟨*cident*⟩ [**'('** ⟨*ident-seq*⟩ **')'**] **':='** ⟨*exp*⟩
⟨*def-match-case*⟩→ ⟨*ident*⟩ [**'('** ⟨*exp-seq*⟩ **')'**] **'~='** ⟨*exp*⟩

Local Scoping

⟨*local-scope-cont*⟩→ ⟨*step*⟩
⟨*local-scope-concl*⟩→ ⟨*step*⟩

Expressions

⟨*exp*⟩→ ⟨*inst-exp*⟩
| ⟨*decl-exp*⟩
| ⟨*sub-noun-adj-exp*⟩
| ⟨*refinement-exp*⟩
| **'('** ⟨*exp*⟩ **')'**
| **'Noun'** [**'{'** ⟨*step-seq*⟩ **'}'**]
| **'Adj'** **'('** ⟨*exp*⟩ **')'** [**'{'** ⟨*step-seq*⟩ **'}'**]
| **'self'** | **'super'**

⟨*inst-exp*⟩→ ⟨*cident*⟩ [**'('** ⟨*exp-seq*⟩ **')'**]
⟨*decl-exp*⟩→ ⟨*elementhood-decl*⟩ | ⟨*decl*⟩
⟨*sub-noun-adj-exp*⟩→ ⟨*ident*⟩ **'<-'** ⟨*exp*⟩
⟨*refinement-exp*⟩→ ⟨*exp*⟩ ⟨*exp*⟩

Declarations

⟨*elementhood-decl*⟩→ ⟨*ident*⟩ [**'('** ⟨*cat-seq*⟩ **')'**] **':'** ⟨*exp*⟩
⟨*decl*⟩→ ⟨*ident*⟩ [**'('** ⟨*cat-seq*⟩ **')'**] **':'** ⟨*cat*⟩

Categories

⟨*cat*⟩→ **‘term’** [**‘(’** ⟨*exp*⟩ **‘)’**]
| **‘noun’** [**‘(’** ⟨*exp*⟩ **‘)’**]
| **‘set’** [**‘(’** ⟨*exp*⟩ **‘)’**]
| **‘adj’** **‘(’** ⟨*exp*⟩ **‘,’** ⟨*exp*⟩ **‘)’**
| **‘dec’** **‘(’** ⟨*cat*⟩ **‘)’**
| **‘stat’**
| ⟨*cvar*⟩

Complex identifiers

⟨*cident*⟩→ { ⟨*exp*⟩ **‘.’** } ⟨*ident*⟩

Atoms

⟨*label*⟩→ **‘#’** denumerably infinite set of labels

⟨*cvar*⟩→ **‘ ‘ ’** denumerably infinite set of category variables

⟨*ident*⟩→ denumerably infinite set of identifiers

Appendix B

XML Syntax Document Type Definition

Below follows the DTD of the XML Syntax for MathLang-Core version 2.3. The external ID for usage of this DTD in MathLang-Core documents (see [17, section 2.8]) is: `http://www.macs.hw.ac.uk/~paulvt/mlc23.dtd`.

Note that in the following DTD the substructure of the elements `type`, `error env`, and `location` is left undefined by specifying `ANY` as the content. The structure will be defined at a later stage and is irrelevant for this report.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--

  DTD for MathLang-Core XML syntax, version 2.3
  ULTRA group
  $Id: mlc23.dtd,v 1.3 2006/01/04 11:52:54 paulvt Exp $

-->

<!-- Entities used in the definition -->
<!ENTITY % exp
  "( adj | dec | inst | noun | refinement | self | subnoun | super )">

<!ENTITY % category
  "( cterm | cset | cnoun | cadj | cstat | cdec | cvar )">

<!-- ================ -->
<!-- The definition -->
<!-- ================ -->

<!-- Document/Root element -->
<!ELEMENT mathlang ( meta?, step )>
```

```
<!-- Node meta data -->
<!ELEMENT meta ( type?, error?, env?, location? )>

<!ELEMENT type ANY>
<!ATTLIST type
    version          CDATA #IMPLIED
    timestamp        CDATA #IMPLIED
    first_version    CDATA #IMPLIED
    first_timestamp  CDATA #IMPLIED>

<!ELEMENT error ANY>
<!ATTLIST error
    version          CDATA #IMPLIED
    timestamp        CDATA #IMPLIED
    first_version    CDATA #IMPLIED
    first_timestamp  CDATA #IMPLIED
    id               CDATA #IMPLIED
    ref              CDATA #IMPLIED>

<!ELEMENT env ANY>
<!ATTLIST env
    version          CDATA #IMPLIED
    timestamp        CDATA #IMPLIED
    first_version    CDATA #IMPLIED
    first_timestamp  CDATA #IMPLIED>

<!ELEMENT location ANY>
<!ATTLIST location
    version          CDATA #IMPLIED
    timestamp        CDATA #IMPLIED
    first_version    CDATA #IMPLIED
    first_timestamp  CDATA #IMPLIED
    start            CDATA #REQUIRED
    end              CDATA #REQUIRED
    file             CDATA #IMPLIED>

<!-- Steps -->
<!ELEMENT label ( meta?, name )>

<!ELEMENT local ( step? )>

<!ELEMENT phrase ( meta?, ( %exp; | def | defcase ) )>

<!ELEMENT step ( meta?, label*,
                 ( step* | (local, step) | phrase) )>

<!-- Pseudo expressions -->
<!ELEMENT def ( meta?, cid, id*, %exp; )>

<!ELEMENT defcase ( meta?, id, (%exp;)+ )>

<!-- Expressions -->
<!ELEMENT adj ( meta?, %exp;, step )>
```

```
<!ELEMENT dec ( meta?, id, (%category;)*, (%exp;)? )>

<!ELEMENT inst ( meta?, cid, (%exp;)* )>

<!ELEMENT noun ( meta?, step )>

<!ELEMENT ref ( meta?, name )>

<!ELEMENT refinement ( meta?, %exp;, %exp; )>

<!ELEMENT self ( meta? )>

<!ELEMENT subnoun ( meta?, id, %exp; )>

<!ELEMENT super ( meta? )>

<!-- Categories -->
<!ELEMENT cterm ( meta?, (%exp;)? )>

<!ELEMENT cset ( meta?, (%exp;)? )>

<!ELEMENT cnoun ( meta?, (%exp;)? )>

<!ELEMENT cadj ( meta?, %exp;, (%exp;)? )>

<!ELEMENT cstat ( meta? )>

<!ELEMENT cdec ( meta?, %category; )>

<!ELEMENT cvar ( meta?, name )>

<!-- Identifiers -->
<!ELEMENT id ( meta?, name )>

<!ELEMENT cid ( meta?, (%exp;)*, name )>

<!-- Names -->
<!ELEMENT name ( #PCDATA )>
```

Appendix C

Example Encodings

This appendix contains MathLang encodings in the plain syntax of which excerpts are used in the document.

C.1 Simple Example

This example is used to explain the plain syntax in Section 2.4.

```
(**** Simple Examples for MathLang–Core 2.3 ****)

(*** EXPRESSIONS ***)

(** Declarations **)
(* Simple identifier declarations *)
true : term;
nat : noun;
N : set(nat);

(* Declaration of identifiers with arguments *)
S(term) : term;
not(term) : term;
=(term, term) : stat;

(* Binder declarations *)
forall(dec('x), stat) : stat;
exists(dec('x), stat) : stat;

(* Elementhood declarations *)
0 : nat;
n : N;

(** Definitions **)
(* Simple definition *)
false := not(true);
is_nat(n) := exists(m : N, =(n, m));
```

```
29  (* Definition by matching case *)
    positive(0) ~= false;
    positive(S(n)) ~= true;

    (** Nouns & Adjectives **)
34  (* Simple nouns & adjectives *)
    nat := Noun;
    odd := Adj (nat);
    even := Adj (nat);

39  (* More sophisticated noun & adjectives *)
    line := Noun { length : term; };
    finite := Adj (line) { exists(l : N, =(self.length, l)); };

    (* Refinement *)
44  line_segment := finite line;
    two_fold := even nat;

    (* Sub-noun *)
    line_segment <- line;
49
    (* Adjective statement *)
    l : line_segment;
    l <- finite;


54
    (*** STRUCTURE ***)

    (** Phrases **)
    forall(k : N,
59    forall(l : N, =( f(k), f(m) )));
    (* All of the above *)

    (** Blocks **)

64  { is_zero(0) ~= true;
      is_zero(S(n)) ~= false; };

    (** Local scoping **)

69  { n : N; or(term, term) : stat; } |>
       or( not(is_zero(n)), is_zero(n) );

    (** Labels & References **)

74  label #L1 1 : nat;

    ref #L1 |> 1 : odd nat;
```

C.2 Reinhard Diestel — Graph Theory

This is the main example of Chapter 4. It is an encoding into the MathLang plain syntax of the last lines of the Chapter 1 introduction and a part of Section 1.1 of Reinhard Diestel's Graph Theory [4, pages 1–3].

C.2.1 Original Text

This a transcription of the original text where sentences or paragraphs that have not been translated have been left out (marked by [...]).

1. The Basics

[...] By $\mathbb{N}$ we denote the set of natural numbers, including zero. The set $\mathbb{Z}/n\mathbb{Z}$ of integers modulo n is denoted by $\mathbb{Z}_n$; its elements are written as $\bar{i} := i + n\mathbb{Z}$. For a real number x we denote by $\lfloor x \rfloor$ the greatest integer $\leq x$, and by $\lceil x \rceil$ the least integer $\geq x$. Logarithms written as 'log' are taken at base 2; the natural logarithm will be denoted by 'ln'. A set $\mathcal{A} = \{A_1, \ldots, A_k\}$ of disjoint subsets of a set A is a *partition* of A if the union $\bigcup \mathcal{A}$ of all the sets $A_i \in A$ is A and $A_i \neq \emptyset$ for every i. Another partition $\{A'_1, \ldots, A'_l\}$ of A *refine* the partition $\mathcal{A}$ if each A'_i is contained in some A_j. By $[A]^k$ we denote the set of all k-element subsets of A. Sets with k elements will be called *k-sets*; subsets with k elements are *k-subsets*.

1.1 Graphs

A *graph* is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$; [...] we shall always assume tacitly that $V \cap E = \emptyset$. The elements of V are the *vertices* (or *nodes*, or *points*) of the graph G, the elements of E are its *edges* (or *lines*). [...]

A graph with vertex set V is said to be a graph *on* V. The vertex set of a graph G is referred to as $V(G)$, its edge set as $E(G)$. These conventions are independent of any actual names of these two sets: the vertex set W of a graph $H = (W, F)$ is still referred to as $V(H)$, not as $W(H)$. [...]

The number of vertices of a graph G is its *order*, [...] its number of edges [...] Graphs are *finite*, *infinite*, *countable* and so on according to their order. [...]

For the *empy graph* $(\emptyset, \emptyset)$ we simply write $\emptyset$. A graph of order 0 or 1 is called *trival*. [...]

A vertex v is *incident* with an edge e if $v \in e$; then e is an edge *at* v. The two vertices incident with an adge are its *endvertices* or *ends*, and an edge *joins* its ends. [...]

Two vertices x, y of G are *adjacent*, or *neighbours*, if xy is an edge of G. Two edges $e \neq f$ are *adjacent* if they have an end in common.

> If all the vertices of G are pairwise adjacent, then G is *complete*. A complete graph on n vertices is a K^n; a K^3 is called a *triangle*.
>
> Pairwise non-adjacent vertrices or edges are called *independent*. [...] a set of vertrices or of edges is *independent* (or *stable*) if no two of its elements are adjacent.
>
> Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. We call G and G' *isomorphic*, and write $G \simeq G'$, if there exists a bijection $\varphi : V \to V'$ with $xy \in E \Leftrightarrow \varphi(x)\varphi(y) \in E'$ for all $x, y \in V$. Such a map φ is called an *isomorphism*; if $G = G'$, it is called an *automorphism*. [...]
>
> A class of graphs that is closed under isomorphism is called a *graph property*. [...] 'containing a triangle' is a graph property: if G contains three pairwise adjacent vertrices then so does every graph isomorphic to G. A map taking graphs as arguments is called a *graph invariant* if it assigns equal values to isomorphic graphs. [...]
>
> We set $G \cup G' := (V \cup V', E \cup E')$ and $G \cap G' := (V \cap V', E \cap E')$. If $G \cap G' = \emptyset$, then G and G' are *disjoint*. If $V \subseteq V$ and $E \subseteq E'$, then G' is a *subgraph* of G (and G a *supergraph* of G'), written as $G' \subseteq G$. [...] we say that G *contains* G'. If $G' \subseteq G$ and $G' \neq G$, then G' is a *proper subgraph* of G. [...]

C.2.2 Encoding in Plain Syntax

```
(* Reinhard Diestel – Graph Theory *)
(* Original Author: Reinhard Diestel *)
(* Author Encoding in MathLang–Core: Paul van Tilburg *)

(** Preamble **)
{
  (* Constructors *)
  Set(dec('a), term) : set;
  Set'(dec('a), stat) : set;

  (* Binders *)
  forall(dec('i), stat) : stat;
  exists(dec('i), stat) : stat;
  iota(dec('x), term) : term;
  Union(dec('s), set) : term;

  (* Nouns, terms, sets *)
  natural_number : noun;
  real_number : noun;
  integer : noun;

  empty_set : set;

  0 : term;
  1 : term;
  2 : term;
```

```
    3 : term;
    e : term;

    (* Functions/identifiers *)
    and(stat, stat) : stat;
    disjoint(set, set) : stat;
    eq(term, term) : stat;
    eq_set(set, set) : stat;
    element_of(term, term) : term;
    geq(term, term) : stat;
    iff(term, term) : stat;
    in(term, set) : stat;
    impl(stat, stat) : stat;
    intersect(set, set) : set;
    leq(term, term) : stat;
    not(stat) : stat;
    nr_of_elements_of(set) : term;
    or(stat, stat) : stat;
    refer_to(set, set) : stat;
    range(term, term) : set;
    subset(set, set) : stat;
    superset(set, set) : stat;
    union(set, set) : set;
};

(** Chapter 1: The Basics **)
N : set(natural_number);
0 : N;
Z : set(integer);

n : natural_number |>
  { Z_mod(n) := Set(z : Z, mod(z, n));
    (* MISSING: notation of overline(i) := i + nZ *) };

x : real_number |>
  { floor(x) :=
      iota(i : integer, and(leq(i, x),
                            forall(j : integer, impl(leq(j, x), leq(j, i)))));
    ceil(x) :=
      iota(i : integer, and(geq(i, x)
                            forall(j : integer, impl(geq(j, x), geq(j, i)))));
    log_with_base(term, term) : term;
    log(x) := log_with_base(2, x);
    ln(x) := log_with_base(e, x); };

{ A : set; A_(term) : set; k : natural_number;
  forall(i : range(1, k),
    and(subset(A_(i), A),
        forall(j : range(1, k), disjoint(A_(i), A_(j))))); } |>
  { part_A := Set(i : range(1, k), A_(i));
    { B : set; B_(term) : set;
      forall(i : range(1, k),
        and(subset(B_(i), B),
            forall(j : range(1, k), disjoint(B_(i), B_(j))))); } |>
```

```
      { partition_of(B, A) :=
          and(eq(A, Union(i : range(1, k), B_(i))),
              forall(i : range(1, k), not(eq(B_(i), empty_set)))); };
84    partition_of(part_A, A);

      { A' : set; A'_(term) : set; } |>
        { part_A' := Set(i : range(1, k) A'_(i));
          refines(part_A', part_A) :=
89          forall(i : range(1, k),
              exists(j : range(1, k), subset(A'_(i), A_(j)))); };

      k_set_of_subsets(k, A) :=
        Set'(A' : set, and(subset(A', A), eq(nr_of_elements_of(A'), k)));
94    k_set(k, A) := Noun { eq(nr_of_elements_of(A), k); }
      k_set_of_subsets <- k_set;
    };

  (** Section 1.1: Graphs **)
99  { V' : set; E' : set; } |>
    graph(V', E') :=
      Noun { V := V';
             E := E';
             subset(self.E, k_set_of_subsets(2, self.V));
104          eq_set(intersection(self.V, self.E), empty_set); };

  { V : set; E : set; G : graph(V, E); } |>
    { vertex : noun;
      node := vertex;
109   point := vertex;
      v : G.V |> v : vertex;

      edge : noun;
      line := edge;
114   e : G.E |> e : edge;

      on(V, G) := eq_set(V, G.V);
      vertex_set(G) := G.V;
      V'(G) := vertex_set(G);
119   edge_set(G) := G.E;
      E'(G) := edge_set(G);

      { W : set; F : set; H : graph(W, F); } |>
        {
124      W(term) : set;
         vertex_set(H) ~= V'(G);
         vertex_set(H) ~= not(W(G));
        };

129   v : vertex;
      e : edge;

      nr_of_vertices(G) := nr_of_elements(G.V);
      order(G) := nr_of_vertices(G);
134   nr_of_edges(G) := nr_of_elements(G.E);
```

```
finite := Adj(graph);
infinte := Adj(graph);
countable := Adj(graph);

empty := Adj(graph) { and(eq_set(super.V, empty_set),
                          eq_set(super.E, empty_set)); };
empty_graph := empty graph;

trival := Adj(graph) { or(eq(order(super), 0), eq(order(super), 1)); };

incident(v, e) := element_of(v, e);
at(e, v) := incident(v, e);

{ v1 : vertex; v2 : vertex;
  incident(v1, e); incident(v2, e); not(eq(v1, v2)); } |>
  { endvertex(term, term) : stat;
    joins(term, term, term) : stat;

    end(v, e) := endvertex(v, e);
    endvertex(v1, e);
    endvertex(v2, e);
    joins(e, v1, v2); };

(* MISSING: notation of xy, X–Y edge, E(X, Y), E(v) *)
{ x : vertex; y : vertex; } |>
   ed(x, y) := Noun { v1 := x; v2 := y; };

{ x : G.V; y : G.V; } |>
  { adjacent_v(x, y) := element_of(ed(x, y), G.E);
    neighbours(x, y) := adjacent_v(x, y);
  };

{ e : G.E; f : G.E; not(eq(e, f)); } |>
  adjacent_e(e, f) := exists(w : vertex, and(end(w, e), end(w, f)));

complete(G) :=
  Adj(graph) { forall(v1 : super.V,
                 forall(v2 : super.V, adjacent_v(v1, v2))); };

K_(n) := Noun { G : complete graph; eq(nr_of_elements(G.V), n); };
triangle := K_(3);

{ v1 : vertex; v2 : vertex; } |>
  independant_v(v1, v2) := not(adjecent_v(v1, v2));

{ e1: edge; e2: edge; } |>
  independant_e(v1, v2) := not(adjecent_e(e1, e2));

independant_set_v(V) :=
  forall(v1 : V,
    forall(v2 : V, not(adjacent_v(v1, v2))));
stable_v(V) := independant_set_v(V);
```

```
189     independant_set_e(E) :=
          forall(e1 : E,
            forall(e2 : E, not(adjacent_e(e1, e2))));

        bijection(term, term) : noun;
194     { V' : set; E' : set; G' : graph(V', E'); phi : bijection(V, V'); } |>
          { isomorphism(G, G') :=
            Noun { forall(x : G.V,
                     forall(y : G.V,
                       iff(in(ed(x, y), G.E),
199                        in(ed(phi(x), phi(y)), G'.E)))); };

            exists(phi : term, phi : isomorphism(G, G')) |>
              isomorphic(G, G'); };

204     { phi : bijection(V, V); } |>
          automorphism(G) :=
            Noun { forall(x : G.V,
                     forall(y : G.V,
                       iff(in(ed(x, y), G.E),
209                        in(ed(phi(x), phi(y)), G.E)))); };

        graph_property :=
          Noun { class_of_graphs : set(graph(V, E));
                 exists(phi : isomorphism,
214                 forall(g_1 : class_of_graphs,
                      forall(g_2 : class_of_graphs, isomorphic(g_1, g_2)))); };

        pairwise_adjacent(term, term, term) : stat;
        containing_a_triangle : graph_property;
219     containing_a_triangle.class_of_graph :=
          Set'(G : graph(V, E),
            eq(nr_elements_of(Set'(v1 : G.V,
                                   Set'(v2 : G.V,
                                     Set'(v3 : G.V,
224                                    pairwise_adjacent(v1, v2, v3))))), 3));

        graph_invariant :=
          Noun { map(term) : term; (* map(graph) : term *)
                 forall(G : graph(V, E),
229                 forall(G' : graph(V', E'),
                      and(isomorphic(G, G'), eq(map(G), map(G'))))); };

        { V' : set; E' : set; G' : graph(V', E'); } |>
          { union_gr(G, G') := graph(union(G.V, G'.V), union(G.E, G'.E));
234         intersect_gr(G, G') := graph(intersect(G.V, G'.V), intersect(G.E, G'.E));
            disjoint(G, G') := eq(intersect_gr(G, G'), empty_graph);
            subgraph(G', G) := and(subset(G'.V, G.V), subset(G'.E, G.E));
            supergraph(G, G') := subgraph(G', G);
            contains(G, G') := subgraph(G', G);
239         proper_subgraph(G', G) := and(subgraph(G', G), not(eq(G, G')));
          };
      };
```

Appendix D

Implementation Output Documents

This appendix contains the output documents generated by the printers described in Section 3.5.

D.1 Plain Document

```
{
  0 : term;
  s(term) : term;
  x : term;
  y : term;
  a(x, 0) ~= x;
  a(x, s(y)) ~= s(a(x, y));
};
```

D.2 XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mathlang SYSTEM "http://www.macs.hw.ac.uk/~paulvt/mathlang/mlc23.dtd">
<mathlang>
 <step>
  <step>
   <dec>
    <id> <name>0</name> </id>
    <cterm/>
   </dec>
  </step>
  <step>
   <dec>
```

```
      <id> <name>s</name> </id> <cterm/>
      <cterm/>
     </dec>
    </step>
    <step>
     <dec>
      <id> <name>x</name> </id>
      <cterm/>
     </dec>
    </step>
    <step>
     <dec>
      <id> <name>y</name> </id>
      <cterm/>
     </dec>
    </step>
    <defcase>
     <id> <name>a</name> </id>
     <inst>
      <cid> <name>x</name> </cid>
     </inst>
     <inst>
      <cid> <name>0</name> </cid>
     </inst>
     <inst>
      <cid> <name>x</name> </cid>
     </inst>
    </defcase>
    <defcase>
     <id> <name>a</name> </id>
     <inst>
      <cid> <name>x</name> </cid>
     </inst>
     <inst>
      <cid> <name>s</name> </cid>
      <inst>
       <cid> <name>y</name> </cid>
      </inst>
     </inst>
     <inst>
      <cid> <name>s</name> </cid>
      <inst>
       <cid> <name>a</name> </cid>
       <inst>
        <cid> <name>x</name> </cid>
       </inst>
       <inst>
        <cid> <name>y</name> </cid>
       </inst>
      </inst>
     </inst>
    </defcase>
   </step>
  </mathlang>
```

D.3 ASCII Tree

D.3.1 Untyped ASCII Tree

```
Document
 `-- Block
     |-- Phrase
     |   `-- Dec
     |       |-- Id
     |       |   `-- "0"
     |       `-- Cterm
     |-- Phrase
     |   `-- Dec
     |       |-- Id
     |       |   `-- "s"
     |       |-- Cterm
     |       `-- Cterm
     |-- Phrase
     |   `-- Dec
     |       |-- Id
     |       |   `-- "x"
     |       `-- Cterm
     |-- Phrase
     |   `-- Dec
     |       |-- Id
     |       |   `-- "y"
     |       `-- Cterm
     |-- DefCase
     |   |-- Id
     |   |   `-- "a"
     |   |-- Inst
     |   |   `-- CId
     |   |       `-- "x"
     |   |-- Inst
     |   |   `-- CId
     |   |       `-- "0"
     |   `-- Inst
     |       `-- CId
     |           `-- "x"
     `-- DefCase
         |-- Id
         |   `-- "a"
         |-- Inst
         |   `-- CId
         |       `-- "x"
         |-- Inst
         |   |-- CId
         |   |   `-- "s"
         |   `-- Inst
         |       `-- CId
         |           `-- "y"
         `-- Inst
             |-- CId
```

```
            |   `-- "s"
            `-- Inst
                |-- CId
                |   `-- "a"
                |-- Inst
                |   `-- CId
                |       `-- "x"
                `-- Inst
                    `-- CId
                        `-- "y"
```

D.3.2 Typed ASCII Tree

```
Document
 `-- Block : Step
     |-- Phrase : Step
     |   `-- Dec : Dec (() -> Term)
     |       |-- Id : () -> Term
     |       |   `-- "0"
     |       `-- Cterm : Term
     |-- Phrase : Step
     |   `-- Dec : Dec ((Term) -> Term)
     |       |-- Id : (Term) -> Term
     |       |   `-- "s"
     |       |-- Cterm : Term
     |       `-- Cterm : Term
     |-- Phrase : Step
     |   `-- Dec : Dec (() -> Term)
     |       |-- Id : () -> Term
     |       |   `-- "x"
     |       `-- Cterm : Term
     |-- Phrase : Step
     |   `-- Dec : Dec (() -> Term)
     |       |-- Id : () -> Term
     |       |   `-- "y"
     |       `-- Cterm : Term
     |-- DefCase : Def ((Term, Term) -> Term)
     |   |-- Id : (Term, Term) -> Term
     |   |   `-- "a"
     |   |-- Inst : Term
     |   |   `-- CId : () -> Term
     |   |       `-- "x"
     |   |-- Inst : Term
     |   |   `-- CId : () -> Term
     |   |       `-- "0"
     |   `-- Inst : Term
     |       `-- CId : () -> Term
     |           `-- "x"
     `-- DefCase : Def ((Term, Term) -> Term)
         |-- Id : (Term, Term) -> Term
         |   `-- "a"
         |-- Inst : Term
```

```
|   `-- CId : () -> Term
|       `-- "x"
|-- Inst : Term
|   |-- CId : (Term) -> Term
|   |   `-- "s"
|   `-- Inst : Term
|       `-- CId : () -> Term
|           `-- "y"
`-- Inst : Term
    |-- CId : (Term) -> Term
    |   `-- "s"
    `-- Inst : Term
        |-- CId : (Term, Term) -> Term
        |   `-- "a"
        |-- Inst : Term
        |   `-- CId : () -> Term
        |       `-- "x"
        `-- Inst : Term
            `-- CId : () -> Term
                `-- "y"
```

D.4 Graphical Tree

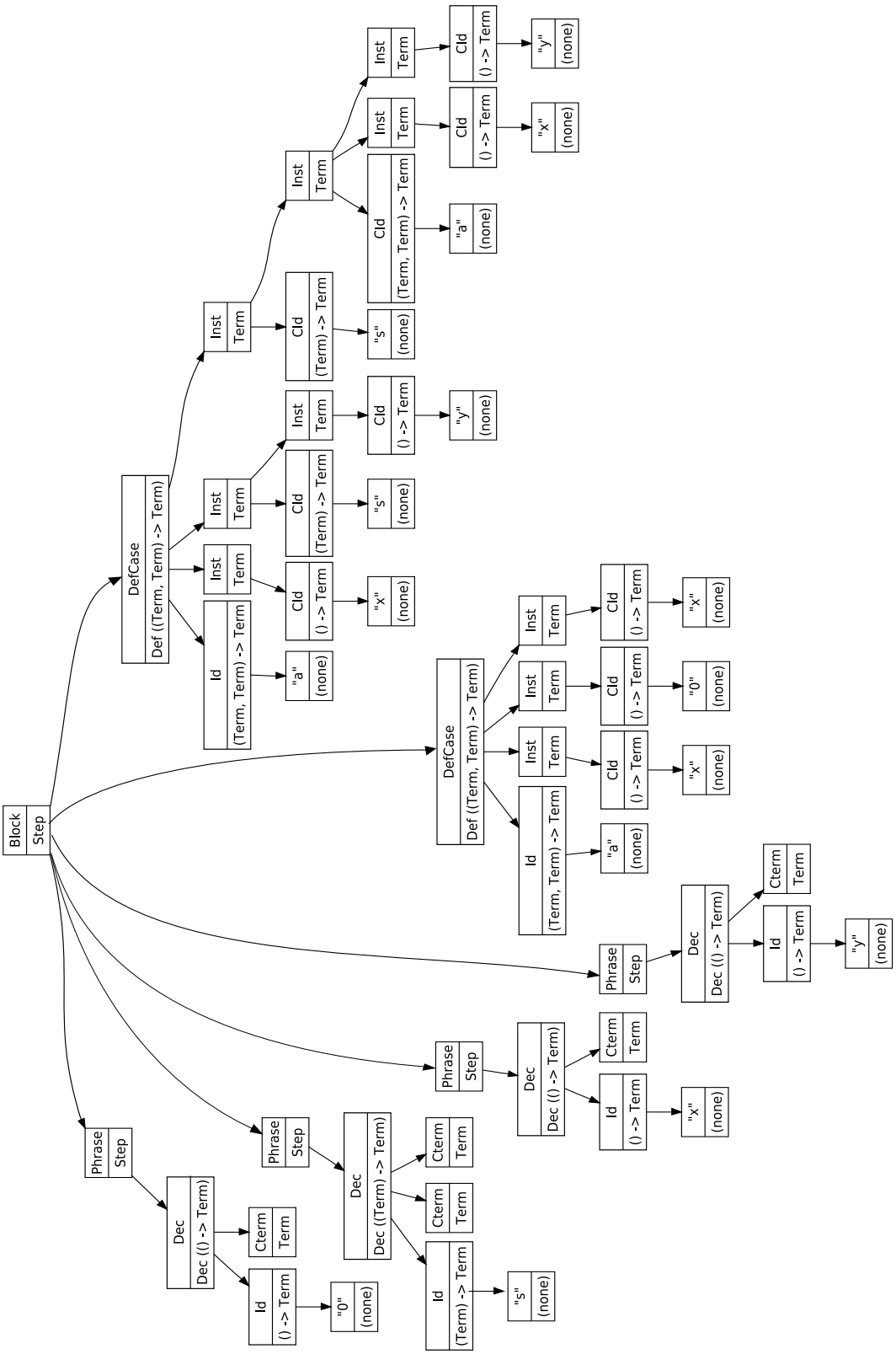
D.4.1 DOT File

```
digraph mathlang {
  n57 [shape=record, label="{ Block | Step }"];
  n4 [shape=record, label="{ Phrase | Step }"];
  n3 [shape=record, label="{ Dec | Dec (() -\> Term) }"];
  n1 [shape=record, label="{ Id | () -\> Term }"];
  n0 [shape=record, label="{ \"0\" | (none) }"];
  n1 -> { n0 };

  n2 [shape=record, label="{ Cterm | Term }"];
  n3 -> { n1 n2 };
  n4 -> { n3 };

  n10 [shape=record, label="{ Phrase | Step }"];
  n9 [shape=record, label="{ Dec | Dec ((Term) -\> Term) }"];
  n6 [shape=record, label="{ Id | (Term) -\> Term }"];
  n5 [shape=record, label="{ \"s\" | (none) }"];
  n6 -> { n5 };

  n7 [shape=record, label="{ Cterm | Term }"];

  n8 [shape=record, label="{ Cterm | Term }"];
  n9 -> { n6 n7 n8 };
  n10 -> { n9 };

  n15 [shape=record, label="{ Phrase | Step }"];
  n14 [shape=record, label="{ Dec | Dec (() -\> Term) }"];
```

```
    n12 [shape=record, label="{ Id | () -\> Term }"];
    n11 [shape=record, label="{ \"x\" | (none) }"];
    n12 -> { n11 };

    n13 [shape=record, label="{ Cterm | Term }"];
    n14 -> { n12 n13 };
    n15 -> { n14 };

    n20 [shape=record, label="{ Phrase | Step }"];
    n19 [shape=record, label="{ Dec | Dec (() -\> Term) }"];
    n17 [shape=record, label="{ Id | () -\> Term }"];
    n16 [shape=record, label="{ \"y\" | (none) }"];
    n17 -> { n16 };

    n18 [shape=record, label="{ Cterm | Term }"];
    n19 -> { n17 n18 };
    n20 -> { n19 };

    n32 [shape=record, label="{ DefCase | Def ((Term, Term) -\> Term) }"];
    n22 [shape=record, label="{ Id | (Term, Term) -\> Term }"];
    n21 [shape=record, label="{ \"a\" | (none) }"];
    n22 -> { n21 };

    n25 [shape=record, label="{ Inst | Term }"];
    n24 [shape=record, label="{ CId | () -\> Term }"];
    n23 [shape=record, label="{ \"x\" | (none) }"];
    n24 -> { n23 };
    n25 -> { n24 };

    n28 [shape=record, label="{ Inst | Term }"];
    n27 [shape=record, label="{ CId | () -\> Term }"];
    n26 [shape=record, label="{ \"0\" | (none) }"];
    n27 -> { n26 };
    n28 -> { n27 };

    n31 [shape=record, label="{ Inst | Term }"];
    n30 [shape=record, label="{ CId | () -\> Term }"];
    n29 [shape=record, label="{ \"x\" | (none) }"];
    n30 -> { n29 };
    n31 -> { n30 };
    n32 -> { n22 n25 n28 n31 };

    n56 [shape=record, label="{ DefCase | Def ((Term, Term) -\> Term) }"];
    n34 [shape=record, label="{ Id | (Term, Term) -\> Term }"];
    n33 [shape=record, label="{ \"a\" | (none) }"];
    n34 -> { n33 };

    n37 [shape=record, label="{ Inst | Term }"];
    n36 [shape=record, label="{ CId | () -\> Term }"];
    n35 [shape=record, label="{ \"x\" | (none) }"];
    n36 -> { n35 };
    n37 -> { n36 };

    n43 [shape=record, label="{ Inst | Term }"];
```

```
    n39 [shape=record, label="{ CId | (Term) -\> Term }"];
    n38 [shape=record, label="{ \"s\" | (none) }"];
    n39 -> { n38 };

    n42 [shape=record, label="{ Inst | Term }"];
    n41 [shape=record, label="{ CId | () -\> Term }"];
    n40 [shape=record, label="{ \"y\" | (none) }"];
    n41 -> { n40 };
    n42 -> { n41 };
    n43 -> { n39 n42 };

    n55 [shape=record, label="{ Inst | Term }"];
    n45 [shape=record, label="{ CId | (Term) -\> Term }"];
    n44 [shape=record, label="{ \"s\" | (none) }"];
    n45 -> { n44 };

    n54 [shape=record, label="{ Inst | Term }"];
    n47 [shape=record, label="{ CId | (Term, Term) -\> Term }"];
    n46 [shape=record, label="{ \"a\" | (none) }"];
    n47 -> { n46 };

    n50 [shape=record, label="{ Inst | Term }"];
    n49 [shape=record, label="{ CId | () -\> Term }"];
    n48 [shape=record, label="{ \"x\" | (none) }"];
    n49 -> { n48 };
    n50 -> { n49 };

    n53 [shape=record, label="{ Inst | Term }"];
    n52 [shape=record, label="{ CId | () -\> Term }"];
    n51 [shape=record, label="{ \"y\" | (none) }"];
    n52 -> { n51 };
    n53 -> { n52 };
    n54 -> { n47 n50 n53 };
    n55 -> { n45 n54 };
    n56 -> { n34 n37 n43 n55 };
    n57 -> { n4 n10 n15 n20 n32 n56 };

}
```

D.4.2 PostScript Rendering



Bibliography

[1] *Twenty-five years of Automath research*, 1994. R.P. Nederpelt and J.H. Geuvers and R.C. de Vrijer.

[2] *N.G. de Bruijn*. The Mathematical Vernacular, a language for mathematics with typed sets. In *Workshop on Programming Logic*. 1987. In [1].

[3] *Daniel de Rauglaudre. Camlp4 – Reference Manual*. `http://caml.inria.fr/pub/docs/manual-camlp4/`. Version 3.07.

[4] *Reinhard Diestel. Graph Theory*. Springer-Verlag, Heidelberg, third edition edition, 2005.

[5] *ULTRA Group*. The Abstract Syntax Tree of MathLang-Core. Version ML-C 2.3.0.

[6] *ULTRA Group*. Explanation of the Plain Syntax of MathLang-Core. Version ML-C 2.3.0.

[7] *ULTRA Group*. The MathLang Manual. Version ML-C 2.3.

[8] *ULTRA Group*. The Plain Syntax of MathLang-Core. Version ML-C 2.3.0.

[9] *ULTRA Group*. The Plain Syntax Parser of Mathlang-Core. Version ML-C 2.3.0.

[10] *Heath. The 13 Books of Euclid's Elements*. Dover, 1956.

[11] *F. Kamareddine and R. Nederpelt*. A Refinement of de Bruijn's Formal Language of Mathematics. *Journal of Logic, Language and Information*, 13(3):287–340, 2004.

[12] *Fairouz Kamareddine, Manuel Maarek, and J. B. Wells*. Flexible Encoding of Mathematics on the Computer. In *Mathematical Knowledge Management, 3rd Int'l Conf., Proceedings*, volume 3119 of *LNCS*, pages 160–174. Springer-Verlag, 2004. ISBN 3-540-23029-7.

[13] *Fairouz Kamareddine, Manuel Maarek, and J. B. Wells*. MathLang: Experience-Driven Development of a New Mathematical Language. In *Proc. [MKMNET] Mathematical Knowledge Management Symposium*, volume 93 of *ENTCS*, pages 138–160. Elsevier Science, Edinburgh, UK (2003-11-25/---29), February 2004.

[14] *Fairouz Kamareddine, Manuel Maarek, and J. B. Wells.* Toward an Object-Oriented Structure for Mathematical Text. In *Mathematical Knowledge Management, 4th Int'l Conf., Proceedings*, Lecture Notes in Artificial Intelligence, pages 217–233. Springer-Verlag, 2006. ISBN 3-540-31430-X.

[15] *Graphviz Team. The DOT Language.* `http://www.graphviz.org/pub/scm/graphviz2/doc/info/lang.html`.

[16] *World Wide Web Consortium (W3C). Document Object Model (DOM).* `http://www.w3.org/DOM/`.

[17] *World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Third Edition).* `http://www.w3.org/TR/REC-xml`.

[18] *F. Wiedijk.* Formal Proof Sketches. In *Proceedings of TYPES'03*, volume 3085 of *LNCS*, pages 378–393. Springer-Verlag, December 2004.

[19] *Xavier Leroy (with Damien Doligez, Jacques Garrigue, Didier Rémy and Jéôrme Voullion). The Objective Caml system – Documentation and user's manual.* `http://caml.inria.fr/pub/docs/manual-ocaml/`. Release 3.09.