Mathematical documents faithfully computerised: the grammatical and text & symbol aspects of the MathLang framework

Manuel Maarek

Submitted for the degree of Doctor of Philosophy

Heriot-Watt University

School of Mathematical and Computer Sciences

June 2007

Abstract

A comprehensible computerisation of mathematical texts requires jointly to reflect the mathematician's phrasing and to put his thoughts in a formalised shape. The MathLang project faces these antagonist goals by decomposing the computerisation/formalisation process by means of discerned aspects. This thesis is concerned with two of these aspects which are, firstly the definition of a formal grammar for informal mathematical argumentation and secondly the elaboration of an authoring and encoding method linking the mathematician's own phrasings with unequivocal explanations. We have developed these two MathLang aspects to a prototype stage where texts and symbols are related to checkable grammatical constructs. We have experimented with our system the edition of texts from the mathematical literature.

Acknowledgments

I would first like to thank Fairouz Kamareddine and Joe Wells who were kind enough to supervise my work. I am grateful for their encouragement, support, foresight and, above all, their confidence in me. I would also like to thank Thérèse Hardin, who introduced me to them, and who, along with Jim Howie, reviewed my thesis. I appreciate the help of the members of the MathLang project who contributed to this thesis: Fairouz Kamareddine, Joe Wells, Krzysztof Retel, Paul van Tilburg and Robert Lamar, as well as the students who first used the software we developed: Hamish A. Ross, M. Isabel Lopez Fernandez, Mona Petrie, Amalia Retzepi, Anastasios Tsaousis, Jing He and Anastasios Asimakopoulos. Special thanks to Alasdair J.G. Gray and to my other colleagues and friends from the ULTRA group, the MACS School and Heriot-Watt University for making our workspace so comfortable and welcoming. I would also like to thank the InfoLibre association who gave me the opportunity to be a user of free software, and to my former colleagues Renaud Rioboo and Virgile Prevosto for the many things I learned from them. I would also like to express my gratitude to the members of the Ωmega group who kindly invited me to Saarbrücken three times.

L'enthousiasme de mon oncle Marcel et le temps qu'il dédia aux jeunes étudiants que nous étions Louis et moi-même sont à l'origine de mon intérêt pour les mathématiques et leurs représentations. Je l'en remercie. Voici un extrait d'un texte qu'il écrivit en 1985 et qui est remarquablement en lien avec le sujet de ma thèse.

> «La mathématique formelle constitue une sorte d'horizon de la mathématique réelle ». Construction sous-jacente, jamais écrite, mais que l'on pourrait toujours produire en cas de contestation. *[Maa85]*

Je remercie Sarah, Françoise, Gérard, Hector, Gabriel, Élie, Jeanne, Kurt, et toute ma famille pour leur soutien. Je remercie particulièrement Nataša pour sa joie et le temps partagé. I am thankful to the Cheesee-Peasee regulars, to the 20s participants, to Mão No Chão's angoleiros, and finally, to my friends from Edinburgh, Paris and elsewhere. Je remercie les adeptes des samedis Cheesee-Peasee, les participants des 20s, les angoleiros de Mão No Chão, et enfin les amis d'Édimbourg, de Paris et d'ailleurs.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Mathematics has always played a central role in the body of human knowledge; used as a conceptualising medium and a computation tool in many sciences, it appeals philosophical reflection on the insight of reasoning. Mathematics is perceived as a pure and aesthetic expression of human thoughts. The mathematics we use today is the result of millennia of refinements. The identification of mathematics, as occurred in ancient times, was characterised by early steps into mathematical abstraction and the use of rigorous discourse. The question of its dating is of minor importance compared to its definiens. Mathematics is the use of a meta-language for describing quantities and the arrangement of things. The early advancements in mathematics were the result of, for example, the symbolisation of absence by a symbol “0” of Hindu-Arabic origins [Ifr99], or the enunciation of a treatise by Euclid [Hea56] establishing the general properties of geometrical figures. The development of a language support for mental constructs is certainly a founding aspect of mathematics. As time has passed, so mathematicians and philosophers have clarified the relationship between mathematical thoughts and their textual and symbolic representations.

> By analysis of the mechanism of proofs in suitably chosen mathematical texts, it has been possible to discern the structure underlying both vocabulary and syntax. This analysis has led to the conclusion that a sufficiently explicit mathematical text could be expressed in a conventional language containing only a small number of fixed “words”, assembled according to a syntax consisting of a small number of unbreakable rules: such a text is said to be *formalized.*
>
> *[Bou54, Bou68, Chapter I]*

The formalisation of mathematics was made possible by the definition of a rigorous and expressive communication medium. The isomorphism between written mathematics and mathematical thinking is therefore a prerequisite as G. Frege addressed.

> For brevity I have called a sentence true or false though it would certainly be more correct to say that the thought expressed in the sentence is true or false. *[Fre14]*

Later, the development of modern mathematics celebrated the expressive power of abstraction by defining new symbolic abstract representations.

More recently, mathematicians and scientists were granted new experimentation possibilities with the advent of powerful calculators. In the mean time, this new computational capability was also applied to other domains such as editing and publishing. In only a few decades, computers have changed the way we approach documents and, by extension, mathematical documents. The impact on the understanding of formal mathematics is tremendous and not yet measurable. From an endeavor half a century ago, formalisation has become today an assisted and feasible task. Computer-based formalisation and theorem proving became mandatory skills in every logicians' curriculum. But this formal approach of mathematics by computers comes with a high price in work-time. Even if the result is of an unquestionable truthfulness because it has been computer-checked, many mathematicians are reluctant to exchange their traditional "we can see that" for lines of proof scripts. Here is N.G. de Bruijn's view.

> Many people like to think that what really matters in mathematics is a formal system (usually embodying predicate calculus and Zermelo-Fraenkel set theory), and that everything else is loose informal talk *about* that system. Yet the current formal systems do not adequately describe how people actually think, and, moreover, do not quite match the goals we have in mathematical education. Therefore it is attractive to try to put a substantial part of mathematical vernacular into the formal system. One can even try to discard the formal system altogether, making the vernacular so precise that its linguistic rules are sufficiently sound as a basis for mathematics. *[dB87, §1.4]*

This failure of the theorem-proving community to win the mathematicians to their systems initiated a new era of computer-assisted mathematics in the mid 90s. The

QED manifesto [CAD94] once described this aspiration to “help mathematicians cope with the explosion in mathematical knowledge”.

We discuss in this thesis the consequences of computer-based formalisation for mathematical authoring habits and we present our approach, within the MathLang project, for computerising mathematical texts. The MathLang project, initiated in 2000 by F. Kamareddine and J.B. Wells [KW00, KW01], is an ongoing project of the ULTRA group[1] with the aim of “computerising mathematics in a manner which keeps computerization as close as possible to the mathematician’s text while at the same time providing a formal structure supporting mathematical software systems” using J.B. Wells’s own words. MathLang’s proposal to achieve this aim is to divide the computerisation into aspects. This decomposition approach into levels was first enunciated in 1987 by N.G. de Bruijn in his Mathematical Vernacular [dB87] and is an inspiring work for the MathLang project.

This thesis is concerned with the definition, implementation and experimentation of two aspects of the MathLang framework. The first aspect, which we have named the Core Grammatical aspect (CGa), involves a formal abstract syntax and grammar for valid mathematical justifications. The second aspect, which we have named the Text & Symbol aspect (TSa), involves an authoring method and language for relating mathematical texts with their explicated semantic role.

1.1 Contributions

We summarise the contributions of this thesis in the following points.

- MathLang’s Core Grammatical aspect (CGa). A language that captures the structure of mathematical reasonings without committing itself to any logic or semantic foundation. This aspect consists of:
  - A formal, concise object-oriented abstract syntax,
  - A type system giving full benefits from encoding efforts,
  - A software implementation based on well-known standards, and
  - A set of encodings of literary texts as test cases for the language’s expressiveness.

[1] ULTRA is an acronym for Useful Logics, Types, Rewriting, and their Automation. ULTRA (`http://www.macs.hw.ac.uk/ultra/`) is a research group in the School of Mathematical and Computer Sciences (MACS) at Heriot-Watt University.

- MathLang's Text & Symbol aspect (TSa). This is our solution for restoring natural language as a computerised input method for mathematics. It consists of:
  - An authoring method based on the annotation of raw natural language mathematical texts,
  - A set of generic annotations, which we call *syntax souring* in contrast to the usual *syntax sugaring*, to explicate the morphology of typical abbreviations and aggregations in mathematical texts,
  - A set of rewriting rules formally defining the souring transformations,
  - A mathematician-oriented implementation of this method within a user-friendly scientific editor, and
  - A set of annotated texts showing the extent to which this method is applicable.
- An analysis of how the preponderant role MathLang's aspect-oriented computerisation of mathematical texts could play a role in winning the mathematical community back to computer-based formalisation.

Publications and collaborators

Different elements of these contributions have been published in proceedings of events organised by the Mathematical Knowledge Management (MKM) community. Following is a summary of these publications related to this thesis, co-authored with F. Kamareddine, J.B. Wells, K. Retel and R. Lamar. We list in this summary the sections of this thesis related to each publication.

- [KMW04b] — In 2003, we presented at the MKM Symposium the initial language of the CGa aspect of MathLang (i.e. called MWTT[2] in this thesis) which we tested on a whole textbook chapter. Partially presented in Sections 3.4.1 and 4.4.
- [KMW04a] — In 2004, we presented at the Third MKM Conference our initial technique for associating meaning and representation aspects of mathematical texts. Partly reflected in Sections 3.4.2 and 6.3.2

[2] MWTT is an acronym for MathLang-Weak Type Theory (WTT); MWTT is an extension of WTT (see Section 2.2) towards what became MathLang-CGa.

- [KMW06] — In 2005, we presented at the Fourth MKM Conference our object-oriented approach to describing the structure of mathematical texts. An extended version is included in Sections 3.4.3, 4.1, 4.2 and 4.3.
- [KLMW07] — In 2007 we will be presenting at the Sixth MKM Conference our genuine approach to restoring natural language as a computerised mathematics input method. An extended version is included in Sections 5.1, 5.2 and 6.4.

The work presented in this thesis has gained directly and indirectly from comments, inputs and collaborations with members of the MathLang project (F. Kamareddine, J.B. Wells, K. Retel, P. van Tilburg, R. Lamar), undergraduate and postgraduate students at Heriot-Watt University (H.A. Ross, M.I. Lopez Fernandez, M. Petrie, A. Retzepi, A. Tsaousis, J. He and A. Asimakopoulos) and a fruitful collaboration[3] with the Ωmega group at the Universität des Saarlandes (particularly with S. Autexier, A. Fiedler, H. Lesourd and M. Wagner). Currently there are two further PhD students working on the MathLang project (K. Retel who is currently writing up his thesis and R. Lamar who has just completed his first year PhD studies). The research of K. Retel and R. Lamar involves the design, development and implementation of existing and new aspects of the MathLang framework.

In connection with these publications and collaborations, a number of presentations[4] were given to disseminate the MathLang's approach on computerising mathematics.

1.2 Outline

In Chapter 2, we begin with a presentation of the background on which this thesis is based including the related works in the field. In Chapter 3, we draw a picture of the fundamental original insights to the MathLang project and to this work. Chapter 4 contains the complete definition of MathLang-CGa and its type system. The MathLang-TSa authoring method is explained in Chapter 5 along with several examples. Finishing with the thesis' achievements, Chapter 6 describes the current CGa-TSa framework's implemented software. In Chapter 7, we envision the possible followup work based on this thesis.

[3] Funded by the British Council.

[4] Presentation materials available at http://www.macs.hw.ac.uk/~mm20/ and http://www.macs.hw.ac.uk/~fairouz/

Chapter 2

Background and Related Works

In this chapter we present the background of this thesis and the related work in the field of Mathematical Knowledge Management (MKM). Sections 2.1 and 2.2 present two languages on which we based part of our research, namely the Mathematical Vernacular (MV) and the Weak Type Theory (WTT). We then present in Section 2.3 the MathLang project around which the thesis took place. Section 2.4 draws a picture of the current situation of mathematics on computers.

2.1 De Bruijn's Mathematical Vernacular

The Mathematical Vernacular was presented by professor N.G. de Bruijn in [dB87] in 1987. In this article he defines a language which represents what should be a formal language for mathematics. Choosing the word *vernacular* was not innocuous by the leader of the Automath project. He ardently defines what he believes to be a formal definition of the mathematical *lingua franca*.

> The word "vernacular" means the native language of the people, in contrast to the official, or literary language (in older days in 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. *[dB87, §1.2]*

MV is both a summary of two decades of the Automath experience and a series of freshly conceived, disinterested intuitions and ideas for a comprehensible language for mathematics. To understand how the idea of MV grew up, it is first necessary to describe its background, the Automath project.

2.1.1 Automath

Automath is a language for formalising mathematical texts and for automating the validation of such formalised mathematics. The language was developed within the Automath project initiated by N.G. de Bruijn in the late sixties. The Automath project was the first attempt to digitize and formally prove mathematics with the assistance of a computer. The language Automath was developed by N.G. de Bruijn and his collaborators and its correctness was empirically proved by the formalisation of E. Landau's *Foundations of Analysis*. This formalisation of the entire book in Automath was performed by L.S. van Benthem Jutting in his PhD thesis [vBJ77a] in 1977.

This section is a brief introduction to Automath. This introduction is partial because it focuses on Automath's representation of mathematical knowledge and less on the formalisation of mathematics. Our main sources of references on Automath are [NGdV94, Kam03, Aut] from which we especially refer to [dB91a, dB70, dB80, vBJ94].

2.1.1.1 Language

Automath is a formal language (or more precisely a family of languages) for encoding mathematics on computers. A text encoded in any of the Automath languages can be automatically checked by a computer program. Each Automath language offers different extensions. Table 2.1 recalls these extensions and their implementations. A recent implementation of an Automath checker was based on the AUT-68 and AUT-QE languages. It has been implemented by F. Wiedijk[1].

2.1.1.1.1 Book content An Automath text is represented in the form of *books*. Each book is a sequence of *lines*. Each new line in a book could refer to elements contained in previous lines. Each line comes in a *context* which is composed by a set of *assumptions* and *variable* declarations from previous lines. These components are called *block openers*. This name gives a clear meaning to context's items which open the availability of a variable or an assumption. Nested block openers form

[1] See F. Wiedijk's web page dedicated to Automath: `http://www.cs.ru.nl/~freek/aut/` (last visited 2007-04-21).

AUT-68	The initial Automath language.	[vBJ94]
AUT-SL	Also named λ-Automath, it is equivalent to AUT-68 where book and lines are put into a single line (SL). Each definition is an abstraction. The context variables/arguments of a definition are transformed into a sequence of abstractions. Each head-expression (see Section 2.1.1.1.4) is transformed into an sequence of applications. AUT-SL is a curried normal formed of AUT-68.	[dB94b]
AUT-QE	The theory is defined in D.T. van Daalen's PhD thesis [vD80]. The implementation is used by L.S. van Benthem Jutting in his translation [vBJ77a] of E. Landau's *Foundations of Analysis* [Lan51]. AUT-QE is a sub-language of AUT-Π.	[vD80]
AUT-QE-NTI	NTI stands for "no type inclusion". As a result it is weaker than AUT-QE.	[dB94a]
AUT-Π	Invented by J. Zucker, it is an extension of AUT-QE with pairs, projections, injections and disjoint union type. Used in the formalisation of classical mathematics and to represent natural deduction.	[vD80]
AUT-4	The extension of AUT-QE with 4-expressions; `type` is the only 1-expression (see Section 2.1.1.2.1). Degrees of `prop`, propositions and proof are increased by one.	[dB94c]
AUT-SYNT	Extends AUT-QE with a variable `synt` and a set of predefined *meta*-functions such as a mechanical type-inferrer.	[vBJ77a]
AUT-$\Delta\Lambda$	Is a more theoretical version of AUT-SL which uses de Bruijn indices.	[dB78]

Table 2.1: List of Automath versions, their extensions and implementations

a context. A line either defines a new notion in the book or extends the current context.

2.1.1.1.2 Context In an Automath book, contexts have an interesting chain-like structure. Notions are defined one after the other but in different contexts. The contexts constantly get new links to create new notions and some times unchain some links to start over a new fresh context. An overall picture of the chaining and unchaining of contexts in a book could be seen as a tree of contexts, see Figure 2.1. This chaining of context elements is also called a *telescope* [dB91b].

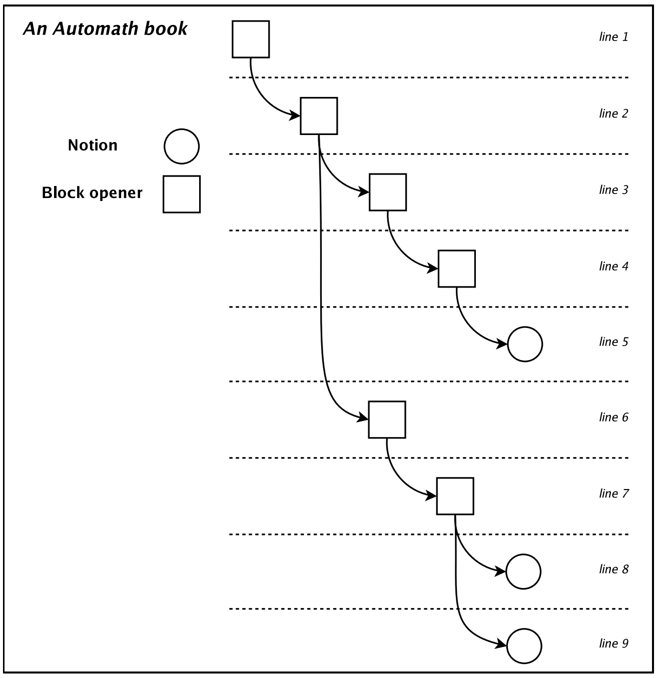


Figure 2.1: An Automath context tree

2.1.1.1.3 Line content Syntactically each line and block opener is defined by four distinct elements. Table 2.2 is an illustration of these four elements: indicator, identifier, definition and category. We give here a short description of each element.

Indicator informs on the context of the current entity to be defined. This indicator refers to a previously defined block opener *b*. If this indicated block opener had itself a context *c* (reference to another block opener), the entity we are currently concerned with will have as context *c* extended with the block opener *b*. Figure 2.1 gives a diagrammatic view of this chaining of context elements. If the indicator is set to 0 then the definition takes place in an empty context.

Identifier is a name for the assertion, variable, notion, type, set or symbol being defined. It should not be similar to any identifier already defined.

Definition is the actual value associated with the identifier. If the identifier to be defined is a variable, the definition is represented by the empty symbol "—" and the entity defined is a block opener. If the symbol to be defined does not

	ξ	:=	—	`type`	(1)
ξ	x	:=	—	ξ	(2)
ξ, x	y	:=	—	ξ	(3)
ξ, x, y	is	:=	PN	`type`	(4)
ξ, x	$reflex$	:=	PN	$is(x, x)$	(5)
ξ, x, y	$asp1$	:=	—	$is(x, y)$	(6)
$\xi, x, y, asp1$	$symm$	:=	PN	$is(y, x)$	(7)
$\xi, x, y, asp1$	z	:=	—	ξ	(8)
$\xi, x, y, asp1, z$	$asp2$	:=	—	$is(y, z)$	(9)
$\xi, x, y, asp1, z, asp2$	$trans$	:=	PN	$is(x, z)$	(10)

Example from [dB70] defining equality in an arbitrary category.

Table 2.2: Automath example of line elements

have a definition because it is part of the root of the theory enunciated in the book, then PN or `PRIM` – which stands for primary notion – is provided as definiens.

Category is the identifier's type. It is either `type` (and `prop` in AUT-QE) or an expression of category `type` (and `prop` in AUT-QE). This stratification of types is in line with the tradition of hierarchy of types of *Principia Mathematica* [WR13], the Simple Type Theory [Ram26, HA28] and the simple typed λ-calculus [Chu40].

This simple decomposition of a line into these four elements is impressively expressive. All sorts of reasoning steps can be represented by combining the modes of each element. For instance, a variable is an identifier without definition (the empty symbol – stands for its definition). The scope of a variable is delimited. The variable scopes between its declaration in a block opener (a line defining the variable) and the first following line that un-chains the block opener from its context (the indicator of this line refers to a block opener preceding the variable's block opener). In Figure 2.1 the block opener on line 6 un-chains the block opener of lines 3 and 4 and therefore closes the scope of the two variables they were respectively introducing.

2.1.1.1.4 Expressions The expressions that could stand for an identifier's definition or category could have one of the following four forms.

Variable expression. Variables that are declared in the context or that are declared by an abstraction are expressions.

Head expression. An identifier defined in a previous line of the book forms an expression if provided with the right number of parameters.

If a line of the form `[x][y][z] a(x,y,z):=...` is part of a book then the head-expression `a(A,B,C)` is a valid expression in the following lines (`A`, `B`, `C` being valid expressions). This example is taken from [vBJ94].

Abstraction expression. The abstraction of a variable in an expression where this variable may occur freely is an expression. The Automath syntax for abstraction is `[x]A` which is equivalent to the λ-calculus term $(\lambda x.A)$.

Remark 1 *According to this definition,* `[x]x` *is a valid expression. On the contrary, its sub-expression* `x` *is not. The definition in [vBJ94] states that if A is an expression in a context formed by* `[x]` *then* `[x]A` *is a valid expression.*

Application expression. The application of an expression to another is an expression. The Automath syntax for application is `<B> A` which is equivalent to the λ-calculus term $(A\ B)$.

An expression in Automath is therefore either a variable, an abstraction, an application or an instantiation of a constant (head expression).

2.1.1.2 Typing: checking the books' correctness

The Automath language and its extensions are defined with a type system that describes how to systematically check the well-formation of Automath books and the correctness of the demonstration in a book.

2.1.1.2.1 Degree of an expression The Automath type system associates to any identifier in a valid book a unique type expression. This type expression is itself a valid expression in the book. To avoid Russell's Paradox, each identifier is assigned a degree. A type expression should be of one degree less that the degree of the expression it is the type of. Therefore it is impossible to state the antinomic x is of type x.

There are three degrees of expressions:

- *3-expressions* are objects.
- *2-expressions* are types.
- *1-expression* is the super-type expression `type`.

Some Automath extensions (see table 2.1) adapt this hierarchy of expressions.

- AUT-QE adds a new super-type `prop`. Expressions of type `prop` are propositions and expressions with a proposition as type expression are proofs. This is the Automath version of the *propositions as type* principle.

- AUT-4, as its name suggests, introduces a 4th degree of expressions. This additional degree is actually a shifting of `prop`, propositions and proofs. In AUT-4 `prop` is not a super-type anymore. Propositions are expressions of degree 3 and proofs are the new 4-expressions. This new degree gives a different interpretation for proofs. In [dB94c] the definitional equality (see Section 2.1.1.2.2) between 4-expressions is considered as *irrelevant*. Effectively the fact that two proofs are or are not equal is irrelevant. The important equality is between what they prove. The relevance of definitional equality is therefore the main difference in the interpretations of 3- and 4-expressions. AUT-4 gives satisfaction to a classical view of proofs for which the relevance of a proof is in what it proves.

2.1.1.2.2 Reductions and definitional equality Automath has substitution and reduction operations in its *meta-theory*. Two reductions are used to obtain the normal form of an expression. This normal form is used to compare expressions and find out if they have the same "meaning". The definitional equality of two expressions is based on these two reductions. The reductions are defined as follow (see [vBJ94]):

δ-reduction This reduction expands or inlines a definition. If

$$[x_1] \dots [x_n]\ a(x_1, \dots, x_n) := E$$

is a line of the book and, on a certain context, A_1, ..., A_n are expressions, then we have

$$a(A_1, \dots, A_n) \rightarrow_\delta E[x_1, \dots, x_n := A_1, \dots, A_n]$$

where $E[x_1, \dots, x_n := A_1, \dots, A_n]$ denotes the expression E where x_1, ..., x_n are simultaneously replaced by A_1, ..., A_n.

β-reduction This reduction is the traditional λ-calculus' β-reduction which calculates the result of a function application.

$$< A > [x]B \rightarrow_{\beta} B[x := A]$$

if $< A > [x]B$ is an expression.

The relation $\rightarrow$ is the compatible closure of $\rightarrow_{\delta}$ and $\rightarrow_{\beta}$. The relation $\twoheadrightarrow$, called *reduction*, is the reflexive and transitive closure of $\rightarrow$. The *definitional equality* is the relation $\overset{D}{=}$, the smallest equivalence relation which contains $\twoheadrightarrow$.

2.1.1.2.3 Typing rules In [vBJ77a] a small set of basic typing rules defines the Automath type system. Automath typing is unique modulo definitional equality. We refer to D.T. van Daalen's rules from [KLN03] for the complete set of rules.

2.1.1.3 Perspectives offered by two decades of Automath

The Automath experience over two decades has played two important roles. One in the history of mathematics, as the first attempt to create a formal language for mathematics on the computer. And the other in computer science, as the first computer-based language for automatic proof checking. N.G. de Bruijn published in 1991 an article [dB91a] which is a mixture of a philosophical summary of the Automath experience and a list of perspectives offered by this experience. In this article, he presents what he calls *justification systems* (in which he includes Automath). He introduced this notion of justification when putting face to face automated checking and automated proving. The first one being the computer task of systematic logical checking of proofs and the other one being the ability to ask computers to *invent* a valid proof for a given theorem. He considered justification to be an automated checking that does not only deal with proofs but with mathematics in general. Formalisation has two means: correctness of proofs and better *understanding* of mathematics. N.G. de Bruijn proposed to clarify the actions one is doing when formalising mathematics. One should distinguish a formalism for mathematics and a computer based language for checking mathematical knowledge.

The main question that arises when one starts to distinguish proof checking and formalisation of mathematics is the interrogation over the trust that could be put on formalised mathematics. This interrogation existed also for proof checking and was answered in many ways. One is to provide a systematic definition of the logic and calculus implemented by such a system. Another consists in checking a proof generally considered difficult to be checked by a human (e.g. the proof of the four-colour theorem in Coq [Gon] or the proof of the Jordan Curve Theorem in Mizar [Kor05]).

> One of the first questions people ask when hearing about a justification system is whether it would guarantee absolute dependability. This can mean two things. In the first place there is the matter of absolute dependability of mathematics, whatever the foundations may be. I think there is little hope ever to get final answers to that question.
>
> The second thing it can mean is this: once we have accepted a rigorous formalization of some piece of mathematics, and we have accepted the idea that "mechanical" verification gives a kind of absolute guarantee of correctness, we ask whether this guarantee would be weakened by leaving the mechanical verification to a machine. This is a very reasonable, relevant and important question. It is related to proving the correctness of fairly extensive computer programs, and checking the interpretation of the specifications of those programs.*[...]* *[dB91a]*

This above quotation raises an interesting question. How should we consider formalised text? N.G. de Bruijn discards here the ideal of an absolute foundation for mathematics. N. Bourbaki[2] in the *Elements of Mathematics* [Bou39] proposed a systematic approach to describing mathematics based on axiomatic set theory and claimed that:

> The verification of a formalized text is a more or less mechanical process *[...].* *[Bou54, Bou68, Chapter I]*

Nevertheless no tangible computer-based formalisation of the *Elements of Mathematics* has been accomplished or even attempted. This could have two causes and therefore two implications for computer-based foundations for mathematics:

Is N. Bourbaki's formalism not computerisable? If no logical framework can encode and check the *Elements of Mathematics* then how can we speak about a foundation for mathematics that discards such a contribution?

[2] N. Bourbaki is the pseudonym chosen by a group of French mathematicians (André Weil, Jean Dieudonné, Szolem Mandelbrot, Claude Chevalley, Henri Cartan were some of the founding members) in the 1930's. They wrote under this name a full treatise of modern mathematics: *Elements of Mathematics* [Bou39].

Is N. Bourbaki's formalism valid after all? If the actual formalism on which N. Bourbaki's *Elements of Mathematics* is based is proved to be invalid for any logical system then does it mean that the entire *Elements of Mathematics* should be banned from any incorporation in a formalised mathematical library?

N.G. de Bruijn proposes an answer that disambiguates the situation and clarifies ones actions when formalising mathematics. One should distinguish a formalism for mathematics from a computer based language for checking mathematical knowledge.

> The work may be subdivided. One can think of a first stage where a person with some mathematical training inserts a number of intermediate steps whenever he feels that further workers along the belt might have trouble, and a second stage where the logical inference rules are supplied and the actual coding is carried out. For the latter piece of work one might think of a person with just some elemenary mathematics training, or of a computer provided with some artificial intelligence. But we should not be too optimistic about that: programming such jobs is by no means trivial. *[dB91a]*

N.G. de Bruijn clearly differentiates two tasks in the work of formalising mathematics. The first one is close to traditional mathematical demonstration which identifies the elements that compose a proof. The second one is a systematic justification of every single logical step in a proof. This subdivision assumes that the foundation used in the formalisation and the chosen representation of mathematical entities are adequate for both steps. In practice, the systematic verification requires adaptation and, in the worst case, profound changes to a working proof to making it suitable for a specific formalism. Herein lies the difficulty of choosing a suitable logical foundation when formalising mathematics. N.G. de Bruijn takes the party of the "mathematicians' approach" which is to be keen on protecting liberties and choices even if this would discard a full formalisation. This approach is opposed to a "logicians' approach" for which any mathematics could be and should be sooner or later formalised in a universal logical framework.

> The language of mathematics is *not* talking about a limited number of things. If it were, a justification system might try to take a kind of model-theoretical approach by testing every statement in that world

> of objects. The language of mathematics cannot be verified on the objects: there are too many of them. The only thing we can do is applying the rule that things are correct if they have been correctly said. The notion of correctness is not formulated in terms of a mathematical reality, but involves rules about how a statement should be related to material that has been said before. Many non-mathematicians who hear about verification systems get the idea that such systems can handle only "constructive" situations like finite mathematics. This confusion depends on that wrong idea of implementing mathematical reality.
> *[dB91a]*

N.G. de Bruijn, being a mathematician inclined to make good use of computers, aims to reconcile the working-mathematician with the working-logician by raising the intrinsic question of the definition of a mathematical vernacular.

> I think that in formalizing mathematics, and in particular in preparing mathematics for justification, it is usually elegant as well as efficient to do everything in the *natural* way. That word of course does not mean "like in nature"; it can at most mean "like normally in our culture".
>
> *[...]*
>
> But of course, since the word "natural" means "cultural", it is subject to change. *[dB91a]*

The way mathematicians represent and therefore think of mathematical objects is highly dependent on their unconscious social and cultural context. No formalisation is fully suitable – unless your cultural background is logic. A mathematician is the most likely to be able to computerise his own mathematical works. According to de Bruijn, refinements could slowly move this computerised knowledge into a formalised one.

> Coming back to the idea of refinement, I must confess that it is difficult to keep it pure on the long run. The Genius at the beginning of the assembly line may look with one eye at what happens at the other end of the line, and may adapt his ideas to the needs of the technology displayed there. Technical realization can have influence on design.
>
> Moreover I am not sure that the idea of systematic refinement has an eternal value worth fighting for. After all, it is extremely conservative, and there is nothing against a revolution now and then. *[dB91a]*

In the last quotation, N.G. de Bruijn is right when he predicted that a *revolution* was possible. When we look at the theorem-proving community we could definitely characterise it as a revolution in the sense that it has invented a new way of approaching formal mathematics thanks to computers. This revolution has its singular language which is formal mathematics. It has its own community formed by users of theorem provers. This community is homogeneous in its deep belief that any mathematics needs to be formalised to reach a complete understanding. At the same time, this community is heterogeneous in the way each of its groups defends its turf as definite foundation for mathematics.

> Many mathematicians dislike pushing formalization to the extreme. The idea is that it kills intuitive thinking. I do not entirely agree. It may be true that unnatural formalization replaces intuitive thinking by an entirely different process of formula manipulation, but natural formalization supports intuition rather than destroying it. Formalization and intuitition should be each other's best friends rather than enemies.
>
> But part of what we call intuitive thinking is not of the kind that can be refined to proofs. That part cannot be formalized. Our brain processes are not based on logic or any other foundation of mathematics, and nevertheless they produce wonderful things. But all mathematicians agree that the results of intuitive thinking have to be justified by rigorous reasoning, even though there may be different opinions about the level of formality. *[dB91a]*

When presenting and discussing these perspectives we mainly insisted on the issue of the representation of mathematics. This should not lower in the reader's eyes the importance Automath played and still plays in the proof checking community. Automath is still a large source of inspiration for this community, see R.L. Constable's article [Con03] and other articles from the book *Thirty five years of Automating Mathematics* [Kam03], published to celebrate the anniversary of the Automath project. We also recommend readings of *Selected Papers on Automath* [NGdV94] and visits to the *Automath archive* web-site [Aut].

2.1.2 1987's article on the Mathematical Vernacular

In this section we present the Mathematical Vernacular (MV) which is the fundamental base on which was started this PhD research. In his 1987's article [dB87], N.G. de Bruijn describes MV as what should be a formal language for mathematics. He located MV as an encoding of mathematics in between full formalisation an mathematical natural language:

> The idea to develop MV arose from the wish to have an intermediate stage between ordinary mathematical presentation on the one hand, and fully coded presentation in Automath-like *[hence fully automated/ computerized]* systems on the other hand. *[dB87, §1.6]*

In N.G. de Bruijn's vision, MV is a formally defined language which represents informal mathematics (e.g. mathematics as we are used to seeing). MV does not require the mathematics described to be fully-formalised with all derivation steps logically justified. This language reproduces the traditional way mathematics is written; that is to say, with holes in the demonstration assumed to be easy to infer by the human reader. MV differs slightly from this traditional mathematical vernacular by enforcing the well-declaration of every symbol or notion prior to any use.

> A proof written in MV may be restricted to showing a sequence of resting points only. The derivation from point to point may be suppressed, or at least be treated quite informally. This seems to come close to the current ideal of mathematical presentation: impeccable statements, connected by suggestive remarks. *[dB87, §1.9]*

> In the first place MV allows us to omit parts of the proofs, at least as no definitions are suppressed. *[dB87, §1.13]*

In this section we present and discuss the key components of MV and what makes it an original approach for computerising mathematics. We first present in three steps the language constructions: lines and books in Section 2.1.2.1, contexts and flags in Section 2.1.2.2 and atomic expressions of the language in Section 2.1.2.3. The next two sections present MV's typing (Sections 2.1.2.5 and 2.1.2.6).

2.1.2.1 Lines and books

Line. As we mentioned earlier, Automath inspired MV. Similarly to Automath, MV has a line-by-line structure and each *line* is stated with a specific context.

Book. A *book* is formed by adding a new line to an existing book [dB87, §10.3]. A book is a finite partially ordered set of lines [dB87, §4.1, §8.5].

Context and line body. Each line is composed by a context (see Section 2.1.2.2), and the actual knowledge the line introduces, is called the line body. There are four kinds of line bodies: *definitional*, to introduce new symbols, *primitive*, to introduce primitive mathematical symbols, *assertional*, to make statements from preceding lines and *axiomatic*, to introduce a founding statement.

Valid line. A line is considered as valid if it is a correct continuation of the book formed by its preceding lines. By correct continuation we mean that all identifiers in the line are used in a suitable way according to the book.

Valid book. A valid book remains a valid book (in the sense of syntactic structure) if we omit the last line [dB87, §4.2]. The empty book (book without any line) is considered as valid.

2.1.2.2 Contexts and flags

A line has its own *context*. This context is a finite sequence of context items. These context items are either *assumptional* and *declarational*. Even if two consecutive lines share the same context, the context items are repeated for each line. N.G. de Bruijn introduced a notation to represent these context items with less repetition. Because such items are commonly shared from line to line, this notation is essential for human reading of MV books. The sharing of context items is represented in Automath by a pointer (indicator) to the last context item from which it is easy to chain-back every context items (we described this chaining of context items in Section 2.1.1.1.2). For MV, N.G. de Bruijn uses a more "natural" representation of contexts. This representation uses the *flag notation*[3]. A flag is composed by two parts: a *head* and a *flagstaff*. The flag's head contains the context item in question. The flagstaff covers all the consecutive lines for which this item occurs in the context. The flagstaff is a representation of the scoping of the context item. A flag illustrates the multiple occurrences of the same item in contexts of consecutive lines. A flag and its included lines are called a *block*. Flags are only a syntax sugaring in MV because they are not properly part of the language's definition. There exists an automatic conversion from normal notation to flag notation and vice-versa. This reverse conversion is not complete, because two consecutive flags holding the same information in their heads would be merged into a single flag if converted back and forth. Flags are only an implicit grouping of lines. The effective meaning of flags is the occurrence of similar context items in

[3] As N.G. de Bruijn mentioned, flags were firstly introduced by F. Fitch in 1950.

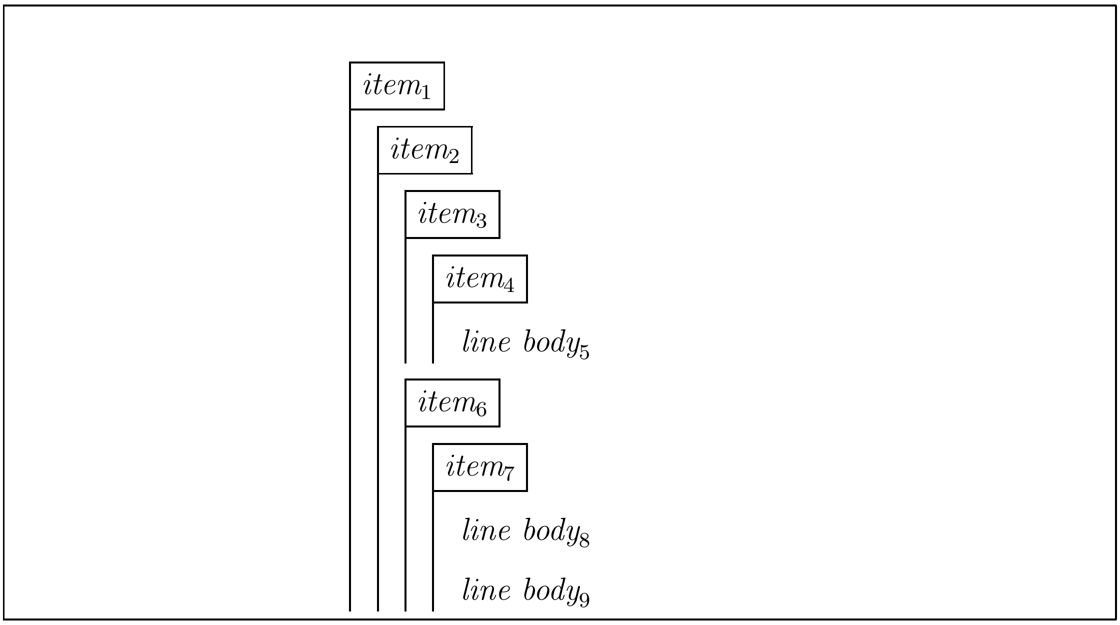


Figure 2.2: MV context with flag notation. Similar content as presented to illustrate Automath contexts in Figure 2.1

a similar order in consecutive lines. See Figure 2.2 for an example of uses of flags, this figure presents the same content as in Figure 2.1.

> The rules of MV do not just explain how mathematical sentences have to be formed, but also how they have to be manipulated in order to build new correct material. In particular they will help us to disclose the rules of the game of axioms, definitions, theorems and proofs.
>
> *[dB87, §1.7]*

2.1.2.3 Grammatical categories

The main specificity of MV is to propose a grammar for natural language mathematical text that differs from traditional grammars of natural languages. Grammars of natural languages are sets of rules governing the use of a language. MV is a grammar for mathematical language which dictates the way mathematical reasoning is to be constructed.

> In contrast to what one might expect at first sight, the grammar of the mathematical vernacular is not harder, but very much easier than the one of natural language. We can get away with only three grammatical categories (the sentence, the substantive and the name), because mathematicians can take a point of view that is very different

> from the one of linguists. The main thing is that mathematical language allows mathematical notions to be *defined*; it can even define words and sentences. In choosing these new words and sentences we have almost absolute freedom, just like in mathematical notation. We hardly need linguistic rules for the formation of new words and new sentences. It usually pleases us to form them in accordance with natural language traditions, but it is neither necessary nor adequate to set linguistic rules for them. *[dB87, §1.10]*

There are three grammatical categories in MV [dB87, §1.10, §3], usually extended to four with adjectives.

Statement. MV's statements are fact averred or not. They can play the role of either a sentence or a sub-sentence. A statement corresponds to a predicate in logic [KN02] and to the duet subject + predicate in traditional linguistic grammar [Cho57]. MV is only concerned with the grammatical role statements play in mathematical discourse and not with their veracity.

Example 1 *"$p \in R$" and "if AB meets d then AB is not parallel to CD" are examples of statements.*

Substantive. A class, a kind, a type or a family of mathematical objects is called substantive in MV. A substantive is a generic characterisation that has no equivalent in linguistic of natural languages. Substantives could be seen as types in type theory [KLN04] and concepts or classes in the ontology datamodel [W3C04].

Example 2 *"natural number" and "continuous function" are examples of substantives.*

Additionally, it is possible in MV to form sub-substantives to a given substantive. This relation is denoted $\ll$. By some means this relation implies the existence of an adjective (see below) which, when combined with the given substantive, would be equivalent to the sub-substantive.

Example 3 *Once we have the substantive "rectangular" we can form "square" as a sub-substantive of "rectangular": "square $\ll$ rectangular". This example is taken from [dB87, §1.14]. Here, square could be also defined as shortcut for "equilateral rectangular".*

Name. In MV, mathematical objects are names. If an identifier stands for an object, then this identifier is a name. A name can also be an entire phrase describing a mathematical object. Names correspond to individuals or instances in the ontology data-model.

Example 4 *"π", "the isosceles triangle ABC", "x" and "$\frac{a+b}{n}$" are examples of names.*

Adjective. Mathematicians commonly use adjectives to characterise an object or to restrict a class of objects to a sub-class that shares common characteristics.

Example 5 *Adjectives can either be used to form a new substantive: "continuous function", or to form a statement with a name "f is continuous".*

N.G. de Bruijn considers this set of four grammatical categories to be enough for a language for mathematics. To complete this descriptions of MV's grammatical categories we should mention the way sets are treated in MV. Substantives and sets are both represented by substantives since the difference between them lies in a level not covered by MV.

> It is customary to make the distinction between sets and classes. Roughly speaking, sets are classes over which we allow quantification. Usually we think of the classes which are no sets as those which are just too big to be sets, like the class of all sets. *[dB87, §1.18]*

A hierarchy of sets and substantives could be therefore constructed with adjectives and the sub-substantive relation. This is why N.G. de Bruijn described MV as a kind of typed set theory [dB87, §1.14].

2.1.2.4 Identifiers and parameters

Identifier. In MV, identifiers [dB87, §5] are the main material of texts. An identifier is an atomic piece of text and could either be a variable, a constant or a binder [dB87, §20].

An identifier usually gets its meaning from the line context or the binder introducing it in the case of variables, from a definition line or primitive line in the case of constants and from the MV grammar in the case of binders.

A primitive is a constant defined "without explanations in terms of known things" [dB87, §7.3]. Primitives are equivalent to Automath's primary notions (see Section 2.1.1.1.3).

Example 6 *"p", "AB", "not_parallel", "continuous", "function" and "rectangular" are among the identifiers used in the examples of Section 2.1.2.3.*

It is essential to differentiate MV words (identifiers) and the words used to form mathematical natural language texts. MV is interested with expressing mathematical discourse in a formal symbolic manner. MV words therefore represent atomic discourse entities. An MV word corresponds to one or more natural language words. It might happen that an MV word does not correspond to any word in its mathematical natural language equivalent.

> As an example we quote a definition: "We say that the vectors *p* and *q* are locally independent in the sense of Prlwtzkowsky if...". *[...]* The fact that the words "in the sense of" have been taken just in this order, does not play a role we consider to be essential for MV. It plays a role in readability, memorizability and possibly in parsability. *[dB87, §3.4]*

Parameter. In MV, parameters are a special case of variables. They are used when defining a parametrized constant. The set of variables contained in the context of a definitional line is the set of parameters for the constant being defined. This feature follows from an Automath tradition (see Section 2.1.1.1.4).

> It is essential that each one of the context variables occurs at least once in the parametrized constant. *[dB87, §5.5]*

2.1.2.5 Typings and clauses

Two levels of typing are defined for MV [dB87, §1.17, 3.6]. The first one named *low typing* – present at the language level – expresses the belonging of an object in a set or a kind of objects. The second one named *high typing* – also present at the language level – indicates whether an expression is a statement or a substantive. Both kinds of typing are part of the language itself in the same way Automath had different levels of expressions. MV's low-typing is equivalent to Automath's typing of 3-expressions (by means of 2-expressions) and MV's high typing is equivalent to Automath's typing of 2-expressions (by means of 1-expressions).

> *[...]* the rules of MV will not contain *all* of what is usually called the foundation of mathematics. Once we have reached a certain level, the language is strong enough to allow us to write the rest of the foundation of mathematics in an MV book. *[dB87, §8.1]*

When describing MV, N.G. de Bruijn presented an important notion called *clause*. A clause of a line body symbolises the information that could be highlighted from the MV text. For the axiomatic and assertional lines the clause is a high typing which indicates that the line body is a statement. For primitive and definitional lines the clause is the high typing of the symbol introduced. There is a second clause for definitional lines which is the high typing of the definition body.

2.1.2.6 Validation and automation

In the article [dB87], N.G. de Bruijn gives his opinion on automating the validation of MV books.

> One might think of a direct machine verification of books written in MV, but this will be by no means so "trivial" as in Automath. Checking books in MV may require quite some amount of artificial intelligence. *[dB87, §1.13]*

Some sets of rules are defined to check the well formation of an MV text. They are expressed in a way which indicates how a valid MV content can be extended.

> The rules of *[MV's]* grammar will be production rules, in the sense that they all describe ways to extend a valid book by adding a new line. *[dB87, §8.3]*

The rules are organised in several groups. The first one groups the basic rules to validate contexts, clauses [dB87, §9], books [dB87, §10] and common structures [dB87, §11]. The rules T1–T13 [dB87, §12] specify how the well formation of the substantives' hierarchy is verified. The further groups deal with equality [dB87, §13], sets [dB87, §14], pairs [dB87, §15], functions [dB87, §16] and logic [dB87, §17].

We see that an important part of these rules actually deals with validation of the mathematical or logical content of MV texts. This intention of validation means that the MV checker is not restricted to the grammatical formation of text. The basic rules are the only ones entirely focused on grammatical correctness.

> These rules BR1 to BR9 are hardly of a logical or mathematical nature. Or, rather, they describe how to *handle* logic and mathematics.
>
> *[dB87, §9.1]*

By restricting high and low typing to a small set of simple types and by restricting the checking to grammatical matters, R. Nederpelt updated MV to the so-called Weak Type Theory (WTT) [Ned02]. Concurrently, F. Kamareddine gave the meta-theory of WTT which was eventually published in [KN04].

2.2 The Weak Type Theory

The Weak Type Theory (WTT) was originally designed by R. Nederpelt as a refinement of MV [Ned02]. Concurrently, F. Kamareddine gave WTT's meta-theory which was eventually published in [KN04], see Section 2.2.4. WTT is an abstract language and a type system which could be used to express mathematical reasoning in a formal way, see Sections 2.2.1, 2.2.3 and 2.2.5. To each element of the language, the system attributes a weak type, see Section 2.2.2.

2.2.1 Linguistic categories

WTT defines an abstract syntax for a formal language of mathematics. The elements of this syntax are classified according to four language levels. *Variables*, *constants* and *binders* belong to the *atomic* level. *Terms*, *sets*, *nouns* and *adjectives* belong to the *phrase* level. *Statements* and *definitions* belong to the *sentence* level. Finally, *contexts*, *lines* and *books* belong to the *discourse* level. Statements are also sometimes listed in both phrase and sentence levels.

WTT makes explicit the grammatical role of each linguistic piece of text. Therefore, the linguistic categories of the atomic level are split into disjoint subsets depending on the grammatical category of the identifier. This grammatical category is indicated by upper indices adjoint to the symbol representing the linguistic category (see Section 4.4). Variables (respectively constants and binders) are split into term and set variables (respectively term, set, noun, adjective and statement constant, and term, set, noun, adjective and statement binders). Statement constants are sometimes divided into relational constants, such as $\geq$, and logical constants, such as $\wedge$. The sets of variables, constants and binders are given beforehand and are infinite. Variables have to be declared in a context prior to being instantiated to form a phrase. Constants have to be defined by a definition prior to forming a phrase or a sentence.

Similarly to MV, a context is a sequence of assumptions and declarations. A line is the context plus sentence tuple. And a book is an ordered sequence of lines. WTT inherits MV flags described in Section 2.1.2.2.

2.2.2 Weak types

The use of the adjective *weak* as an attribute for a type system would indicate that WTT is a small type system. But weak should be seen as an attribute to the word type. A weak type is by definition not prevalent nor potent. By extension, weak typing is a light or generic judgment. WTT defines eight weak types. These types – `book`, `cont`, `term`, `set`, `noun`, `adj`, `stat` and `def` – correspond directly to the grammatical categories of the the abstract syntax. This list of weak types is highly related to the way the WTT typing rules are written. Some weak types could have been added to this list for completeness but were not essential in the type system. For instance, `line`, `dec` and `phrase` could well be weak types as in MWTT Section 4.4 (see Section 4.2.1.1 where a homogeneous set of weak types is defined for MathLang-CGa).

2.2.3 Type system

WTT defines a set of well formation criteria to validate WTT mathematical texts. WTT implements N.G. de Bruijn's idea of a line-by-line language where validity is defined according to the relation between a line and the book formed by the previous lines. A typing judgment for a book B is stated in the empty environment.

$$\vdash B :: \texttt{book}$$

A typing judgment for a context C is stated in the environment formed by a well-typed book B.

$$B \vdash C :: \texttt{cont}$$

A typing judgment for a term t (respectively a set s, a noun n, an adjective a, a statement p and a definition d) is stated in the environment formed by a well-typed book B and a well-typed context C (C being well-typed in B).

$$\begin{array}{rcl} B;C & \vdash & t :: \texttt{term} \\ B;C & \vdash & s :: \texttt{set} \\ B;C & \vdash & n :: \texttt{noun} \\ B;C & \vdash & a :: \texttt{adj} \\ B;C & \vdash & p :: \texttt{stat} \\ B;C & \vdash & d :: \texttt{def} \end{array}$$

WTT rules are equivalent to MV's basic rules (BR1–BR9 of [dB87, §9]). See Section 4.4.2 for the set of typing rules of MWTT which is an extension of WTT.

2.2.4 Meta-theory

In [KN04], one finds the followings:

Corollary 1 (Decidability of weak type checking) *Weak type checking is decidable: there is a decision procedure for the question $B; C \vdash E :: t$?.* *[KN04, Corollary 4.21]*

Corollary 2 (Weak typability) *Weak typability is computable: there is a procedure deciding whether an answer exists for $B; C \vdash E ::$? and if so, delivering the answer.* *[KN04, Corollary 4.21]*

They show that WTT has subject reduction with respect to the unfolding of definitions in a book. The unfolding of definitions is equivalent to δ-reduction (see 2.1.1.2.2), we note it $\rightarrow_\delta$.

Theorem 1 (Subject reduction) *If $B; C \vdash E :: t$ and $B \vdash E \rightarrow_\delta E'$, then $B; C \vdash E' :: t$.* *[KN04, Theorem 4.28]*

And finally, they prove that WTT's unfolding of definitions is strongly normalising.

Theorem 2 (Strong normalisation) *Let $\vdash B :: \texttt{book}$. For all subformulas E occurring in B, relation $\rightarrow_\delta$ is strongly normalising (i.e., definition unfolding inside a well-typed book is a well-founded procedure).* *[KN04, Theorem 4.40]*

2.2.5 Expressiveness

Make grammatical roles explicit. From a computer scientist point of view, WTT's abstract syntax may sound highly redundant. The identifiers are to be given and their grammatical category fixed before any WTT text can be written. This makes WTT somehow *more* explicit than strongly typed programming language. From a mathematician point of view this set-based definition of WTT uses familiar jargon. This follows N.G. de Bruijn's idea of *a language for mathematics with typed sets.*

Weak-validation. WTT does not make any analysis on the mathematical meaning of a text but because each identifier is part of a grammatical category, the grammatical correctness is validated. For example, it is possible to write in WTT that a variable x is equal to 1 and to 0:

$\boxed{x : \mathbb{N}}$

$x = 1$ *and* $x = 0$

This is grammatically correct even if semantically incorrect. But WTT derivation rules will lead to an error with the following example:

$\boxed{x : \mathbb{N}}$

$x = 1 \Rightarrow 1$

This example would correspond to the sentence "let x be a natural number, we have that x implies 1" which could not mean anything if 1 is a number and not a particular statement.

2.2.6 Perspectives

WTT inspired some researches at Technische Universiteit Eindhoven which are concerned with moving from WTT to Type Theory. G. Jojgov and R. Nederpelt describe in [JN04, Joj06] ways to extend the analysis of a WTT document by recording assumptions in a context as future proof obligation for a full formalisation. Ultimately, the goal is to reach a robust formalisation similar to those performed by theorem provers (see Section 2.4.3).

Later, in Section 4.4, we describe our early work MWTT which is a refinement of WTT. We do not give the set of typing rules of WTT since the typing rules of MWTT (see Section 4.4.2) are based on those of WTT. Hence the reader can get the feel of the WTT rules from those of MWTT.

2.3 The MathLang Project

MathLang is a framework for mathematics on computers.

1. *MathLang is* ***a framework****.* It is meant to be used for communication and as a concrete support for human mind formulation. MathLang is a well structured framework aimed to synthesize the common mathematical language.

2. *MathLang is* ***for mathematics****.* It is meant to be open to any branch of mathematics and to any topic that uses mathematics as a base language. MathLang mimics mathematics in its incremental construction of a body of knowledge.

3. *MathLang is* ***for computerisation****.* MathLang is meant to be a medium for a human-system, human-human via a digital support, and system-system communication. MathLang is a computer-based framework and therefore offers automation facilities.

2.3.1 MathLang's Philosophy

The language used by mathematicians to express, show and explain mathematics has been refined by millennia of discoveries and advancement in abstract reasoning. We inherited from this Common Mathematical Language (CML) which is perceived as the most rigorous of the natural languages (see Section 2.4.1.1). A number of mathematicians and philosophers have advocated the use of a formal language (instead of CML) in order to exclude any ambiguity from the reader. The philosophical discussions over the essence of mathematical deductions and demonstrations led to the elaboration of logics. The search for a foundation of mathematics where proofs could be expressed in a formal language is ongoing and has already led to accepted and time-honored foundations.

A number of mathematicians and philosophers have advocated the use of a formal language (instead of CML) in order to exclude any ambiguity from the reader. The philosophical discussions over the essence of mathematical deductions and demonstrations led to the elaboration of logics. The search for a foundation of mathematics where proofs could be expressed in a formal language is ongoing and has already led to accepted and time-honored foundations. Nevertheless, despite these influential efforts, CML remained the communication medium for mathematicians. To accommodate the use of this informal but expressive language it is common to assume the feasibility of consistently transcribing "formal" CML text into a formal foundation.

> In general he [the mathematician] is content to bring the exposition to a point where his experience and mathematical flair tell him that translation into formal language would be no more than an exercise of patience (though doubtless a very tedious one).
>
> *[Bou54, Bou68, Chapter I]*

With their computational capabilities, computers offer the possibility to put formalisation into practice. Many systems (see a comprehensive comparison in F. Wiedijk's work [Wie03, Wie06]) offer computational tools to automatically check the correctness of formal proofs. These systematic validations are time-wise impractical by humans. Automath (see Section 2.1.1) and Mizar (see Section 2.4.3.2) are among these precursor computer-based languages and theory checking systems. Computers can also in a way "invent" mathematical knowledge. The research field of automatic deduction aims at assisting the mathematician or even replacing the mathematician by searching for a valid proof of a given property. But this revolutionary approach to mathematics (as we presented in Section 2.1.1.3, page 17, using N.G. de Bruijn's own words) has its skeptics. Some people regard formal logic as being of philosophical interest but consider that mathematics should not be restricted only to formal proofs. They have not found any interest in using formal computer-based systems. This opposition between computer-based formal mathematics and more intuition-based mathematics raises the following question: *Does all mathematics need to be formalised?*

- If the answer to this question is positive then we end up with other practical questions. How should we consider branches of mathematics that have not been formalised? What is the role in the universal body of mathematics for non-formalised materials? And materials known to contain logical or foundational mistakes (therefore non-fully-formalisable)? How do we treat draft mathematics and unaccomplished theories that can not yet fit into a formal system? We are at a period of mathematics where computer-checked proofs are gaining respect and importance in the mathematical community –

the Four-Colour Theorem is the first major theorem computer-proved [Gon] (using the Coq proof assistant, see Section 2.4.3.1), and the Mizar formalisation of the Jordan Curve Theorem [Kor05] in the Mizar Mathematical Library [MML]. Nevertheless, the dreams of a comprehensive library for formal mathematics has not come true [BW05], and it is not clear if it will ever come true.

- If the answer to the above question is negative then we need to question the usability of proof assistant software by mathematicians. For most such software, the only possible outcome for a document, is to be a logically valid one. This orientation creates early choices for the authors. In such framework, the design of a theory and the elaboration of proofs are oriented towards achieving full-formalisation. Mathematicians can only think of using a formal system if they are sure a formalisation will be carried out to its final close and obviously if they need such validations.

MathLang's philosophy is to bridge the gap between common mathematical writings and full-formalisation. Our starting point is to propose a solution for putting mathematics on the computer that:

- Leaves the author free to edit any mathematical texts (in terms of a branch of mathematics and also in terms of a level of correctness).
- Gives a framework in which CML is the medium for the human reader but where the reasoning, expressed in CML, is computerised in a comprehensive manner for computer-based analysis,
- Opens the possibility of semantical and logical refinements and even formalisation if achievable or needed.

In the MathLang project and in this thesis we pursue the idea of providing a medium for handling mathematical knowledge on computers. We believe this goal could only be achieved by merging in one framework both the traditional-mathematical-vernacular distinctive aspects and the advanced computer-based technologies for managing formal mathematics. In Section 2.3.2 we explain why the MathLang project seeks the computerisation of mathematics in opposition to its formalisation. Considering computerisation as an authoring trend, we prolong this reflection by comparing, in Section 2.3.3, the act of encoding and the act of translating. In Section 2.3.4 we explain the MathLang project choice to decompose the computerisation/formalisation of mathematics into meaningful aspects. These discussions continue in Chapter 3 concerning the particular aspects defined in this thesis.

2.3.2 Computerisation vs. formalisation

Prior to any discussion we shall first define the two notions of *computerisation* and *formalisation* as they could mean different things in different contexts.

By *computerisation* we mean the process of putting a document in a computer format. This could mean a wide variety of processes because the variety of computer languages is vast (see Section 2.4). We consider here mathematical documents in their printed form as found in mathematical textbooks or scientific journals. This preemptive consideration puts a constraint on computerisation. A computerised document is no more than a computerised version of such a traditional document. In this sense, computerisation is concerned with both (1) providing the knowledge contained in a mathematical document and (2) reflecting the original style in which the document existed (or by extension would have existed in the case of fresh authoring).

By *formalisation* we mean the process of extracting the essence of the knowledge contained in a mathematical document and providing it, in a formal document, in a complete, correct and unambiguous format. A formalised document should clarify all notions contained in a document as well as making explicit all the details of any logical deductions it contains. Again, this process could be done in a vast variety of manners depending on the foundation and the framework used (see Section 2.4). Formalisation is therefore concerned only with the first point of the previous paragraph (i.e. providing the document's mathematical knowledge). The second point (i.e. reflecting the document's original style) is more or less tackled in formalisation frameworks (see Section 3.2.1). It is usually an artifact in the sense that it attempts to recreate a seemingly natural document out of a formal one.

Our choice for MathLang is to provide a language for the computerisation of mathematics. We root this choice in N.G. de Bruijn's vision (see Section 2.1.1.3). We indicate here the major characteristics of computerisation.

Accessibility. In Section 2.1.1.3, we discussed the difficulty of choosing a suitable foundation when formalising mathematics. Formalisation effectively requires some expertise because it forces one to make choices: the choice of a terminology for representing concepts, the choice of the underlying logic for representing the deduction steps, the choice of a formal system between Zermelo-Frankel set theory [HK68], type theory [KLN04], category theory [Lan71] and others, and the choice of a proof checker, see Section 2.4.3 and [Wie06]. These choices are not easy to make and require a certain amount of knowledge in the theoretical and implementation aspects of formalisation. MathLang chose to offer a framework accessible for non-experts in formalisation where these choices are not required until being really needed.

Inclusion. As mentioned above MathLang's goal is not to restrict the mathematician-user to a particular logic, foundation or system. For a language to computerise mathematics, it is important to embrace all authoring habits. By authoring habits we mean the author's formalisation choices as well as the author's writing style. The language should be adaptive enough so that the user does not have to adapt his own way of doing things. This inclusion is even more important when we consider the encoding of existing libraries of mathematics such as mathematical textbooks and journal articles. A computerisation of such documents has to adapt to their style otherwise we shift from computerisation to translation (see Section 2.3.3).

Reusability. An important aspect of the computerisation of mathematical documents is the dissemination. After the process of computerisation, a normal continuation is to consider moving towards semantic refinement and more precision in the logic implemented. This might not be done or feasible for every document (we discuss this issue in Section 2.1.1.3) but this possibility should be taken into consideration when designing a language for computerising mathematics.

The MathLang project aims at computerising mathematics and considers computerisation as a process that should not be constrained and should capture the natural language parts of mathematical texts as well as making its content explicit.

2.3.3 Encoding vs. translation

The way we treat mathematical authoring has strong implications on the way we incorporate existing mathematical documents in a computer-based corpus of mathematics. The formalisation of major treatise of mathematics is often effectively a translation from its natural language form to a formal proof. A formalisation offers several advantages. Firstly, it makes the original document's theory and results accessible and reusable inside the formal framework in which the formalisation was done. Secondly, it permits to formally prove the content of the document which was to a certain extent only proof-read by the editorial-board and the readers.

A number of research teams around the world have been and still are working on creating a digitised library out of these formalised documents. Among others we cite here: L.S. van Benthem Jutting [vBJ77b] of the Automath project who formalised E. Landau's *Foundations of Analysis* [Lan30] in Automath, and the project[4] lead by G. Bancerek [BR02] who formalised in Mizar with 15 other authors most parts of *A Compendium of Continuous Lattices* [GHK$^+$80] and included this formalisation in Mizar Mathematical Library (MML).

In the MathLang project our primary goal is to accommodate computerised mathematics with mathematicians' needs. We base our effort on computerising existing mathematical documents (see Section 5.3). We are therefore interested in encoding mathematical texts more than translating them into a formalised form. The current aspects of MathLang do not aim at providing yet another foundation for mathematics and its digital library but to provide a suitable authoring method (see Sections 5.1 and 5.2) for the working mathematician.

2.3.4 Decomposition into Aspects

Back in 2000, F. Kamareddine and J.B. Wells started the project MathLang. One of the initial characteristic of the language and framework they were willing to develop was their decomposition in terms of *levels* of computerisation. MathLang's proposals [KW00, KW01] included libraries of computerised mathematical texts, bridging with a number of proof checkers (Mizar, Ωmega, Isabelle, etc) and different levels of formalisation and computerisation of mathematics. According to these proposals, the computerisation or formalisation should be facilitated by a step-by-step approach where a CML text is first translated into language similar to WTT (reaching the first level) and then refined level after level with more logics and semantics. This method should be more accessible than a direct formalisation because the first level do not require any particular expertise in formalisation.

As a result of the research and experiments carried out on the various computerisation in MathLang by undergraduate and postgraduate research since 2000 (including this thesis), J.B. Wells proposed in 2005 replacing the *levels* of formalisations by the so-called *aspects*. The notion of aspect permits a greater focus

[4] `http://megrez.mizar.org/ccl/` (last visited 2007–04–20)

on knowledge capture by each decomposition elements contrast to the notion of level which implies a stratification and therefore an obligation to meet one level before another. Figure 2.3 depicts the current aspect-oriented MathLang method for authoring mathematics. The computerisation/formalisation path goes from the top-left to the bottom-right corner of the figure. CGa (see Chapter 3) and Document Rhetorical aspect (DRa) [KMRW07b] (see Section 7.1.1.2 for current and future works) inputs could be done sequentially, simultaneously or independently. The interfacing with CML is managed by TSa (see Chapter 5).

2.4 Mathematics on Computer, a State of the Art

In this section we draw the picture of the situation of mathematics on computer today. This state of the art is not exhaustive but is the result of a thought search over available techniques for encoding mathematics. We try to present a wide overview of the activities of the Mathematical Knowledge Management community whose flourishing activities are presented at the MKM conferences and events [MKM01, MKM03, MKM04a, MKM04b, MKM06a, MKM06b, MKM07]. We organised the summary of these activities into three categories of system. All the languages and systems from each category share the same end-goal. Section 2.4.1 is interested in typesetting systems and their support languages. Section 2.4.2 investigates the available languages for semantical encoding of mathematics. Finally, in Sections 2.4.3 and 2.4.4 we present well-advanced systems and languages for the formalisation of mathematics on computer.

All along this section we make an effort to illustrate the use, capability and expressiveness of these languages. We would like this census not to simply be a basic enumeration of what exists but also a help in comparing these languages. For this reason we use one main example as a connecting link throughout this section. We use a proof of the irrationality of square root of two due to G.H. Hardy and E.M. Wright [HW80]. Figure 2.4 shows this proof as presented by F. Wiedijk in [Wie06]. In [Wie06], F. Wiedijk juxtaposes versions of this proof as encoded and validated by each prover participating in this experiment. In [Wie03], F. Wiedijk presents the result of the comparison of these formalised and computerised proofs and likewise compares the provers themselves. We extend this comparison in Section 3.2.1 to a different level (i.e. different from pure formalisation).

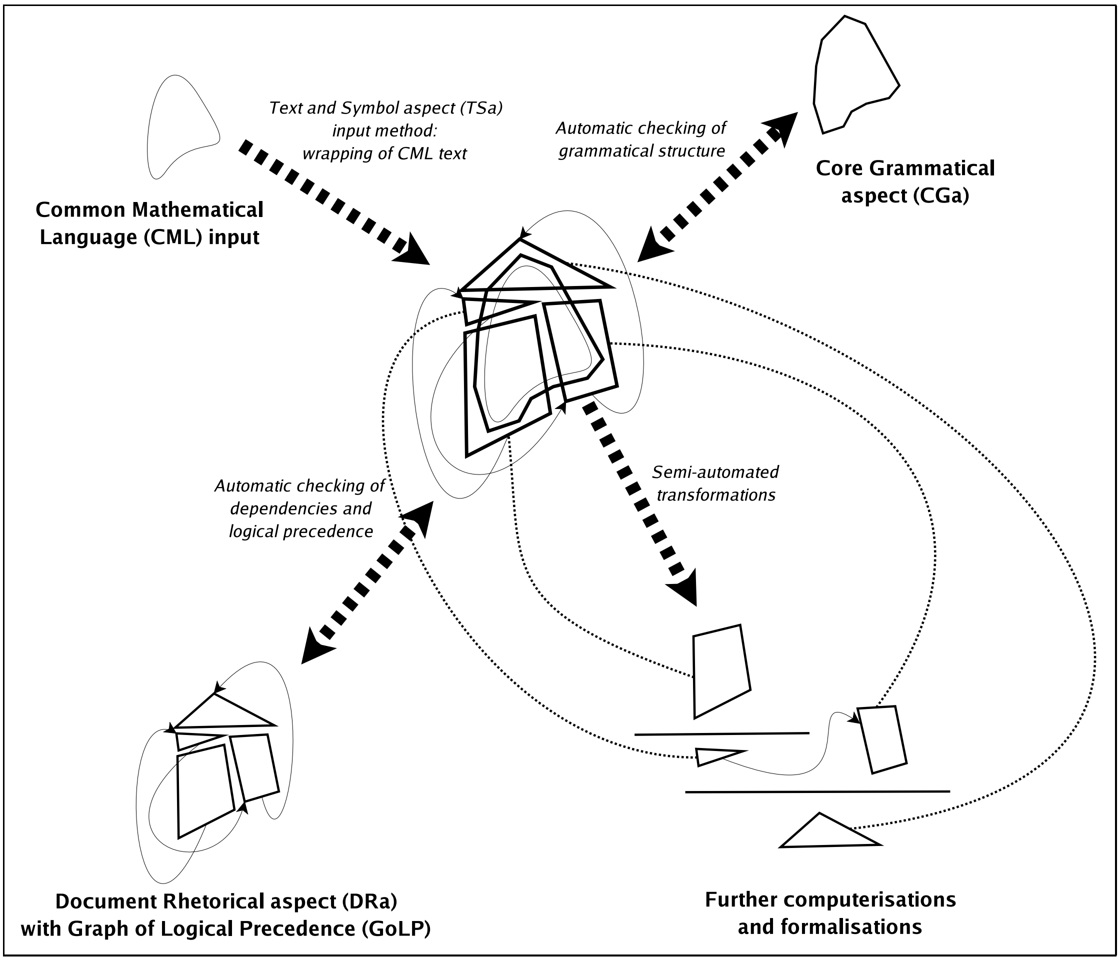
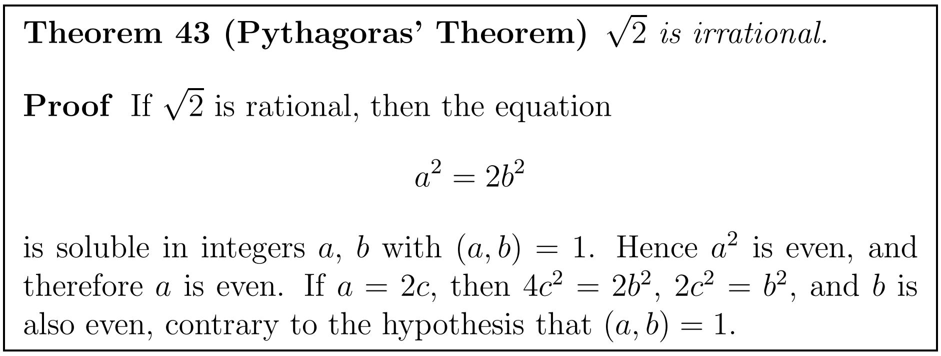


Figure 2.3: MathLang project

This diagram illustrates both the MathLang decomposition by means of knowledge aspects and the MathLang authoring process. The central piece represents a MathLang document while the corner pieces are elements of the authoring process.

CML (top-left corner) is the starting point of any MathLang authoring. **TSa** (see Chapter 5) offers facilities to associate portions of text with their meaning in terms of computerised data.	**CGa** (top-right corner) makes explicit the grammatical role played by the elements of the texts. An automatic checking validates this grammatical aspect (see Chapter 4).
DRa (bottom-left corner) gives the narrative structure of a text which includes the relationships (pictured as arrows) between labeled text entities (pictured as polygons) and **GoLP** is an interpretation of these relationships in terms of logical precedence [KMRW07b] (see Section 7.1.1.2).	Starting from a MathLang document we experimented with **further computerisations/formalisations**. We illustrate this point (bottom-right corner) with a proof-tree-like interpretation which is a possible starting point for building a formalised proof [KMRW07a] (see Section 7.1).

Thick arrows draw the MathLang authoring process which starts with CML and continues to further formalisations. CGa and DRa steps could be made simultaneously or separately. See Figure 3.1 for more a description of the shapes meaning.



Theorem 43 (Pythagoras' Theorem) $\sqrt{2}$ *is irrational.*

Proof If $\sqrt{2}$ is rational, then the equation

$$a^2 = 2b^2$$

is soluble in integers a, b with $(a, b) = 1$. Hence a^2 is even, and therefore a is even. If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$, and b is also even, contrary to the hypothesis that $(a, b) = 1$.

Figure 2.4: Pythagoras' proof of irrationality of $\sqrt{2}$ by G.H. Hardy and E.M. Wright [HW80] as presented by F. Wiedijk in [Wie06]

2.4.1 Mathematical Typesetting Systems and Languages

2.4.1.1 The Common Mathematical Language

The mathematics we are taught from primary school to university level is almost always written in a distinctive language style. This style differs from other natural languages by its extensive use of abstraction and its rigor. This very same language can designate first order equations, geometric problems or demonstrations in general topology. Of course the background requested to understand what is expressed depends on the knowledge of the reader, but what remains is a certain uniformity of the language. This linguistic style has been called the Common Mathematical Language (CML) in [KN04]. Its ingredients are symbols composed into formulas and natural language chunks. Recognisable text constructions, familiar labels and fonts constitute the visible substance of CML texts. However, an important part of what makes the consistences of a text – such as the connections between pieces of text, a justification of an obvious deduction or the origin of a variable – is left to the reader's understanding.

Mathematical typesetting systems are familiar to mathematicians but also to anyone wanting to print texts with mathematical formulas on paper or screen. The users of these systems could be as different as students, scientists or publishers. Most of the traditional typewriting systems could be combined with an equation editor for inserting mathematical formula. Modern commercial and Free office software suites include such editors. The open office-suites such as KOffice (`http://www.koffice.org/`), OpenOffice (`http://www.openoffice.org/`) or Gnome Office (`http://www.gnome.org/gnome-office/`) have their own open-formats that follow the XML recommendations. They have built-in or plugin facilities for editing mathematical formulas. In this section we mainly focus on formats specifically dedicated to scientific and mathematical editing.

2.4.1.2 LaTeX

The most commonly used language for authoring mathematics is LaTeX. It is widely accepted by publishers of mathematical journals and textbooks due to its typesetting quality. LaTeX is a system and a programming language for publishing with the TeX typesetting program created by D.E. Knuth [Knu84a]. LaTeX, designed by L. Lamport [Lam94], provides programming features for automating most aspects of typesetting and offers extensive facilities for specialising its use.

```
\begin{theorem}[Pythagoras’ Theorem]
  $\sqrt{2}$ is irrational.
\end{theorem}
\begin{proof}
  If $\sqrt{2}$ is rational, then the equation
  $$a^2 = 2b^2$$
  is soluble in integers $a$, $b$ with $(a,b)=1$.  Hence
  $a^2$ is even, and therefore $a$ is even.  If $a=2c$, then
  $4c^2=2b^2$, $2c^2=b^2$, and $b$ is also even, contrary to
  the hypothesis that $(a,b)=1$.
\end{proof}
```

Listing 2.1: LaTeX's code of corresponding Figure 2.4's example

Listing 2.1 presents the LaTeX code we used to input our example in Figure 2.4. This LaTeX code is a succession of:

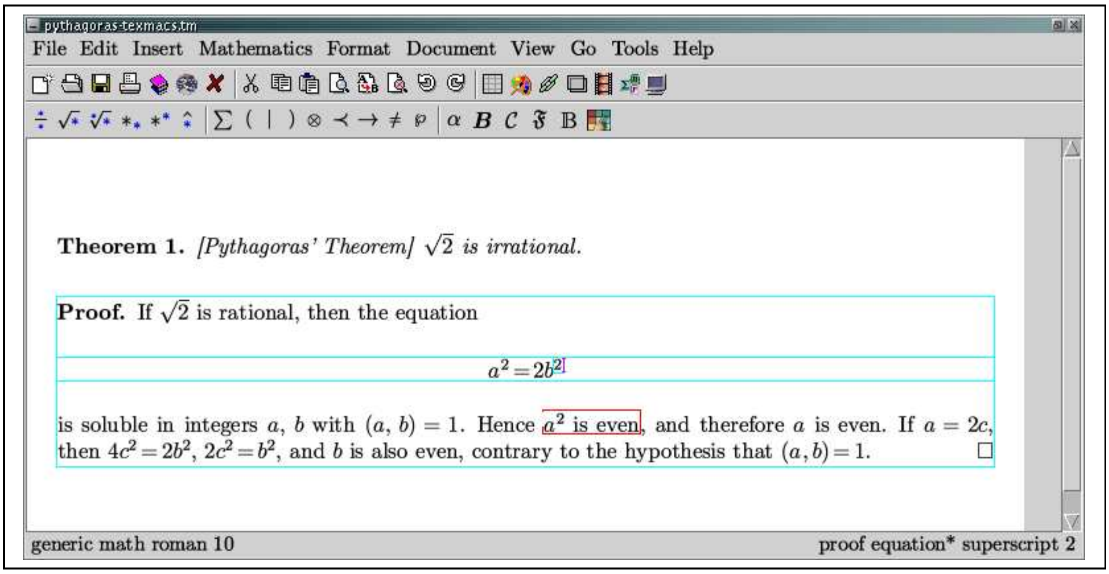
- letters forming natural language words,
- LaTeX commands (starting with the backslash sign `\`) and arguments (delimited by the curly brackets `{` and `}` or square brackets `[` and `]`),
- LaTeX mathematical mode (delimited by single `$` or double `$$` dollar sign) for formulas.

A computer program, such as the LaTeX compiler, can interpret this code to produce a visual output similar to the one shown in Figure 2.4.

LaTeX is a one-dimensional encoding (sequence of characters) of a two-dimensional CML text (on-paper or on-screen output) and is therefore a computerised version of CML. Would the fact that LaTeX is a computerised language help any semantical analysis? The environments `theorem` and `proof` hold some semantics as they delimit a piece of text and their name indicates the role this piece of text plays in the text. Formulas could also be recognised if they are composed of standard symbols. When having a careful look at the formulas of our example we note that some may cause difficulties to an automatic analyser. A human reader could easily understand, in the context of arithmetic, that (a, b) refers to the greatest common divisor (gcd) but this recognition is not straightforward for a computer software. The background context might help to differentiate between a simple tuple (a, b) (representing coordinates for example) and the gcd ((a, b) should be interpreted as $\mathrm{gcd}(a, b)$). Furthermore, formulas remain ambiguous in LaTeX, as explained in details in [Pad03], due to the fact that the precedence of symbols is not explained and that many operations such as the multiplication or, as we saw here, the gcd are not associated with specific LaTeX commands. Even if we consider the environments and the formulas to be identifiable, the task of inferring the role played by formulas within the natural language text remains tremendous as natural language recognition techniques are still under development. M. Kohlhase proposed in [Koh04] to carefully choose TeX macros and LaTeX commands to help such recognition. Even in the best case, LaTeX can not be expected to capture the semantic content of natural language text any better than OMDoc (see Section 2.4.2.2).

2.4.1.3 MathML

MathML [W3C03], which was the first language implementing the XML recommendations [Worb], describes a method to encode both the appearance and the semantics of mathematical formulas. MathML deals with the rendering aspect of mathematical formulas with MathML-Presentation and with the encoding of the meaning of formulas with the MathML-Content. Many mathematical software offer MathML input/output capabilities. Its expressiveness is restricted by the fixed set of symbols it defines and the absence of a macro system (at the time of writing this thesis, a macro system for MathML is part of the future extensions listed in [W3C03, Ch. 7]). Even if MathML is widely used and has important software support it is too closely associated to its presentational aspect. MathML is similar to the LaTeX mathematical mode in its way of encoding in an XML tree the appearance structure of a formula [Pad03]. MathML permits to adjoin to the encoding of the visual representation of a formula its semantical interpretation. MathML is restricted to formulas and relies on XHTML [W3C06], for example, for shaping the formulas inside a document.



pythagoras-texmacs.tm
File Edit Insert Mathematics Format Document View Go Tools Help
generic math roman 10
proof equation* superscript 2

Theorem 1. *[Pythagoras' Theorem]* $\sqrt{2}$ *is irrational.*

Proof. If $\sqrt{2}$ is rational, then the equation

$$a^2 = 2b^2$$

is soluble in integers a, b with $(a, b) = 1$. Hence a^2 is even, and therefore a is even. If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$, and b is also even, contrary to the hypothesis that $(a, b) = 1$. □

Figure 2.5: TeXmacs user interface

2.4.1.4 TeXmacs

TeXmacs [vdH01, vdH04] is a free scientific text editor. It offers the same typesetting quality as LaTeX but integrated in a What You See Is What You Get (WYSIWYG) editor. Figure 2.5 is a screenshot of the TeXmacs Graphical User Interface (GUI). This figure shows the edition of Figure 2.4's text within TeXmacs. TeXmacs internal representation is a markup language which is an disrelated mixture of XML-like tree structure and LaTeX-like commands, see Listing 2.2 for an illustration with the source code of the document shown in Figure 2.5.

TeXmacs is used as an interface for many external systems such as computer algebra systems or theorem provers, thanks to its plugin system. The TeXmacs plugins for Coq TmCoq and tmEgg are respectively presented in [AR03] and [MG06]. In [ABFL05, WAB06, ABFL06, AFNW07], the Ωmega group at the Universität des Saarlandes presents its recent work in using TeXmacs as an interface for their proof assistant [BCF$^+$97]. We discuss in details our use of TeXmacs as an interface for MathLang in Section 6.3.

```
<TeXmacs|1.0.6.7>

<style|generic>

<\body>
  <\theorem>
    [Pythagoras' Theorem] <with|mode|math|<sqrt|2>> is irrational.
  </theorem>

  <\proof>
```

```
    If <with|mode|math|<sqrt|2>> is rational, then the equation

    <\equation*>
      a<rsup|2>=2b<rsup|2>
    </equation*>

    is soluble in integers <with|mode|math|a>, <with|mode|math|b> with
    <with|mode|math|(a,b)=1>. Hence <with|mode|math|a<rsup|2>> is even, and
    therefore <with|mode|math|a> is even. If <with|mode|math|a=2c>, then
    <with|mode|math|4c<rsup|2>=2b<rsup|2>>,
    <with|mode|math|2c<rsup|2>=b<rsup|2>>, and <with|mode|math|b> is also
    even, contrary to the hypothesis that <with|mode|math|(a,b)=1>.
  </proof>
</body>

<\initial>
  <\collection>
    <associate|font-base-size|12>
  </collection>
</initial>
```

Listing 2.2: TeXmacs sources of the document shown in Figure 2.5

2.4.2 Semantic Markup Languages

Following the advance in the standardisation of data models, several project were started in the past decades to propose generic semantical encodings of mathematics. These efforts lead to the creation of document markup languages such as OpenMath[5], MathML[6] and OMDoc[7].

These markup languages were originally aimed at offering a standardised and open format for encoding mathematical formulas. The first project, OpenMath was started in 1993 by mainly European research groups. The resulting markup language OpenMath followed the XML recommendations in its later versions. The World Wide Web Consortium (W3C) is developing MathML, see Section 2.4.1.3 dedicated to MathML.

2.4.2.1 OpenMath

OpenMath (due to the fact that it was initiated by research groups) focuses on a more semantical encodings of formulas in comparison with MathML. OpenMath [AvLS96, BCC$^+$04] is a language for encoding mathematical formulas. Mathematical symbols are defined in OpenMath Content Dictionaries (OMCD) (symbol

[5] http://www.openmath.org/

[6] http://www.w3.org/Math/

[7] http://www.omdoc.org/

definitions and definitional properties). The OpenMath standard library provides numerous OMCD including an impressive number of symbols. Users are also free to write their own OMCDs. Developers of computational systems can define in an OMCD the set of symbols recognised by their system. The communication of OpenMath formulas between two systems is made possible after both systems have been *taught* how to read each others OMCD. However, OpenMath lacks sufficient theoretic and software support for checking the well-formation of formulas. In [Str03, Str04], A. Strotmann gives a clearer understanding of the OpenMath language itself. He used categorial semantics to analyse the OpenMath markup language. Nevertheless his work was not aimed at analysing the semantics of OpenMath objects. There exists currently one implemented validation tool for OpenMath objects. This is simply a syntactical analysis of the structure of the XML encoding of the OpenMath object. A semantical validation process for OpenMath objects is described in [CC99] by O. Caprotti and A. Cohen. For each symbol definition (`<CDDefinition>`), a type signature (`<Signature>`) could be provided. It uses the Extended Calculus of Constructions (ECC) to validate OpenMath contents. This analysis is based on some type annotation given by the OpenMath user in OMCDs. In [Dav99], a more complex system, the Simple Type System (SST) [Dav00], has been used for the same purpose. But unfortunately, no implementation of these validation processes are currently available. See Section 7.2 for envisioned outcome of this thesis on the particular subject of checking OpenMath objects. The issue of the rendering of OpenMath formulas is yet to be solved. There is ongoing work on tackling this issue, see [MLUM06].

2.4.2.2 OMDoc

In the semantic markup languages MathML-Content, symbolic formulas are encoded using a library of predefined symbols. In OpenMath, the symbols are defined in some OMCDs. OMDoc [Koh06] was created by M. Kohlhase to add a theory/ document level to OpenMath. There are many ways to write our examples from Figure 2.4 in OpenMath/OMDoc. We sketch one possible encoding in Listing 2.3.

```
<assertion id="th" type="theorem">
   <commonname> Pythagoras' Theorem
   <FMP> <OMOBJ> $\sqrt{2} \notin \mathbb{Q}$
   <CMP> <OMOBJ> $\sqrt{2}$ is irrational.
<proof id="pr-th" for="th">
   <CMP> If <OMOBJ> $\sqrt{2}$ is rational, then the equation
```

```
            <OMOBJ> $a^2 = 2b^2$ is soluble in integers <OMOBJ> $a$,
            <OMOBJ> $b$ with <OMOBJ> $(a, b) = 1$. Hence <OMOBJ> $a^2$ is
            even, and therefore <OMOBJ> $a$ is even. If
            <OMOBJ> $a = 2c$, then <OMOBJ> $4c^2 = 2b^2$, <OMOBJ> $2c^2 = b^2$,
            and <OMOBJ> b is also even, contrary to the
            hypothesis that <OMOBJ> $(a, b) = 1$.
```

Listing 2.3: An OMDoc/OpenMath encoding of Figure 2.4's example.
For readability and brevity, we show only the opening tag of each XML element; instead we use indentation to express nesting. We also use traditional mathematical output prefixed with the OMOBJ *tag for OpenMath formulas instead of showing the XML tree.*

OMDoc provides a wide variety of elements. An OMDoc document is organised into theories (theory elements). Theories are related to OMCDs as they both group a set of symbols. The notion of theory supplants the one of dictionary by including altogether, theorems, lemmas and other assertional elements, proofs and their components, examples, exercises and other textbook ingredients, and, a powerful inheritance system (which nevertheless has expressiveness restriction due to the wish to keep a simple morphism from theories to OMCD, see [Maa02, MP03]).

Definitions of symbols and assertions make use of two (possibly combined) elements: the FMP elements (Formal Mathematical Properties) which is expressed in term of a symbolic formal expression, and the CMP elements (Commented Mathematical Properties) which is expressed using a mixture of natural language text and OpenMath or MathML formulas. Thus, for natural language mathematics, one must choose between retaining knowledge of the precise phrasing and presentation chosen by the mathematician, or capturing more of the structure via conversion to symbolic formula. Of course, one could do both like in our example above, keeping the uninterpreted natural language while adding a symbolic formula, but then the format does not support verifying the mutual consistency. Generally, one does not expect the formal checking of mathematics encoded in OMDoc.

The structure of an OMDoc document is more rigid than with the mainstream mathematical typesetting systems. This is due to the fact that the semantical aspect of the language leads the structure of the document. Formating a proof with OMDoc is equivalent to following the proof structuring as proposed in [Ler83, Lam95]. This issue was tackled in the latest version (version 1.2) of OMDoc presented in [Koh06].

2.4.3 Theorem Provers

Mathematical discourse is interested in describing knowledge and demonstrating the truthfulness of what it describes. This notion of truth is central in formal mathematics where the aim of any proof is to be convincing enough to bring a claim to a level of an admitted fact. To meet this goal for truth one obviously needs to avoid falsity and, by extent, inaccuracy in the logical construction. The search for truth can then be replaced by a search for exactitude in all details so that, from primarily agreed facts, one can demonstrate, without any doubt (and without any language subversion) and with precise and conform rules, the truthfulness of a claim. A wide variety of foundations for mathematics has been achieved since the the ancient Greek mathematicians. Since the rise of computers, some of these old and new foundations have been refined or invented to benefit from the computational capabilities of machines [Bar04, BW05].

In 1993, the QED project was initiated by R.S. Boyer with the goal of providing an accessible, comprehensible and formalised computer-based encyclopedia of mathematics. The QED manifesto [CAD94] describes the reasons, the objectives and the steps for the realisation of this system. Seemingly, the project vanished due to failure to concretise but its objectives ares still in the minds of the theorem prover community.

In [Wie03] and its continuation [Wie06], F. Wiedijk lists and compares most of the available theorem provers[8]. In [Har96b], J. Harrison sorts theorem provers into two categories: *declarative* and *procedural* according to their proof style. In [Wie01], F. Wiedijk reuses this classification.

2.4.3.1 Procedural approach

In the procedural proof style, one firstly defines a property to prove. This property is the only element of the starting proof state which consists of the set of proof obligations to fulfill. The proving process consists in applying tactics to the remaining proof obligations. These tactics either provide the proof of a proof obligation or decomposes this one according to its construction. The initial property is considered proved when no proof obligations are left. Listing 2.4 shows a Coq proof of Figure 2.4's theorem. In this listing, line 82 states the theorem to prove, and lines 83–96 contain the sequence of tactics to reach a complete proof.

[8] Visit `http://www.cs.ru.nl/~freek/digimath/` (last visited 2007-04-23) for an ongoing attempt to draw an exhaustive list of systems implementing "mathematics in the computer".

```
Theorem irrationalRsqrt2: (irrational (sqrt (S (S O)))).
Red.
Intros p q H; Red; Intros H0; Case H.
Apply (main_thm p).
Replace (Div2.double (mult q q)) with (mult (S (S O)) (mult q q));
 [Idtac | Unfold Div2.double; Ring].
Case (Peano_dec.eq_nat_dec (mult p p) (mult (S (S O)) (mult q q))); Auto;
 Intros H1.
Case (not_nm_INR ? ? H1); Repeat Rewrite mult_INR.
Rewrite <- (sqrt_def (INR (S (S O)))); Auto with real.
Rewrite Rabsolu_right in H0; Auto with real.
Rewrite H0; Auto with real.
Cut ~ <R> q == R0; [Intros H2; Field | Idtac]; Auto with real.
Apply Rle_ge; Apply Rlt_le; Apply sqrt_lt_R0; Auto with real.
Qed.
```

Listing 2.4: A part of a Coq proof of Figure 2.4's example

Part of Laurent Théry's Coq proof from F. Wiedijk's [Wie06].

There exists numerous procedural proof systems. Among others we mention: Coq[9] [Log06, BC04] based on the calculus of inductive constructions [CH88], PVS[10] [ORS92] based on classical typed higher-order logic, Isabelle[11] [NPW02] which does not hard-wire logics into the system, but offers a meta logic (Isabelle/Pure) to formulate them, Ωmega[12] [BCF+97] based on the CORE calculus [Aut03, Aut05], and Theorema[13] [BCJ+06] based on high order predicate logic.

2.4.3.2 Declarative approach

In the declarative proof style, a proof is a succession of statements (and not tactics). The main declarative proof systems are: Automath (that we presented in Section 2.1.1) and Mizar[14] [Try80, Rud92, RT99] based on Tarski-Grothendieck set theory [Tar39, HK68, Try89]. Mizar is being developed by the Mizar project lead by A. Trybulec since 1973. Listing 2.5 shows a Mizar proof of Figure 2.4's theorem. The syntax of Mizar mimicks natural language mathematics which makes it quite readable by non experts.

[9] http://coq.inria.fr/

[10] http://pvs.csl.sri.com/

[11] http://www.cl.cam.ac.uk/research/hvg/Isabelle/, http://isabelle.in.tum.de/ and http://mirror.cse.unsw.edu.au/pub/isabelle/

[12] http://www.ags.uni-sb.de/~omega/

[13] http://www.risc.uni-linz.ac.at/research/theorema/software/

[14] http://mizar.org/

```
theorem :: Phytagoras' theorem
  sqrt 2 is irrational
proof
  assume sqrt 2 is rational;
  then consider a,b being Integer such that
  A1: b <> 0 and
  A2: sqrt 2 = a/b and
  A3: a gcd b = 1 by Local_TH2;
  A4: b^2 <> 0 by A1,SQUARE_1:73;
  0 <= 2 by NAT_1:18; then
  2 = (a/b)^2 by A2,SQUARE_1:def 4
       .= a^2/b^2 by SQUARE_1:69;
  then A6: a^2 = 2*b^2 by A4,XCMPLX_1:88;
  then a^2 is even by ABIAN:def 1;
  then a is even by PYTHTRIP:2;
  then consider c being Integer such that
  A8: a = 2*c by ABIAN:def 1;
  A9: 4*c^2 = (2*2)*c^2
           .= 2^2*c^2 by SQUARE_1:def 3
           .= 2*b^2 by A8,SQUARE_1:68,A6;
  2*(2*c^2) = (2*2)*c^2
           .= 2*b^2 by A9;
  then 2*c^2 = b^2 by XCMPLX_1:5;
  then b^2 is even by ABIAN:def 1;
  then b is even by PYTHTRIP:2;
  then ex j being Integer st b = 2*j by ABIAN:def 1;
  then 2 divides a & 2 divides b by A8,INT_1:def 9;
  then A11: 2 divides a gcd b by INT_2:33;
  a gcd b = 1 by A3,INT_2:def 4;
  hence contradiction by A11,INT_2:17;
end;
```

Listing 2.5: A part of a Mizar proof of Figure 2.4's example

Part of K. Retel's Mizar version of the proof, inspired by F. Wiedijk's one from [Wie06].

In [Har96a], J. Harrison proposes to use Mizar's declarative proof style as an alternative input for the HOL system (which is based on a procedural language). This idea was later followed in the development of the Isar formal proof language [Wen02, Wen99, WW02] as an alternative interface for the Isabelle system [NPW02]. In [Wie01], F. Wiedijk provides an interesting description of attempts to interface procedural TPs with a declarative style of writing proofs. An application of this idea has involved the development of a declarative proof language for Coq [GW03, Cor07].

2.4.3.3 Formal Proof Sketch

As a matter of fact, theorem provers have been designed to target verified proofs. To reach this target they check every single logical step needed for any assertion. But this key-feature of theorem provers does not suit partial proving or proof planning. How can theorem provers be used for these tasks? Lighter TPs have been proposed as a solution by F. Wiedijk in [Wie04]. He defines Formal Proof Sketch (FPS) a proof language whose validation is attenuated to permit proof sketching. This notion could be defined for any proof language. He defines a Mizar FPS document to be a Mizar document with holes (corresponding to Mizar checker *4 error). Listing 2.6 shows a Mizar FPS proof of Figure 2.4's theorem.

```
environ
 vocabulary SQUARE_1, IRRAT_1, ARYTM, RAT_1, INT_1,
   ARYTM_3, MATRIX_2,ORDINAL2;
 notations ORDINAL2, ORDINAL1, XREAL_0, XCMPLX_0, INT_1,
   INT_2, SQUARE_1, RAT_1, IRRAT_1, ABIAN;
 constructors SQUARE_1, RAT_1, INT_2, POWER, ABIAN, XCMPLX_0;
 registrations REAL_1, XREAL_0, NAT_1, INT_1, SQUARE_1;
 requirements SUBSET, NUMERALS;
begin

theorem :: Phytagoras' theorem
  sqrt 2 is irrational
proof
  assume sqrt 2 is rational;
  consider a,b being Integer such that
  a^2 = 2 * b^2 and
  a,b are_relative_prime;
  a^2 is even;
  a is even;
  consider c being natural number such that
  a = 2*c;
  4*c^2 = 2*b^2;
  2*c^2 = b^2;
  b is even;
  thus contradiction;
end;
```

Listing 2.6: A Mizar FPS proof sketch of Figure 2.4's example

Mizar FPS version reproduced from [Wie06]; environment due to K. Retel.

We notice the relative similitude between the identifiable reasoning steps in Figure 2.4's proof and in Mizar FPS' code. This approach reduces the expense of computerization via formalization (and also loses the certainty of full formalization), but does not appear to greatly improve control over presentation, phrasing and support for semantics-based manipulation.

2.4.4 Other systems

There exists many other systems for mathematics on computer. Computer algebra systems have been for decades widely used by the mathematical community. Axiom[15] and Maxima[16] are among the open-source computer algebra systems. Maple[17], Mathematica[18] and Matlab[19] are among the most popular proprietary ones. We mention particularly the FoCal system[20] [PDH02, PD02, Pre03] developed by the FoCal project lead by T. Hardin. FoCal combines specification, algorithm and proofs for performing verified computations. In 2002, together with V. Prevosto and R. Rioboo, we designed FoCal's documentation system [Maa02, MP03].

Another manner to computerise mathematics involves scanning images of textbooks using optical character recognition. Systems specialised in mathematical texts recognition have gained more accuracy and precision by tackling semantic and contextual recognition [NS06, KSSS06, RRSS06].

[15] `http://www.axiom-developer.org/`

[16] `http://maxima.sourceforge.net/`

[17] `http://www.maplesoft.com/products/Maple11/`

[18] `http://www.wolfram.com/products/mathematica/`

[19] `http://www.mathworks.com/products/matlab/`

[20] `http://focal.inria.fr/`

Chapter 3

Faithful Computerisation of Mathematics

In this chapter we introduce the founding ideas which, we believe, offers an encoding for mathematics which pertain formulas and natural language as well as pertain unambiguous grammatical constructions.

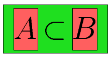
3.1 A Flavor of MathLang's CGa and TSa

The easiest way to get the intuition of what is a TSa-CGa computerisation is to see its principles in action on some examples. In this section, we show how common mathematical constructions are identified in TSa. We then show some concrete examples taken from existing mathematical documents. Here, we only show which texts elements are to be identified in CGa but not precisely how CGa and TSa are defined nor how they are encoded. See Chapter 4 for the definition CGa, Chapter 5 for the definition of TSa and Chapter 6 for their implementation.

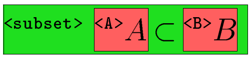
When considering only the justification and argumentation aspect, a mathematical text is typically a succession of deductions derived from facts. These deductions are brought forward by rational arguments which are composed by some assumed facts or standing results from earlier parts of the text. The material of such deductions are concrete or abstract notions and objects, that could often be defined within the mathematical text. This manner of understanding the composition of mathematical discourse, CGa inherits it from and shares it with N.G. de Bruijn's MV (see Section 2.1) and WTT (see Section 2.2).

Example 7 *Let us consider the following formula. "$A \subset B$" Which information* *is it hosting? Let us investigate it in details.*

In the sentence "$A \subset B$", A and B are two identifiers. They have surely been introduced before the sentence was stated. If we assume that A and B are sets, the sentence introduces a new fact: one set is the subset of the other. Here $\subset$ is a subset relation between sets. In CGa, we denote *these three sub-expressions (A, B and $A \subset B$) with their grammatical role (respectively set, set and statement). Here is a view of this sentence with one coloured box per identified piece. See Table 3.1 for the colour coding we use.*



The grammatical role is extended with the denotation of the boxed chunk. An attribute is input by the user. This attribute provides an unambiguous identifier used to explain the expression contained in the box. Here is a version of this sentence with these attributes printed on the top left corner of each box in between the angle brackets `<` *and* `>`*.*



This sentence could therefore be interpreted as a `subset` *statement on the two sets* `A` *and* `B`*. This text with TSa boxes shows the way CGa is to be input by mathematicians (see Chapter 5). A checker analyses the well-grammatical formation of the text. In the case of our example, the checker makes sure that the three identifiers* `A`*,* `B` *and* `subset` *have been properly introduced and are used in accordance to their definition. See Chapter 4.2 for a complete description of the CGa type system.*

Example 8 *Let us see now an example with natural language test. We consider here the sentence: "For every natural number n, $n + 1$ is also a natural number".*

This sentence is an expression which states that the result of adding one to a natural number is a natural number. The implicit meaning of "For every" is the universal quantification $\forall$. It introduces the variable n. This variable represents any natural number. The operator $+$ is applied to n and 1. The result is said to belong to the same type of objects as n . The abstract object a natural number refers to one notion (natural number) or two notions (natural and number) presumably defined earlier or simply assumed to be known by the reader. If we understand number as a type of object (we call them nouns in CGa), then natural could be seen as a refiner for this type (that we call adjectives). It is important to notice the two slight different uses of "being a natural number". In the first part of the sentence

Terms	Sets	Nouns	Adjectives	Statements
Declarations	Definitions	Contexts	Steps	

Table 3.1: Colour coding for CGa's grammatical categories
A description of the grammatical categories could be found in Table 5.1.

it states that n belongs to the type "natural number" but also participates into the declaration of n. As in the second part of the sentence, $n+1$ is an object stated to be of the same type "natural number" but with no declaration of identifier here. A symbolic version of this sentence would look like:

$$\forall n : natural\ number,\ n+1 : natural\ number$$

Here is a CGa version of this sentence.

For every natural number n, $n+1$ is also a natural number

And the version printed with attributes.

<forall>For every <n> <#refinement> <natural>natural <number>number <n>n, <is> <+><n>n + <1>1 is also a <#refinement> <natural>natural <number>number

These two sentences cause no difficulty for a human reader with basic mathematical knowledge. Even taken out of their contexts, their meaning is easily inferred. It is exactly this inference (i.e. the mind action to extract the implicit meaning from such sentences) that we would like to conduct in CGa. Such sentences left as they were (i.e. without explicit grammatical interpretation) are meaningless for a computer software. The extra information we added in our explanation about variables' scoping, belonging of objects to a kind or a set, arity of symbols, and actions – new fact or object introduction – performed by a piece of text, are relevant for a comprehensible encoding of mathematical text.

We shall make it clear here that this extra information is not meant to be purposeful in the sense that they do not make explicit a calculation or a deduction, they simply express some knowledge already contained in the CML version of the text. The difference lies in the fact that they are expressed in a computerised way and therefore in an explicit manner. This extra information is simply grammatical and does not have for now a purpose for a specific computation or proof validation.

Nevertheless, as we already mentioned, this grammatical information could be validated by our checker (see Section 5.1.5 for an example). What we are interested in is the tangible grammatical information we could make explicit when encoding mathematics.

3.2 Needs and Alternatives for Computerising Mathematics

3.2.1 In search for a suitable language for computerising mathematics

In this section we discuss the reasons for which the major languages for mathematics on computer did not fulfill our needs. See Figure 3.1 for a diagrammatic comparison of these languages and systems with MathLang-CGa-TSa.

3.2.1.1 Why not using CML?

The first question to ask is, *Why do we need something else than CML?* There are probably good reasons for CML to be widely used. The main answer to this question is that CML, due to the fact that it is based on natural language, makes any automatic reasoning difficult. What we call CML is the linguistic machinery that mathematicians use to communicate mathematical content. N.G. de Bruijn defined it in [dB87] as a *"very precise mixture of words and formulas"*. Let us see what makes CML be so valuable. Here is a non-exhaustive list of *advantages* of CML.

- *Expressive.* CML is expressive because it is a rhetoric for mind thoughts which combines abstract and concrete notions.
- *Time-honored.* CML is rooted in long lasting traditions.
- *Universal.* CML is approved and used by mathematicians.
- *Diversified.* CML accommodates many branches of mathematics.
- *Adaptive.* CML is not standardised and could be featured with the authors own style.

Some disadvantages of CML in regard to computation are shared with natural language. Even though CML is known as a *precise* natural language, this is not enough to make it easy to compute. Here is a non-exhaustive list of *disadvantages* of CML.

- *Ambiguous.* CML is based on natural language, thus informal and ambiguous.
- *Imprecise.* CML is incomplete and appealing on the reader's intuition.
- *Automation-unfriendly.* CML makes any computation highly dependent to artificial intelligence..

3.2.1.2 Why is MV not enough?

MV was given its name because its author, N.G. de Bruijn put himself in the situation of defining the formal layout of the mathematical vernacular. We insisted in our presentation of MV (see Section 2.1) that its design by N.G. de Bruijn was highly influenced by two decades of proof checking in the Automath project at Technische Universiteit Eindhoven.

This filiation gave to MV a notion of typing which allows grammatical validation with its statements, substantives, names and adjectives (see Section 2.1.2.3). Nevertheless, MV imposes with the rules in [dB87, §12] a hierarchy of substantives (see Section 2.1.2.6) which is a too strong correctness requirement at this level.

Similarly, MV inherited from Automath an explicit notion of context. Contexts in MV are important because they make explicit the scoping of identifiers. Nevertheless, the direct correlation between variables presented in a context and the parameters for a constant being defined (see Section 2.2.1) constrains the user to a level of precision he did not choose. In a formal system this requirement is reasonable because the value held by the instance of a constant is calculated with δ-reduction (inlining of the definiens, see Sections 2.1.1.2.2 and 4.3.3). When a calculation towards proof is not yet an issue, this requirement is misplaced. The parameters listed by the mathematician in the definition of a symbol is important for the representation that this mathematician wants to give to his new symbol. In [vBJ77a], AUT-SYNT is presented as a solution for shortening the number of parameters using type inference to guess some of them. If both an identifier and its type are among a parameters list, one can omit the identifier's type because this information could be inferred by the system. Nevertheless, this solution does not give full liberty to the user to decide which parameter is or is not of a sufficient importance to appear as such. Some hidden parameters may well exist but they should remain as such.

Summarising, MV has some roughness in the definition of its substantives and also has some rigidity in the formation of contexts. MV is close to the usual way mathematicians organise knowledge but does not give the possibility to remain close to informal mathematics in natural language form.

3.2.1.3 Why is WTT not enough?

WTT (see Section 2.2) solves the problem of the logical constraints on substantives by defining its nouns (WTT's descendant of MV's substantives) with weaker rules (see Section 4.4). However the correlation between variables in the context at the time of a definition and sets of parameters of the constant being defined remains.

Another restriction appeared to us with WTT binders. The first grammatical meaning of a binder is to abstract on a variable in an expression. A binder is a constructor that makes use of abstraction to give a particular meaning to an expression. There are different kinds of binders. The most commonly used are the quantifiers. They introduce an abstract object that is either known to exist ($\exists$) or known to have the property of any existing object from a collection of objects ($\forall$). Another widely used kind of binder is the one that ranges the value of the variable is introduces (e.g. Σ, lim, $\cup$, $\int$ are examples of such binders). See Section 2.2.1 for an extended discussion over binders. WTT has a category of identifiers named binders. In contrast to other identifiers (variables and constants), they are predefined in the language and the set of binders is fixed. We consider it to be a too restrictive approach which makes some mathematical assumptions on which binders should be used and how. There is a need for a more generic approach to binders, where the author can define freely his own binders.

Both MV and WTT also lack some structural construction to decompose a document into comprehensive parts and to organise the contexts. In MV, the notion of *block* and *flag* (see Section 2.1.2.2 and [dB87, §4.3–4.5]) were introduced but were not part of the language. Flags in MV and WTT are only a syntactic sugaring for context representation.

The points that we discuss in depth in Section 3.3.1.3 relate to the inability of WTT to express the relation between an object and its components (if it has some). For instance, the fact that a circle has a center is semantically important but this fact has also some grammatical incidences in the way the notion of circle could be used.

Last but not least, WTT is suitable for a symbolic description of the mathematical reasoning but it implies to express this reasoning in a form difficult to read. In this sense, WTT fails as a communication medium.

3.2.1.4 Why is LaTeX inappropriate?

LaTeX (see Section 2.4.1.2) is a typewriting language dedicated to print views of mathematical and scientific texts on paper or on computer screen. LaTeX is a version of CML where some structural aspects have been made precise (for instance with the use of LaTeX environments) and where formulas have been flattened into a one-dimension syntax. Even if some work has been done to analyse the semantical content of LaTeX formula and texts [Pad03, Koh04], LaTeX remains imprecise, permissive and owns every CML disadvantage.

Summarising, LaTeX is undoubtedly a wonderful typesetting system for processing and editing mathematics. It is flexible and has an international community that extended it with numerous packages for language-specific domain-specific uses.

As automatic semantical recognition is concerned, LaTeX is intrinsically based on natural language and therefore shares the disadvantages of CML (see Section 3.2.1.1).

3.2.1.5 Why is OMDoc not suitable?

OMDoc, as we explained in Section 2.4.2.2, offers a format to construct theories by defining their components and their semantic relations. It uses either MathML or OpenMath for the encoding of formula. It also opens the doors to external formal representation by using the extensibility of XML. Nevertheless we believe that this middle ground language misses features needed to capture some important aspects of mathematical texts, especially the portion written with natural language. A systematic encoding of natural language into OMDoc objects is not possible, in particular there are no specialized representatives for nouns or adjectives in OMDoc. Natural language makes most of OMDoc content automation-unfriendly for dealing with reasoning expressed in natural language.

OMDoc offers wide possibilities to incorporate additional content in OMDoc documents but the overall structure of a document has to follow the decomposition into OMDoc theories (one of the reason was originally to support morphism from OMDoc to OMCD). CML follows such well organised theory structure only occasionally which makes the encoding of CML into OMDoc a partial translation instead of a computerisation.

Maybe the main feature missing in OpenMath and OMDoc is user-support automatic tools to validate documents and formulas encoded using these languages. As we already discuss in Section 2.4.2.1 some systems were early designed for validating OpenMath encodings but, to our knowledge, no implementation has been developed. Such feature is also missing for OMDoc. The only validation available for OpenMath and OMDoc are XML Schema validations which are purely syntactic. When OpenMath/OMDoc is used as format by mathematical systems (such as Ωmega[1] [BCF+97], ActiveMath[2] [MBG+03], MathWeb Software Bus[3] [ZK02], MBase[4] [KF01] or Numerical Algorithms Group[5] products [CD03]), a generic validation is not essential as these systems perform their own verifications of the data they treat, but the lack of generic automation rewarding such costly encoding makes the extensive use of OpenMath/OMDoc by working mathematicians unlikely.

3.2.1.6 Why is full formalisation casted aside?

TPs (see Section 2.4.3) have made a tremendous contribution to computerizing mathematics, providing frameworks in which a full formalization can be written and verified automatically. However, they do not support important issues of mathematical text, such as control over presentation and phrasing and processing of the semantic structure. Furthermore, because full formalization is very expensive in human time, most mathematical texts are unlikely to be fully formalized, but might still benefit from some form of computerization.

Full formalisation requires choices that we do not want to make prior to any authoring work. When formalising, some choices have to be made before any encoding could be done because the encoding or formalisation depends on these choices.

- The choice of a *logical framework* implies models for representation and deductions. There is no such thing as “the best formalism”. Depending on the complexity of the system, one might consider the need to work in intuitionistic logic where others are satisfied with classical logic. The level of expressiveness and complexity is often decisive to choose between first or high order and to choose between propositional or predicative logic. The choice

[1] `http://www.ags.uni-sb.de/~omega/`
[2] `http://www.activemath.org/`
[3] `http://www.ags.uni-sb.de/~jzimmer/mathweb-sb/`
[4] `http://www.mathweb.org/mbase/`
[5] `http://www.nag.co.uk/`

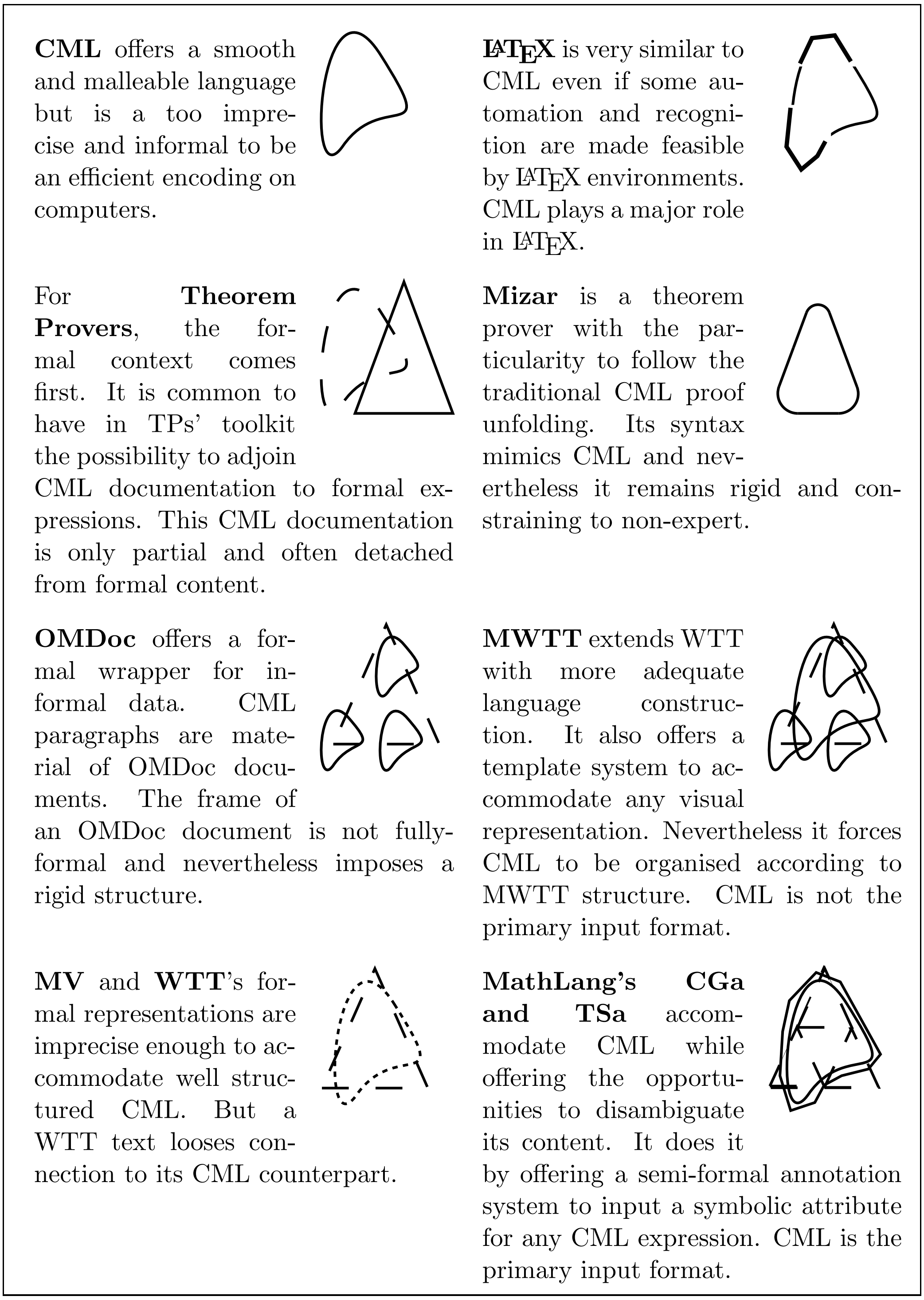
of an underlying logic also goes with the choice of a foundation: set theory, type theory, category theory.

- The choice of an *implementation* is highly dependant on the logical framework. Choosing or combining induction, equivalence and classes, equational reasoning, also depends on the orientation of the formalisation, i.e. towards a proof only or a proof together with computational algorithms.
- Finally (or quite often firstly as people tend to have their favorite system and to accept its choices) the choice of a *proof checker* accordingly. Systems like Coq, Mizar, Isabelle, HOL, PVS, Nuprl are among the notorious theorem provers (see Section 2.4.3). They have made a tremendous contribution to computerizing mathematics, providing frameworks in which a full formalization can be written and verified automatically. However, they do not support important issues of mathematical text, such as control over presentation and phrasing and processing of the semantic structure.

3.2.1.7 Why are FPS and Mizar FPS a partial solution only?

The Mizar language (see Section 2.4.3.2) is a theorem prover with a syntax designed to mimic natural language. A mathematician non-expert in Mizar is likely to make his way into a Mizar article or at least to follow some of its reasoning structure. In that sense, Mizar shares our goal to offer a computerized alternative to CML. Nevertheless being able to write a Mizar article is another matter. The Mizar syntax is strict and needs to be learned properly to envision any Mizar writing. We refer to two articles describing the Mizar syntax and how to parse it [CG04]. In order to use the full power of Mizar, one also needs to have a good knowledge about the Mizar Mathematical Library (MML)[MML].

The Mizar system is aimed at fully formalising mathematics. In that regard Mizar falls into the previous category of Section 3.2.1.6 titled “full formalisation”. But F. Wiedijk introduced FPS (see Section 2.4.3.3) as a lighter version for TPs. This approach reduces the expense of computerization via formalization (and also loses the certainty of full formalization), but does not appear to greatly improve control over presentation and phrasing and support for semantics-based manipulation.



CML offers a smooth and malleable language but is a too imprecise and informal to be an efficient encoding on computers.

LaTeX is very similar to CML even if some automation and recognition are made feasible by LaTeX environments. CML plays a major role in LaTeX.

For **Theorem Provers**, the formal context comes first. It is common to have in TPs' toolkit the possibility to adjoin CML documentation to formal expressions. This CML documentation is only partial and often detached from formal content.

Mizar is a theorem prover with the particularity to follow the traditional CML proof unfolding. Its syntax mimics CML and nevertheless it remains rigid and constraining to non-expert.

OMDoc offers a formal wrapper for informal data. CML paragraphs are material of OMDoc documents. The frame of an OMDoc document is not fully-formal and nevertheless imposes a rigid structure.

MWTT extends WTT with more adequate language construction. It also offers a template system to accommodate any visual representation. Nevertheless it forces CML to be organised according to MWTT structure. CML is not the primary input format.

MV and **WTT**'s formal representations are imprecise enough to accommodate well structured CML. But a WTT text looses connection to its CML counterpart.

MathLang's CGa and TSa accommodate CML while offering the opportunities to disambiguate its content. It does it by offering a semi-formal annotation system to input a symbolic attribute for any CML expression. CML is the primary input format.

Figure 3.1: Approaches for computerising/formalising mathematics
Informal data is represented by blobby shapes (▷). Computerized and more formal data is represented by triangles (△). By curves we represent the "aesthetic" and expressive but ambiguous aspect of human language. By straight lines we represent the precision but roughness of formal languages

3.3 The approach of MathLang's TSa and CGa

3.3.1 Grammatical representation

The computerisation of mathematical texts requires capturing the knowledge contained in the text, in a form as close as possible to the way the text is constructed. As we explained in Section 2.3.2, the representation of mathematical knowledge should not be a complete formalisation to accommodate the mathematicians' needs. We need to capture in a computerisation the intrinsic role that each text piece plays in the building of the text's mathematical knowledge. For this purpose we define a *grammar for mathematical justification.* This grammar is not a grammar for natural language. It represents how justification elements are combined. In comparison to natural language grammars, this one is rigid because it represent a core-language for mathematical justification. We see in Sections 3.3.2, 3.3.2 and 5.2 how we make the bridge between this justification grammar and natural language texts. We follow here N.G. de Bruijn's idea of defining a generic grammar for justification.

> Putting some kind of order in such complex set of habits as the mathematical vernacular really is, will necessarily involve a number of quite arbitrary decisions. The first question is whether one should fell free to start afresh, rather than adopting all pieces of organization that have become more or less customary in the description of mathematics. We have not chosen for a system that is based on what many people seem to have learned to be the only reasonable basis of mathematics, viz. classical logic and Zermelo-Fraenkel set theory, with the doctrine that "everything is a set". Instead, we shall develop a system of typed set theory, and we postpone the decision to take or not to take the line of classical logic to a rather late stage. *[dB87, §1.5]*

This grammar for mathematical argumentation and justification is a distinguished aspect of the MathLang project. We named it Core Grammatical aspect (CGa). We discuss in the following the components of this aspect. See Chapter 4 for its complete definition.

3.3.1.1 Mathematical texts ingredients

Let us attempt to characterise the components of mathematical texts that participate into the construction of its justification knowledge. We do not consider this characterisation as fixed. This characterisation is an offspring of research and reflection carried out by N.G. de Bruijn (see Section 2.1), R. Nederpelt and F. Kamareddine (see Section 2.2), the MathLang project (see Section 2.3) and in this thesis. We extended their views on the grammar of a computerised mathematical vernacular with our reflections following some translation and encoding experiments, see Section 3.4 for more details on our experience driven methodology and Chapter 5 for more details on our encoding experiments.

3.3.1.1.1 A succession of facts If we focus on the justification aspect, a mathematical texts is a sequence of facts brought one after the other. A justification is a linear succession of facts leading to one main fact such as the proof a theorem. These lines of justifications are combined to form a bigger justification usually with a certain global coherence. The smallest justification element is an expression representing a proposition such as sentences like "n is odd" or "the function f converges".

⇒ Successions of justifications are expressed in terms of *steps* (atomic or block of steps)[6].

3.3.1.1.2 A dynamic vocabulary A specificity of mathematical discourse in comparison with other natural languages is the propensity to define new notions, symbols or shortcuts for complex expressions. The lexicon of the mathematical vernacular is dynamically extended. Sentences like "let n be a natural number", "$f(x) = x^2$" or "we call enumeration functions for a set S the functions from $\mathbb{N}$ to S" declare or define n, f and *enumeration functions*, respectively, for the rest of the document in which they are contained.

⇒ The dynamic introduction of identifiers is represented in terms of *declarations* and *definitions*[6].

3.3.1.1.3 Explicit Boundaries The dynamic extension of the vocabulary of mathematical text is done with some limits. Each identifier, symbol or notion has a well-known coverage in the text. Some identifiers are introduced locally inside an expression such as i in $\Sigma_n^{i=1} x^i$. Some others are declared in a specific paragraph or section, for example in the sentence "let us take y such that $y = x^2$" it is certain that the variable y introduced has only a local importance. A y in another section of

[6] See section 4.1 for a formal definition of our grammar.

the document would probably be completely unrelated to the previous y. Another kind of identifier is produced in a manner that increments the body of mathematical knowledge. These identifiers have no real boundaries restricting their use. Usually an identifier labelled as "theorem" is considered as *fait accompli* for the author.

⇒ Boundaries of identifiers are represented in terms of *local scopings* and *bindings*[6,7].

3.3.1.1.4 An extensive use of abstraction An important aspect of mathematical thinking and by extension mathematical authoring is the use of abstraction. Mathematicians have based their reasonings on abstract objects such as numbers or abstraction such as quantification or the representation of all mathematical objects in term of sets.

⇒ The different forms of abstraction are represented in terms of *terms*, *nouns* and *sets*[6,8].

3.3.1.1.5 Ways of characterisations In mathematics it is customary to define notions and operations in a way that permits an easy reuse. Parametrisation is a common characterisations such as in "$f(x) = x^2$" or in "limit of f in a". Mathematicians make also an extensive use of adjectives to refine meanings. For example the word "isosceles" could be combined with a concrete triangle in "ABC is isosceles" or with the word triangle itself in "isosceles triangle" or with another adjective such as in "isosceles rectangular triangle". Another manner of characterisation is to describe something in terms of its features and characters as in the sentence "A is the centre of circle C".

⇒ We represent the different forms of characterisation in terms of *parameters*, *adjectives* and *characters*[6,9].

3.3.1.2 On nouns and sets

N.G. de Bruijn with MV (see Section 2.1) followed by F. Kamareddine and R. Nederpelt with WTT (see Section 2.2) introduced two fundamental grammatical categories in their systems for computerising mathematical texts. These grammatical categories are the *substantive* later called *noun* and *set*, see Sections 2.1.2.3

[7] See Section 4.3.1.4 for an extended notion of *environment* and *scoping*.

[8] See Section 3.3.1.2 for a discussion over the roles of nouns and sets.

[9] See the discussion on the object oriented structure of mathematical texts in Section 3.3.1.3.

and 2.2.1. According to N.G. de Bruijn, nouns (i.e. substantives) are the traditional manner mathematicians have been used to speak about mathematical objects. Nouns therefore play a key role in the grammar of MV.

> *[...]* MV does not take sets as the primitive vehicle for describing elementhood, but substantives *[...]*. *[dB87, §1.12]*

In his view, sets entered the language of mathematics after being conceived as an abstract universal representation for mathematical objects.

> *[...]* Substantives (like point, number, function) seem to be the things we handle in our natural language, and sets are the things we have learned, more or less artificially, to use instead of substantives.
> *[dB87, §1.15]*

Nouns in comparison to sets are more intuitive. Both nouns and sets, as grammatical categories, are closely related to two fundamental notions in modern mathematics that are the notion of *type* and the notion of *set*.

3.3.1.2.1 Sets In a *set theoretical* approach, everything is considered as a set. Mathematical objects are primarily defined by their set-membership: $x \in y$. This approach which originates from G. Cantor's works [Can32], has been a paradigm shift in mathematical thinking. For D. Hilbert, sets are catchall and therefore he considered that there was no need for properties because properties could be expressed in terms of set membership [HB34, HB39]. For G. Cantor $x \in x$ should be possible and B. Russell discovered that accepting such a statement leads to an antinomy called Russell's Paradox. In an attempt to avoid the Russell's Paradox, type theory was invented.

3.3.1.2.2 Types In a *type theoretical* approach, mathematical elements are characterised by their properties. An element is said to belong to a type or a kind: $x : y$. The first extensive application of type theory to mathematics can be found in A.N. Whitehead and B. Russell's *Principia Mathematica* [WR13], creates a hierarchy of types. This hierarchy avoids Russell's Paradox.

3.3.1.2.3 Grammatical point of view In a grammatical point of view, a set is an expression of whom a mathematical object can be an element. A set could be defined by enumerating its elements or simply be an abstract construct. A type or a kind is an expression to whom a mathematical object can belong. A type correspond directly to generic names of mathematical objects such as “triangle” or “number”. The meaning of a noun holds some criterion for elements to belong to it. These criterion define the noun and could be expressed in term of mathematical statements such as “a triangle is a polygon with three sides”.

3.3.1.2.4 Sets of sets We would like to emphasise the difference between sets in CGa and sets in set theory. In CGa an element of the mathematical discourse is a set at a certain point of a document because some other elements are known to be its member. The notion of a set is closely related to the use, in the text, of set membership. The difference with set theory lies in the fact that a CGa set cannot belong to another CGa set (which is fundamentally possible in set theory). MV and WTT share the same notion of sets.

> Coming to a situation like $a \in b \in c \in d$, the Automath style does not allow to write this as a chain of typings like $a : b : c : d$. If b is a set, then let us write $\downarrow b$ for the substantive “element of b”. The chain becomes $a :\downarrow b,\ b :\downarrow c,\ c :\downarrow d$. *[dB87, §1.12]*

In CGa we currently propose a similar solution which is the use of an operator that could be called `element_of` which transforms a given set into a term.

3.3.1.3 Object-oriented structure for mathematical elements

In this section we introduce the object-oriented features of CGa. We first presented this work in [KMW06]. When experiencing the encoding of the first book of Euclid's *Elements* [Hea56] into MWTT (see Section 4.4), we noticed that the language was inadequate to express simple object characteristics (see Section 5.3.2). This experiment was the starting point for defining CGa grammar, see Chapter 4.

3.3.1.3.1 Object-oriented concepts Some programming language research has focused on allowing organizing programs in a way that seems most natural to the programmers. *Classes* [AC96, Cas97, Bru02] are a way of packaging definitions so that it is easy to obtain not only instances (*objects*) but also multiple distinctly modified and extended variants (*subclasses*) via *inheritance*. *Mixins* [BL92, FKF98] are abstract subclass generators that allow reusing modifications and extensions.

Classes and objects In object-oriented programming, a class is usually defined by a set of *fields* and *methods*. An object is an encapsulated sub-program with an internal state that is an *instance* of a particular class. Classes define the common behavior of a group of objects. Fields are named values associated with each instance, while methods are named operations on the instances.

Inheritance Class inheritance avoids repeating the definition of fields and methods shared by several classes. A new class can be defined by inheriting from an existing *parent* class, and the child's set of fields and methods will by default contain those of the parent.

Mixins With simple class inheritance, to make two classes share a common set of new methods without duplicating the method definitions, the classes must inherit from an ancestor class containing the new methods. This may require radical rearrangement of an existing class hierarchy. To alleviate this problem, mixins are subclass definitions that are parameterized on their superclass, and thus act as functions from classes to classes. When a mixin is applied to a class, this makes a new subclass that adds or redefines fields and methods.

3.3.1.3.2 Object-oriented structure for mathematical text We give here a description of the notions that gain expressiveness when moved to an object-oriented conceptualisation.

Nouns are a description of a family of mathematical objects. A noun would usually refer to a generic description. This description characterises a mathematical object by defining its common specificities and behaviours. This generic description would later be brought into life by taking such object. This newly created object would have all the characteristics that its family shares. A noun therefore characterises a family of objects which corresponds exactly to classes in object-oriented programming languages. A noun differs from a set in its way to bring together elements with some specificities. A set has a more liberal way of considering belonging.

Terms are realisations of nouns. A term being of the kind of a noun has all its characteristics. For example, if we say that "ABC is a triangle", it means that the term ABC is a realisation of a triangle. The notion of a term corresponds to object in object-oriented programming languages.

Adjectives are refiners for nouns and objects. It is commonly understood that an "isosceles triangle" is a triangle with two of its sides equal in length. The adjective "isosceles" extends the characterisation of a "triangle". An "isosceles triangle" is still a "triangle" but a "triangle" is not an "isosceles triangle". Adjectives are equivalent to mixins in object-oriented programming languages.

3.3.2 Document annotations and authoring methodology

As a matter of fact, computerised mathematics have been based either on formalisation or on textual digitalisation. The fact that mathematical knowledge has been supported and communicated with a distinguish version of natural language played an ambiguous/minor role in computerised mathematics.

3.3.3 Ways of integrating natural language

It seems that no theorem prover or mathematical computer-language currently has an infrastructure to provide a direct mapping from a typical natural language mathematical text to its own language but they all have methodologies to offer natural language integration. We group these methodologies into four categories.

Proof code with embedded natural language In a typical formal proof language (see Section 2.4.3) there are facilities to incorporate natural language alongside formal definitions and proofs. Natural language text parts are treated as commentaries in a literate proof document and omitted by the verification. This method uses *structured comments*, similarly to programming languages, for generating a documentation out of a programming code. Among others, Coqdoc and FoCDoc [MP03] are the documentation systems for the theorem prover Coq and the computer algebra system FoCal [PDH02, PD02] respectively.

In a similar fashion, recent developments of intuitive text editors have permitted the arising of plugin-interfacing with theorem provers [AR03, ABFL05, MG06].

Inspired by the well-known notion of literate programming [Knu84b], literate proving [CG06] is at its starting point.

Syntax *à la natural language* Formal languages often suffer from rough syntax and strict grammar. To soften the use of formal languages some efforts have been made to adapt these syntaxes and grammars to suit mathematicians' habits. Some developments have gone far in this direction to obtain formal proof documents that *look like* natural language texts. The main examples are Mizar [Rud92] and Isar [Wen99], but more recently some calculi [AS06, Bro06] were developed in pursuit of the idea of a formal representation for pseudo-natural language.

Semantic Web data model Mathematical natural language is a vague and imprecise language which is unfriendly to computation. Web technologies offer a compromise in the way they encapsulate natural language and extend it with semantic tagging and hyper-linking. OMDoc (see Section 2.4.2.2) is a precursor in this domain.

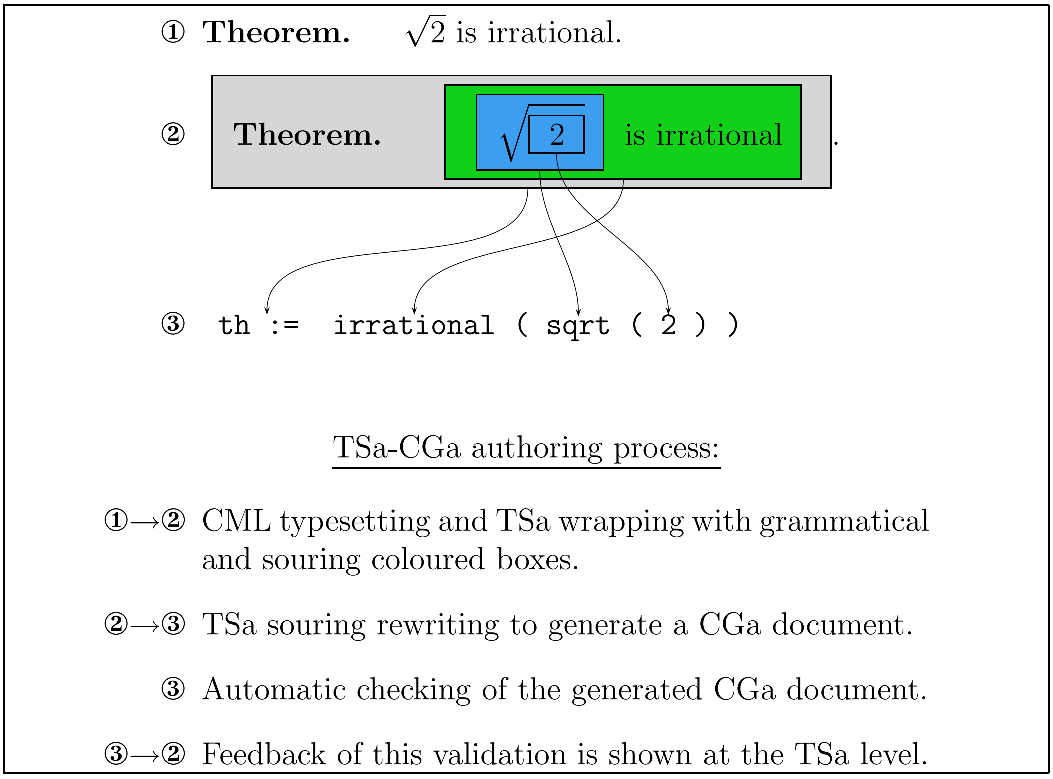
Natural language generator If the starting point is a formally defined language then a natural language representation of the formal content can be produced. The proof assistant HEΛM [APSS01] has this capability. Furthermore, [KMW04a] and [PZ06] provide facilities to personalise the natural language generated.

3.3.3.1 Natural language as computerised mathematics input

The solution we develop differs from the one listed in Section 3.3.3. We propose to restore natural language as a computerised mathematics input method. For doing so we designed a system to decorate the natural language with formal knowledge. This new approach to authoring natural language texts is presented in Section 5.1. As the natural language text is composed, each word or phrase is placed into a certain grammatical category. This is achieved by annotating the original natural language text either during or after its composition, see a short illustration in Figure 3.2.

3.3.3.2 Accommodating natural language complexity: syntax souring

Natural language is quite liberal with only one rule being that a reader should be able to understand the explanation of the author. It is therefore important to accommodate different authoring styles. Instead of forcing the mathematician to use a strict and fixed language we offered him some tools to explicate the morphology of his own natural language style. We developed the notion of *syntax souring* (as "dual" of syntax sugaring). See Section 5.2 for a description of authoring with syntax souring, and Section 6.4.2 for the formal definition of this notion.


① Theorem. √2 is irrational.
② Theorem. √2 is irrational .
③ th := irrational (sqrt (2))

TSa-CGa authoring process:

①→② CML typesetting and TSa wrapping with grammatical and souring coloured boxes.

②→③ TSa souring rewriting to generate a CGa document.

③ Automatic checking of the generated CGa document.

③→② Feedback of this validation is shown at the TSa level.

Figure 3.2: TSa-CGa autoring process

To illustrate the decomposition of mathematical knowledge with MathLang's aspects, we use the example in ①: the definition of a Theorem which states the irrationality of $\sqrt{2}$. We identify in this sentence the grammatical role of each element of the text: **definition** for the entire sentence, **term** for "$\sqrt{2}$" and "2", and **statement** for "$\sqrt{2}$ is irrational". The mathematician attributes to each element its CGa grammatical role by wrapping it into a coloured box following our colour coding system of Table 3.1. The formal interpretation of this sentence is automatically generated from TSa's ② and is printed in ③ using CGa's abstract syntax as defined in Section 4.1. The identifiers `th`, `irrational`, `sqrt` and `2` are provided by the user as arguments for each coloured box of ②. Note that the CGa syntax used in ③ is not meant to be used by the end-user of MathLang, it is only designed for computerisation purposes. The end-user edits his document using the view offered by TSa, as shown by ② (TSa plays the role of a user interface for MathLang). The internal syntax used in our implementations follows XML recommendations (see Section 6.1.1).

3.3.3.3 Separating grammatical and style aspects

As a result we clearly identified two aspects of mathematical writing. Firstly the grammatical aspect CGa (see Section 3.3.1) for expressing the justification and argumentation knowledge contained in a mathematical document. Secondly the text and symbol aspect (TSa) which makes the bridge between natural language expressions and their interpretation expressed in a less ambiguous form.

3.4 Experience-driven Development

One of the founding principles of the MathLang project, as enunciated in [KW00, KW01] by its initiators F. Kamareddine and J.B. Wells, is the willingness to base its development on experiment on real mathematical documents. In this section we tell the story of the experiments that lead to the design and development of the first two aspects of MathLang (CGa and TSa). At each milestone we refer to the relevant sections and chapters of this thesis.

3.4.1 Refinements of WTT based on *Foundations of Analysis*'s translation

This thesis work therefore started with the translation of a mathematical text into WTT (see Section 2.2). We chose E. Landau's *Foundations of Analysis* [Lan51] because we could expect to make some comparison with L.S. van Benthem Jutting's translation in Automath [vBJ77a]. Our objectives for this translation were to:

- test the expressiveness of WTT,
- to check the feasibility of expressing an entire book in WTT,
- to evaluate the degree of difficulty of such translation
- and to see in effect what such translation brings (in terms of validation and accessibility of the data translated).

During this translation we made some significant changes to the language (see Section 4.4 and [KMW04b]) and we implemented a checker. This language and its implementation shared the name of the whole project MathLang (we now refer to it as MWTT).

Theorem 2

$$x' \neq x.$$

Proof Let $\mathfrak{M}$ be the set of all x for which this holds true.

I) By Axiom 1 and Axiom 3,

$$1' \neq 1;$$

therefore 1 belongs to $\mathfrak{M}$.

II) If x belongs to $\mathfrak{M}$, then

$$x' \neq x,$$

and hence by Theorem 1,

$$(x')' \neq x',$$

so that x' belongs to $\mathfrak{M}$.

By Axiom 5, $\mathfrak{M}$ therefore contains all the natural numbers, i.e. we have for each x that

$$x' \neq x.$$

[Lan51, Ch. 1]

$x : \mathbb{N} \rhd \mathrm{Th2}(x) := x \neq x'$ (24)

Proof Theorem 2 {2.2}

$\boxed{\mathfrak{M} : \mathrm{SET}}$

$\boxed{\forall_{x:\mathfrak{M}} \mathrm{Th2}(x)}$

Ax1, Ax3(1) $\rhd 1' \neq 1$ (25)

(25), (Def Th2) $\rhd 1 : \mathfrak{M}$ (26)

$\boxed{x : \mathfrak{M}}$

$\rhd x' \neq x$ (27)

(27), $\mathrm{Th1}(x', x) \rhd x'' \neq x'$ (28)

(28), (Def Th2) $\rhd x' : \mathfrak{M}$ (29)

$\mathrm{Ax5}(\mathfrak{M}, (26), (29)) \rhd \mathbb{N} \subset \mathfrak{M}$ (30)

(30) $\rhd \forall_{x:\mathbb{N}} \mathrm{Th2}(x)$ (31)

$x : \mathbb{N},\ x \neq 1 \rhd \mathrm{Th3}(x) := \exists_{u:\mathbb{N}} x = u'$ (32)

Figure 3.3: Example of a MWTT translation

The MWTT encoding of the whole first chapter of E. Landau'*Foundations of Analysis* [Lan51] is presented in an appendix to [KMW04b] which is available at `http://www.macs.hw.ac.uk/~mm20/papers/Kamareddine+Maarek+Wells:mkm_symposium-entcs-appendix-2004.ps.gz` (last visited 2007–04–22).

3.4.2 Towards an encoding faithful to original texts

After this experiment of translating in MWTT the entire first chapter of *Foundations of Analysis*, we were able to draw some conclusions. The translation of this chapter into MWTT, as we presented in [KMW04b], and the translation of the same chapter in WTT, as G. Jojgov, R. Nederpelt and M. Scheffer presented in [Sch03] (see also [JNS04, Gel04]), may well be faithful to the original text but this translation is hardly readable by a non-WTT/MWTT-expert and hard for anyone to connect each pieces of the translation with its original form (see Figure 3.3). The newly introduced flags and blocks help the reading but the entire document remain very bitter for mathematicians.

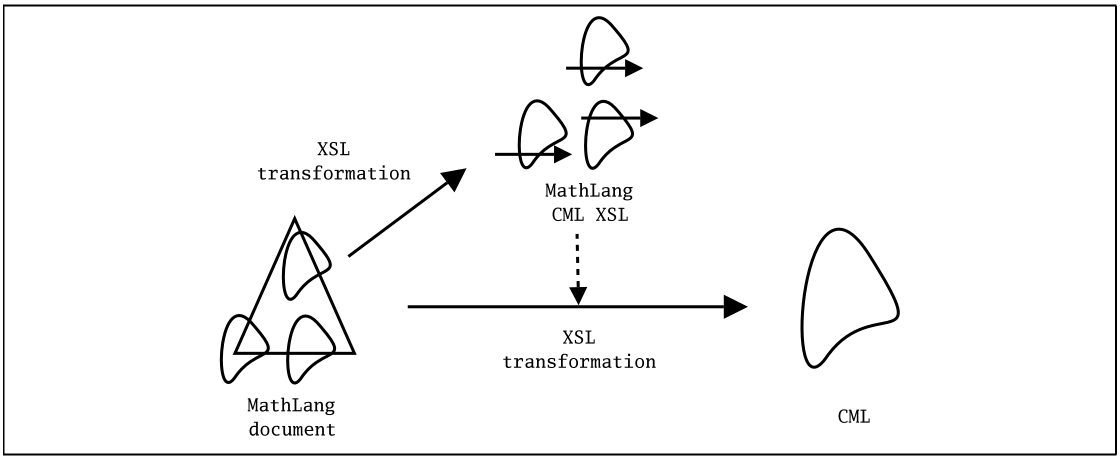


Figure 3.4: Transformation procedure

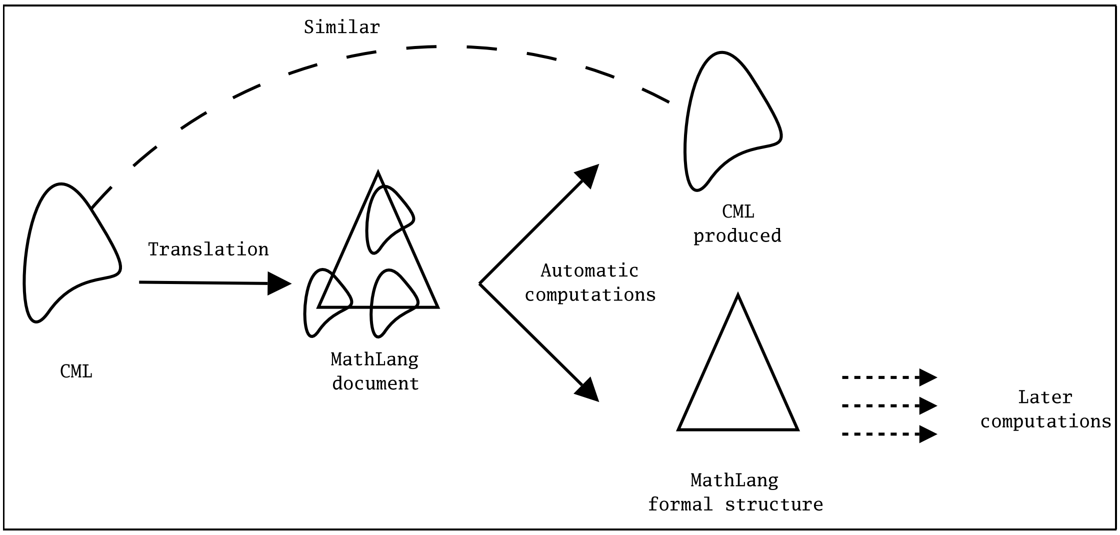


Figure 3.5: Translation process

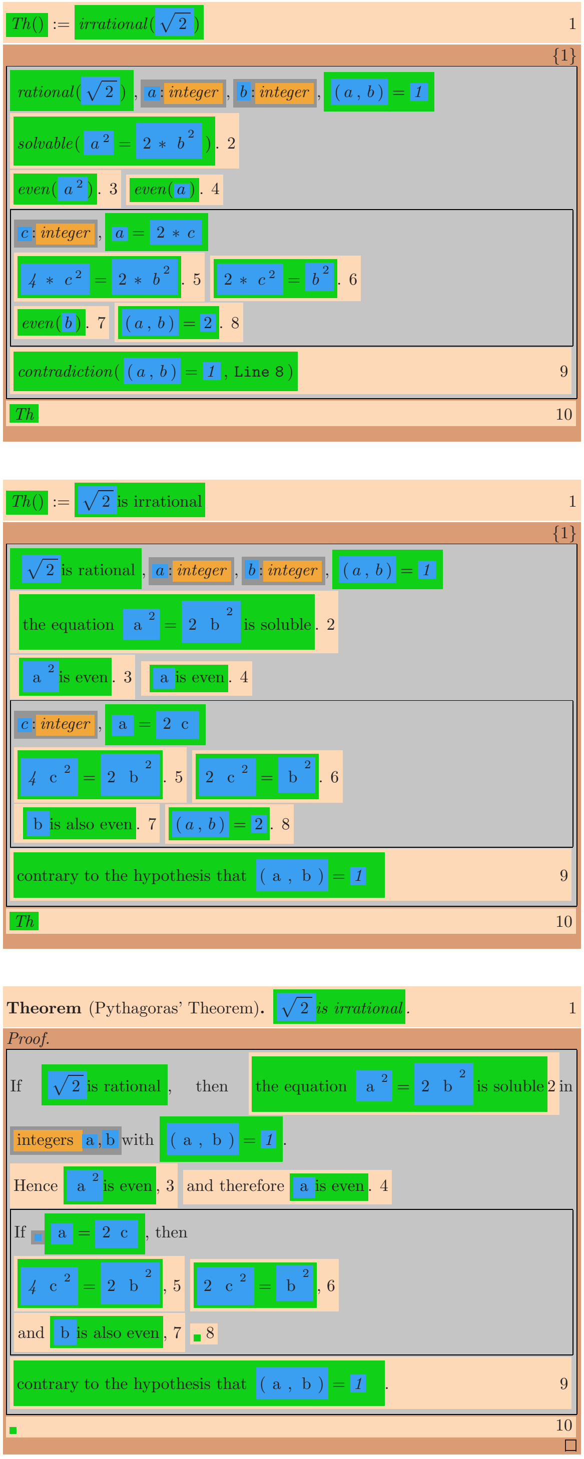
In [KMW04a] we presented a system to allocate local presentation templates which are used to automatically generate, out of a MWTT document, an output view similar to the original natural language text. Figure 3.4 illustrates the transformation procedure that first isolates the local templates to reapply them to the whole document. It results from this application of these templates in a CML document. Figure 3.6 shows three views of the same MWTT encoding of *Pythagoras' proof of the irrationality of* $\sqrt{2}$ by G.H. Hardy and E.M. Wright [HW80, Ch. IV]. This work highlighted the fact that our encoding was flexible in regard to the structure of mathematical texts. If one decorates correctly a MWTT code with natural language templates, it is possible to generate back the original text. Figure 3.5 is an illustration of the MWTT translation process.

Customised views of the MWTT encoding of Pythagoras' proof of the irrationality of $\sqrt{2}$ by G.H. Hardy and E.M. Wright [HW80, Ch. IV] (the example was used by F. Wiedijk in [Wie06] to compare theorem provers):

① Symbolic view

② CML view of symbols

③ CML view



①

Th() := *irrational*($\sqrt{2}$) 1

{1}

rational($\sqrt{2}$), *a* : *integer*, *b* : *integer*, (*a*, *b*) = *1*

solvable($a^2 = 2 * b^2$). 2

even(a^2). 3 *even*(*a*). 4

c : *integer*, $a = 2 * c$

$4 * c^2 = 2 * b^2$. 5 $2 * c^2 = b^2$. 6

even(*b*). 7 (*a*, *b*) = 2. 8

contradiction((*a*, *b*) = *1*, Line 8) 9

Th 10

②

Th() := $\sqrt{2}$ is irrational 1

{1}

$\sqrt{2}$ is rational, *a* : *integer*, *b* : *integer*, (*a*, *b*) = *1*

the equation $a^2 = 2\ b^2$ is soluble. 2

a^2 is even. 3 a is even. 4

c : *integer*, $a = 2\ c$

$4\ c^2 = 2\ b^2$. 5 $2\ c^2 = b^2$. 6

b is also even. 7 (*a*, *b*) = 2. 8

contrary to the hypothesis that (a , b) = *1* 9

Th 10

③

Theorem (Pythagoras' Theorem). $\sqrt{2}$ *is irrational*. 1

Proof.

If $\sqrt{2}$ is rational, then the equation $a^2 = 2\ b^2$ is soluble 2 in integers a, b with (a , b) = *1*.

Hence a^2 is even, 3 and therefore a is even. 4

If a = 2 c, then

$4\ c^2 = 2\ b^2$, 5 $2\ c^2 = b^2$, 6

and b is also even, 7 8

contrary to the hypothesis that (a , b) = *1*. 9

10

□

Figure 3.6: MWTT customised views

3.4.3 Extending the language's expressiveness

Later, when experiencing the translation of the first book of Euclid's *Elements* [Hea56], we faced critical lack of expressiveness from MWTT, and redesigned the language to create MathLang with object oriented features. We explain this move in see Section 3.3.1.3 and we reported this work in [KMW06].

3.4.4 Restoring natural language as a computerised mathematics input method

Following the conclusion (in Section 3.4.2 and 3.4.3) that our encoding was flexible in regard to the structure of mathematical texts, we started to develop a MathLang plugin for the TEXMACS[10] scientific editor. Both our experience gained in developing our template system (see Section 3.4.2) and our experience in building this plugin (see the description of this plugin in Section 6.3.1) opened the possibility to define a system-independent authoring method for computerised mathematics. See Section 3.3.2 for an overview of this method and Sections 5.1, 5.2 and 6.4 for a complete definition of this method.

Conclusion

We presented in this chapter the founding motivations for this thesis' research. We gave an overview of the progress we made throughout this PhD studies' period. We also presented the approach we followed when developing MathLang's CGa and TSa aspects and compared this approach with other existing ones.

[10] http://www.texmacs.org/

Chapter 4

MathLang-CGa, Language Definition

In this chapter we define MathLang's Core Grammatical aspect (CGa). CGa is a formal language derived from MV and WTT which aims at expressing mathematical justification and argumentation. We present its abstract syntax in Section 4.1. In Section 4.2, we provide a type system to check the grammatical well-formation of CGa documents. We recall in Section 4.4 the definition of MWTT's abstract syntax and type system. MWTT is the ancestor of CGa and is a refinement of WTT.

Remark 2 *Throughout this chapter we present and discuss CGa abstract syntax, grammar and type system. It is important to notice that this abstract syntax is not aimed to be used as such by the mathematicians nor software and applications. We discuss in Chapter 5 the interface between the mathematician and the CGa language. We present in Chapter 6 the implementation details and, in particular in Section 6.1 the XML concrete syntax of CGa. We nevertheless implemented a parser for this syntax for experiment purposes and we present its implementation in Section 6.1.2.*

4.1 CGa's Abstract Syntax

In this section we define the abstract syntax of MathLang-CGa. Appendix A.1 contains a summary of the abstract syntax defined here. We are using the `typewriter` font for printing CGa examples. These examples are written in a format that follows the notational conventions described in Section 4.1.1.2. Most material in this section were published in [KMW06].

4.1.1 Notations

4.1.1.1 Metasyntax notational conventions

We list here the key elements of the metasyntax we use to describe the abstract syntax of CGa. This metasyntax is inspired from the Backus-Naur Form notation. In our metasyntax, an arrow on top of a meta-variable represents a sequence of 0 or more meta-variables. For example “$\overrightarrow{exp}$” is a sequence of “*exp*”. The elements of the sequence are separated with a comma “,” in the cases of *ident*-, *category*- and *exp*-sequences. The *step*-sequences elements are separated by semi-colons “;”. The alternatives are separated by the vertical line “ | ”. We use “::=” as the rule symbol. The elements appearing on the right hand side of this symbol are either non-terminal meta-variables (printed in an *italic* font) or the terminal reserved keywords and symbols. The list of these terminals is as follow: “`term`”, “`set`”, “`noun`”, “`adj`”, “`stat`”, “`dec`”, “(”, “)”, “.”, “`label`”, “▷”, “{”, “}”, “:=”, “≪”, “:”, “`Adj`”, “`Noun`”, “`self`”, “`ref`”, “,” and “;” (using the CGa example font, they are printed as “`term`”, “`set`”, “`noun`”, “`adj`”, “`stat`”, “`dec`”, “`(`”, “`)`”, “`.`”, “`label`”, “`|>`”, “`{`”, “`}`”, “`:=`”, “`<<`”, “`:`”, “`Adj`”, “`Noun`”, “`self`”, “`ref`”, “`,`” and “`;`” respectively). We use the back quote sign “‘” to distinguish category variables from other identifiers (see Section 4.1.2.1).

4.1.1.2 Notational conventions

We use the following notational conventions when writing abstract syntax expressions.

1. When an identifier has no parameters we omit the parentheses `(` and `)`. For example, for the instance of an identifier `x` we write `x` in place of `x()` and `x:term` in place of `x():term`.

2. We do not leave double braces in noun and adjective descriptions defined with a block. For example, we write `Noun { a:term; b:term }` and `Adj (n) { c:term; d:term }` instead of `Noun {{ a:term; b:term }}` and `Adj (n) {{ c:term; d:term }}` respectively.

3. We abbreviate category expressions to shorten the syntax of some term, noun and set categories. For example, we write `noun` (respectively `set` and `term`) in place of `noun(Noun{{}})` (respectively `set(Noun{{}})` and `term(Noun{{}})`).

which describes a category expression,

Remark 3 *Note the existence of a category constructor* `noun` *on one hand and of a noun constructor* `Noun` *on the other hand. Category expressions could be build with* `noun` *while noun expressions could be constructed with* `Noun`*. In the following example, three identifiers with one character* `a` *are defined:* `p1` *is a noun,* `p2` *is a term instance of a noun, and* `p3` *is defined as a noun.*

```
{
  p1 : noun ( Noun { a:term } );
  p2 : Noun { a:term };
  p3 := Noun { a:term }
}
```

4.1.2 Language levels

4.1.2.1 Vocabulary level

We firstly define three sets composing the language's vocabulary. These three sets are the set of identifiers, labels and category variables. They are disjoint denumerable infinite sets. We use the meta variables *ident*, *label* and *cvar*. The variables i, l and v range over identifiers, labels and category variables respectively.

ident, i $\in$ $\mathcal{I}$
($\mathcal{I}$ is the denumerably infinite set of identifiers)
label, l $\in$ $\mathcal{L}$
($\mathcal{L}$ is the denumerably infinite set of labels)
cvar, v $\in$ $\mathcal{V}$
($\mathcal{V}$ is the denumerably infinite set of category variables)

An element of the set of identifiers on its own is an identifier in CGa. An identifier attached to an expression expresses the character of this expression. This expression is traditionally a term expression.

cident, *ci*	::=	*ident*	Identifier
	|	*exp.ident*	Character

4.1.2.1.1 Identifier An *identifier* is simply the name given to something. It is generally (and it is case in the concrete syntaxes we implemented, see Section 6.1) a string composed by alpha-numerical characters plus other non-blank and non-reserved symbols. `triangle`, `N`, `Theorem_24`, `x` and `+` are examples of identifiers.

4.1.2.1.2 Character A *character* is a marked feature of a term. Characters are defined by nouns and adjectives. If a term is identified by a noun it inherits everything of this noun. A character corresponds to fields or methods in an object-oriented language jargon. Considering that the noun `circle` has a character `center` representing the center of a circle then it is the case that for every `circle`-term `C`, we have that `C.center` stands for the center of `C`.

4.1.2.2 Category level

The category level describes the way to construct *category* expressions. These special expressions are used when declaring an identifier and the category of its parameters. The category level does not cover all the possible grammatical categories of CGa because it is only concerned with those applicable to the identifier's parameters. The category syntax entry uses grammatical categories' names with lowercase letters.

category, *c*	::=	`term`(*exp*)	Term category
	$\mid$	`set`(*exp*)	Set category
	$\mid$	`noun`(*exp*)	Noun category
	$\mid$	`adj`(*exp*, *exp*)	Adjective category
	$\mid$	`stat`	Statement category
	$\mid$	`dec`(*category*)	Declaration category
	$\mid$	*cvar*	Category variable

4.1.2.3 Expression level

At the expression level we define the way to construct standard expressions of the language.

exp, *e*	::=	$\mathit{cident}(\overrightarrow{\mathit{exp}})$	Instance
	$\mid$	$\mathit{ident}(\overrightarrow{\mathit{category} \mid \mathit{expr}}) : \mathit{exp}$	Elementhood declaration
	$\mid$	$\mathit{ident}(\overrightarrow{\mathit{category} \mid \mathit{expr}}) : \mathit{category}$	Declaration
	$\mid$	`Noun` {*step*}	Noun description
	$\mid$	`Adj`(*exp*) {*step*}	Adjective description
	$\mid$	*exp exp*	Refinement
	$\mid$	`self`	Self
	$\mid$	`ref` *label*	Step reference

4.1.2.3.1 Instance The *instance* of an identifier or of a character is obtained by adjoining this identifier or character to a list of arguments. These arguments correspond to the parameters an identifier of a character may have. For example, `+(sq(x),1)` recalls the definition of plus with the expressions `sq(x)` and `1` as arguments. Similarly, the notion of group is defined as a noun `group` with a character `op` for its a binary operation and if `G` is a group, then `G.op(a,b)` is an instance of the binary operator of `G` applied to `a` and `b`.

4.1.2.3.2 Declaration The *declaration* of an identifier is a valid CGa expression. Declaring an identifier results in providing the number of its parameters, the grammatical category to which they belong respectively and the grammatical category of the identifier itself. There are two ways to express the belonging of an identifier or a parameter to a grammatical category which are either by providing explicitly the grammatical category expressed in term of a category expression (see Section 4.1.2.2), or by providing either a set- or a noun-expression. We named the latter *elementhood declaration.* A declaration is therefore composed by the identifier to be declared, followed by one category or set-/noun-expression per parameter, and finally a category or set-/noun-expression describing the new identifier's grammatical category. The expression `surface(triangle):term` declares a new term-identifier with one argument. This argument is a term of the kind `triangle`. Another example is the declaration of the subset operator: `subset(set,set):stat`, this time we used category expressions only.

4.1.2.3.3 Description As aforementioned, a noun is a denomination for a group of characters. The *description* of a set of characters is an expression which is built by adjoining to the keyword `Noun` (with an uppercase first later) a step (see Section 4.1.2.5). The resulting expression is a noun whom characters are the identifiers declared or defined in the step in question. For example, the expression `Noun { center:term }` is a description fitting the noun `circle` we used in an earlier example.

Adjectives form a category in the CGa grammar. They could cause a noun to be extended and to be more specific. An adjective expression extends a noun expression to form a new noun expression. In that sense, an adjective is a function from noun to noun. An adjective description starts from a noun expression (domain of such function) and describes the additives (in a step) needed to obtain this new noun.

4.1.2.3.4 Refinement The combination of an adjective with a noun is called a *refinement*. For example, the expression `isosceles triangle` is a refinement

of `triangle` using the adjective `isosceles`. The result is a well formed noun-expression
`isosceles triangle`.

4.1.2.3.5 Step reference The last expression alternative is the *step reference* to a previously labeled step. For example, assuming that `section_9` is defined as label, the expression `ref section_9` is a statement-expression.

4.1.2.4 Phrase level

The phrase level describes the atomic elements of the discourse. In CGa these elements could be either a statement-expression, a definition or a sub refinement.

phrase, p	$::=$	exp	Statement phrase
	$\mid$	$\mathit{cident}(\overrightarrow{\mathit{ident}}) := \mathit{exp}$	Definition
	$\mid$	$\mathit{cident}(\overrightarrow{\mathit{exp}}) := \mathit{exp}$	Case definition
	$\mid$	$\mathit{ident} \ll \mathit{exp}$	Sub refinement

4.1.2.4.1 Statement phrase A statement brought forward is a phrase in CGa. For example, the expressions such as `subset(A,B)` or `x:A` are phrases.

4.1.2.4.2 Definition A *definition* assigns to a particular expression a shorthand name. Such definition could be parameterised which therefore means that each instance of the identifier to be defined will need to come with arguments implementing the parameters. The parameterisation of a definition can take two forms in CGa. In the first form, each parameter is represented by a single identifier and these identifiers could occur in the definition's body expression. In the second form, one describes the identifier's behavior depending on the argument input. The full definition of the identifier therefore takes one or more phrases to be completed. Each phrase corresponds to one matching case. Each parameter is therefore represented as an expression taking the role of a pattern. See Section 4.3.1.2 for an additional discussion of definition by matching cases. Piecewise functions are among the mathematical objects whose definitions are simplified by the use of definitions by matching cases. Here is a CGa encoding of the definition of the identity function defined on $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$.

```
{
  x:R |> Id(x) := x;
  Id(neg(infinity)) := neg(infinity);
```

```
        Id(infinity) := infinity
    }
```

4.1.2.4.3 Sub refinement *Sub refinements* compose the third kind of CGa phrases. A sub refinement is similar to a refinement in the sense that it combines descriptors but a sub refinement also alters the definition of an identifier. A sub refinement refines, for the rest of the document, the definition of an already defined noun, adjective, term or set. The refinement is derived from either a noun-expression or an adjective-expression. The notion of sub refinement is inspired by the MV notion of sub-substantives [dB87, §1.14] (we used the same symbol `<<`). The expression `square << rectangular` used as an example in [dB87, §1.14] is also a CGa example assuming that both `square` and `rectangular` are predefined. This phrase-expression means that, from this phrase onward, `square` is given all the characters that `rectangular` holds, `rectangular` being unchanged. Another example that does not use a named noun is the phrase: `triangle << Noun { angles:set }` extends the definition of a triangle (which may for instance contain a set-character `sides`) with the set-character `angles`. Sub refinements permit to spread the definition of a noun inside the document according to needs.

4.1.2.5 Discourse level

The discourse level describes the manner in which argumentation elements can be combined. In CGa, a justification or argumentation or discourse is depicted as a *step*. There are three step constructions: the basic step, the local scoping and the block. Additionally, one can assign a label to step.

step, *s*	::=	*phrase*	Basic step
	|	*step* $\rhd$ *step*	Local scoping
	|	$\{\overrightarrow{step}\}$	Block
	|	`label` *label step*	Step label

4.1.2.5.1 Basic step A phrase is an atomic or basic step in CGa. For example, `Th:=irrational(sqrt(2))`, `subset(A,B)`, `x:A` and `square << rectangular` are basic steps.

4.1.2.5.2 Local scoping A local scoping puts a step as a context for the development of another step. This contextualisation has some incidence on the scoping of identifiers, see Section 4.3.1.4 for a detailed description of scoping. The notion of local scoping unifies the notions of context and flags from MV, WTT and MWTT and the notion of local definition from MWTT.

A local scoping could imply an atomic step be put as a context for a phrase, for example in `a:R |> =(+(a,0),a)`, the local scoping is used restrict the declaration of the variable `a`.

In other cases, a bigger step could be involved in a local scoping. For example, in a proof by induction, the induction hypothesis is to be in a local scoping.

```
{ [A proof of  P  by induction]
  { [Proof of the base]
    [...] P(0);
  };
  { [Proof of the induction]
    { n:N; P(n) } |> { [...] P(n+1) }
  }
}
```

An entire proof (e.g., a proof by contradiction) can be put in context in a local scoping.

```
{ [Proof of the contradiction] [...] }
  |> { [Statement proved by contradiction] [...] }
```

4.1.2.5.3 Block A block is a sequence of steps in CGa. Sequences of statements in a proof are represented by a block. Here is for example a block composed by basic steps.

```
{
  x.(y+1) = x.y';
  x.y' = x.y+x;
  x.y+x = x.y+x.1
}
```

4.1.2.5.4 Step label The last alternative is the *step label* which labels a step for later recall with a step reference.

4.2 CGa's Type System

In this section we define the type system of MathLang-CGa. Appendix A.2 contains a summary of the typing rules defined here. Most material in this section were published in [KMW06].

4.2.1 Types and notations

We assume the following conventions. We denote by $\wp(S)$ the power set of the set S. An ordered pair is denoted (a, b) and functions are taken to be sets φ of ordered pairs with a domain $\mathrm{dom}(\varphi) = \{a \mid \exists b, (a, b) \in \varphi\}$. By $\#S$ we denote the cardinality of the set S. And $\mathbb{N}$ denotes the set of natural numbers.

4.2.1.1 Types

CGa's type system attributes a *type* to each valid identifier and an *atomic type* to each valid construction of the language. We define the set $\mathcal{A}$ of atomic type signatures, the set $\mathcal{T}$ of types and the set $\mathcal{M}$ of *type mappings* as follows.

$$\begin{array}{rcl} \mathcal{A} & ::= & \mathit{Term}(\mathcal{M}) \mid \mathit{Set}(\mathcal{M}) \mid \mathit{Noun}(\mathcal{M}) \mid \mathit{Adj}(\mathcal{M}, \mathcal{M}) \\ & & \mid \mathit{Stat} \mid \mathit{cvar} \mid \mathit{Dec}(\mathcal{T}) \mid \mathit{Def}(\mathcal{T}) \mid \mathit{Sub}(\mathcal{T}) \mid \mathit{Step} \mid \mathit{Categ}(\mathcal{A}) \\ \mathcal{T} & ::= & (\overrightarrow{\mathcal{A}}) \rightarrow \mathcal{A} \\ \mathcal{M} & ::= & \overrightarrow{(\mathcal{I}, \mathcal{T})} \end{array}$$

The variables a, t and m range over $\mathcal{A}$, $\mathcal{T}$ and $\mathcal{M}$ respectively. A $\mathcal{T}$ element associates a sequence of atomic input types to one output type. We denote by $(a_1, \ldots, a_n) \rightarrow a$ a typical $\mathcal{T}$ element. And $\mathcal{M}$ is the set of mappings from $\mathcal{I}$ to $\mathcal{T}$. We denote by $m(i)$ the element t such that $(i, t) \in m$. We use *Categ* as $\mathcal{A}$ elements to disambiguate the types of category expressions from the types of expressions (see Section 4.2.3.1.2).

4.2.1.2 Typing judgments

In Section 4.2.3 we define the set of rules of our type system. Each typing rule has the following form.

$$\textit{typing context} \vdash \textit{construction} : \textit{type judgement}$$

A *typing context* is a well formed *step* of the language. It represents the preceding steps of reasoning in which the *construction* is typed. We have made an effort to keep this major feature of Automath and WTT of typing under a context formed by a concrete expression (in contrast with building a typing environment). *Construction* could be any abstract expression constructed by following CGa's abstract syntax of Section 4.1. A *type judgement* is either a *type signature* or an *atomic type*.

We occasionally use the symbol / to group similar rules

4.2.2 Functions, operations and relations

4.2.2.1 Functions

We list here the functions used in the derivation rules of our type system. These functions are I, dI, DI, L and enum. In the following, ce ranges over both category expressions and expressions. These functions are defined as follows.

Function $I : \mathit{step} \mapsto \wp(\mathcal{I})$ gives the set of identifiers declared, defined or sub refined within a step.

$$\begin{aligned}
I(s_1 \rhd s_2) &= I(s_2)\\
I(\{s_1;\ldots;s_n\}) &= \bigcup_{i\in\{1\ldots n\}} I(s_i)\\
I(\texttt{label}\ l\ s) &= I(s)\\
I(i(i_1,\ldots,i_n) := e) &= \{i\}\\
I(i(e_1,\ldots,e_n) := e) &= \{i\}\\
I(i(ce_1,\ldots,ce_n) : ce) &= \{i\}\\
I(i \ll e) &= \{i\}\\
I(p) &= \emptyset\text{, otherwise}
\end{aligned}$$

Function $dI : step \mapsto \wp(\mathcal{I})$ gives the set of identifiers declared within a step.

$$\begin{aligned}
dI(s_1 \rhd s_2) &= dI(s_2)\\
dI(\{s_1; \ldots; s_n\}) &= \bigcup_{i \in \{1 \ldots n\}} dI(s_i)\\
dI(\texttt{label}\ l\ s) &= dI(s)\\
dI(i(ce_1, \ldots, ce_n) : ce) &= \{i\}\\
dI(p) &= \emptyset, \text{ otherwise}
\end{aligned}$$

Function $DI : step \mapsto \wp(\mathcal{I})$ gives the set of identifiers defined within a step.

$$\begin{aligned}
DI(s_1 \rhd s_2) &= DI(s_2)\\
DI(\{s_1; \ldots; s_n\}) &= \bigcup_{i \in \{1 \ldots n\}} DI(s_i)\\
DI(\texttt{label}\ l\ s) &= DI(s)\\
DI(i(i_1, \ldots, i_n) := e) &= \{i\}\\
DI(i(e_1, \ldots, e_n) := e) &= \{i\}\\
DI(p) &= \emptyset, \text{ otherwise}
\end{aligned}$$

Function $L : step \mapsto \wp(\mathcal{L})$ gives the set of labels defined within a step.

$$\begin{aligned}
L(s_1 \rhd s_2) &= L(s_2)\\
L(\{s_1; \ldots; s_n\}) &= \bigcup_{i \in \{1 \ldots n\}} L(s_i)\\
L(\texttt{label}\ l\ s) &= \{l\}\\
L(p) &= \emptyset
\end{aligned}$$

And function enum : $\wp(\mathbb{N}) \mapsto \wp(\mathbb{N} \times \mathbb{N})$ gives an inorder-enumeration function for a set of natural numbers.

$$\text{enum}(S) = \{(a, b) \mid b \in S \text{ and } a = \#\{c | c \leq b\}\}$$

4.2.2.2 Operations

We define a binary operation on type mappings. In what follows, dom(f) stands for the domain of the function f.

$$m \uplus m' = m \cup \{\ (i, m'(i)) \mid i \in \mathrm{dom}(m') \text{ and } i \notin \mathrm{dom}(m)\}$$

4.2.2.3 Subtyping relations

We need first to define the substitution operation for category variables. A substitution σ is a set of mappings from $\mathcal{V}$ to $\mathcal{A}$. We denote by $\sigma(a)$ the application of a substitution σ to an atomic type a. The substitution application is defined as follows.

$$\begin{aligned}
\sigma(\mathit{Categ}(a)) &= \mathit{Categ}(\sigma(a))\\
\sigma(\mathit{Dec}(() \to a)) &= \mathit{Dec}(() \to \sigma(a))\\
\sigma(v) &= a \text{ if } (v, a) \in \sigma\\
\sigma(a) &= a \text{ otherwise}
\end{aligned}$$

We define the subtyping relations $\dot{\preccurlyeq}$, $\bar{\preccurlyeq}$ and $\preccurlyeq$ between atomic types, types and type mappings respectively.

$$\begin{array}{rcll}
\mathit{Term}(m) & \dot{\preccurlyeq} & \mathit{Term}(m') & \textit{if } m \ \preccurlyeq\ m'\\
\mathit{Set}(m) & \dot{\preccurlyeq} & \mathit{Set}(m') & \textit{if } m \ \preccurlyeq\ m'\\
\mathit{Noun}(m) & \dot{\preccurlyeq} & \mathit{Noun}(m') & \textit{if } m \ \preccurlyeq\ m'\\
\mathit{Adj}(m_1, m_2) & \dot{\preccurlyeq} & \mathit{Adj}(m_1', m_2') & \textit{if } m_2 \ \preccurlyeq\ m_2' \textit{ and } m_1' \ \preccurlyeq\ m_1\\
\mathit{Stat} & \dot{\preccurlyeq} & \mathit{Stat} & \\
\mathit{Dec}(t) & \dot{\preccurlyeq} & \mathit{Dec}(t') & \textit{if } t \ \bar{\preccurlyeq}\ t'\\
\mathit{Def}(t) & \dot{\preccurlyeq} & \mathit{Def}(t') & \textit{if } t \ \bar{\preccurlyeq}\ t'\\
\mathit{Sub}(t) & \dot{\preccurlyeq} & \mathit{Sub}(t') & \textit{if } t \ \bar{\preccurlyeq}\ t'\\
\mathit{Step} & \dot{\preccurlyeq} & \mathit{Step} & \\
v & \dot{\preccurlyeq} & v & \\
\mathit{Categ}(a) & \dot{\preccurlyeq} & \mathit{Categ}(a') & \textit{if } a \ \dot{\preccurlyeq}\ a'\\
t & \bar{\preccurlyeq} & t' & \textit{if } \ t = (a_1, \ldots, a_n) \to a,\ t' = (a_1', \ldots, a_n') \to a',\\
 & & & \quad \exists\ \sigma \textit{ such that } \sigma(a)\ \dot{\preccurlyeq}\ \sigma(a'),\\
 & & & \quad \textit{and } \forall j \in \{1 \ldots n\}, \sigma(a_j) \dot{\preccurlyeq} \sigma(a_j')\\
m & \preccurlyeq & m' & \textit{if } \forall i \in \mathrm{dom}(m), m(i)\ \bar{\preccurlyeq}\ m'(i)
\end{array}$$

4.2.3 Typing rules

4.2.3.1 Rules for the vocabulary level

This section contains the typing rules for the vocabulary level of Section 4.1.2.1.

4.2.3.1.1 Typing of identifiers We define here a set of rules to express the role of the context in the typing of identifiers.

The first three rules IDENT-DEC, IDENT-DEF and IDENT-SUB indicate that if one adjoins a phrase p to a well formed step $\{\overrightarrow{s}\}$ and if this phrase p is a valid (in the context formed by s) declaration, definition or sub refinement of an identifier i, then i gets a type corresponding to this declaration, definition or sub refinement in the context composed by the step $\{\overrightarrow{s}\}$ and the phrase p.

$$\frac{\{\overrightarrow{s}\} \vdash p \mathrel{\textbf{:}} \mathit{Dec}(t) \qquad \mathit{dI}(p) = \{i\}}{\{\overrightarrow{s}; p\} \vdash i \mathrel{\textbf{:}} t}\ \textsc{ident-dec}$$

$$\frac{\{\overrightarrow{s}\} \vdash p \mathrel{\textbf{:}} \mathit{Def}(t) \qquad \mathit{DI}(p) = \{i\}}{\{\overrightarrow{s}; p\} \vdash i \mathrel{\textbf{:}} t}\ \textsc{ident-def}$$

$$\frac{\{\overrightarrow{s}\} \vdash p \mathrel{\textbf{:}} \mathit{Sub}(t) \qquad I(p) = \{i\}}{\{\overrightarrow{s}; p\} \vdash i \mathrel{\textbf{:}} t}\ \textsc{ident-sub}$$

In the rule CHARACTER, the typing of a term-expression e with a specific character i indicates how to get the type of this character's instance.

$$\frac{s \vdash e \mathrel{\textbf{:}} \mathit{Term}(m) \qquad i \in \mathrm{dom}(m)}{s \vdash e.i \mathrel{\textbf{:}} m(i)}\ \textsc{character}$$

As for the rules IDENT-BASIC, IDENT-LOCAL-SCOPING, IDENT-BLOCK and IDENT-LABEL, they repercuss the typing of an identifier through a phrase, a block element, a local scoping and a step labelling respectively.

$$\frac{\{\overrightarrow{s}\} \vdash i : t \qquad i \notin I(p)}{\{\overrightarrow{s};\ p\} \vdash i : t}\ \textsc{ident-basic}$$

$$\frac{\{\overrightarrow{s}; s'; s''\} \vdash i : t \qquad i \in I(s'')}{\{\overrightarrow{s};\ s' \rhd s''\} \vdash i : t}\ \textsc{ident-local-scoping}$$

$$\frac{\{\overrightarrow{s_1};\ \overrightarrow{s_2}\} \vdash i : t}{\{\overrightarrow{s_1};\ \{\overrightarrow{s_2}\}\} \vdash i : t}\ \textsc{ident-block} \qquad \frac{\{\overrightarrow{s}; s'\} \vdash i : t}{\{\overrightarrow{s}; \texttt{label}\ l\ s'\} \vdash i : t}\ \textsc{ident-label}$$

With this set of rules for the vocabulary level, we retrieve the type of an identifier from the context. Combined, they decompose the step context to retrieve the declaration, definition or sub refinement informing on the type of an identifier i.

4.2.3.1.2 Typing of category expressions Category expressions are used in declarations. They permit to set the categories of an identifier' parameters and output. The constructors `term`, `noun`, `adj` and `set` are parameterised with a noun expression. This noun expression gives the set of characters of the described category.

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash e : \mathit{Noun}(m)}{s \vdash \texttt{term}(e)/\texttt{set}(e)/\texttt{noun}(e) : \mathit{Categ}(\mathit{Term}(m)/\mathit{Set}(m)/\mathit{Noun}(m))}\ \textsc{categ-term/set/noun}$$

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash e : \mathit{Noun}(m) \qquad s \vdash e' : \mathit{Noun}(m') \qquad m \preccurlyeq m'}{s \vdash \texttt{adj}(e, e') : \mathit{Categ}(\mathit{Adj}(m, m'))}\ \textsc{categ-adj}$$

$$\frac{\vdash s : \mathit{Step}}{s \vdash \texttt{stat} : \mathit{Categ}(\mathit{Stat})}\ \textsc{categ-stat}$$

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash c : \mathit{Categ}(a)}{s \vdash \texttt{dec}(c) : \mathit{Categ}(\mathit{Dec}(() \to a))}\ \textsc{categ-dec} \qquad \frac{\vdash s : \mathit{Step}}{s \vdash v : \mathit{Categ}(v)}\ \textsc{categ-var}$$

These rules make the bridge between category expressions and $\mathcal{A}$ elements. Note that all the rules CATEG-* attributes a *Categ* atomic type to every category expression. This is not to confuse with types of expressions.

4.2.3.2 Rules for the expression level

This section contains the typing rules for the expression level of Section 4.1.2.3.

4.2.3.2.1 Typing of instances The typing of instances of rule INSTANCE is mainly concerned with the possible instantiation of parameters. Each argument expression should satisfy the type of the corresponding parameter. Note that each argument is typed in a context which includes the potential declarations of what the preceding arguments might be. It is important also to notice the constraint $a' \notin \mathcal{V}$ which enforces any instance to have a concrete atomic type. See Section 4.3.2.2 for a discussion on this language feature.

$$\frac{\begin{array}{c} \vdash s \boldsymbol{:} \mathit{Step} \qquad s \vdash ci \boldsymbol{:} (a_1, \ldots, a_n) \to a \\ \forall j \in \{1 \ldots n\},\ f = \text{enum}(\{q \mid 1 < q < j \text{ and } dI(e_q) \neq \emptyset\}) \\ \text{and } \{s; e_{f(1)}; \ldots; e_{f(j-1)}\} \vdash e_j \boldsymbol{:} a'_j \\ a' \notin \mathcal{V} \qquad (a_1, \ldots, a_n) \to a \bar{\preccurlyeq} (a'_1, \ldots, a'_n) \to a' \end{array}}{s \vdash ci(e_1, \ldots, e_n) \boldsymbol{:} a'} \text{ INSTANCE}$$

4.2.3.2.2 Typing of declarations Declarations introduce new identifiers. The categories of the declared identifier and its parameters could be either explicitly expressed with a category expression or stated in the identifier's elementhood via a noun- or a set- expression. Both cases are handled by the rule DEC with the $ce = c$ and $ce = e$ conditions respectively.

$$\frac{\begin{array}{c} \vdash s \boldsymbol{:} \mathit{Step} \qquad i \notin I(s) \qquad \forall j \in \{1 \ldots n\},\ \text{if } ce_j = c_j \text{ then } s \vdash c_j \boldsymbol{:} \mathit{Categ}(a_j) \\ \forall j \in \{1 \ldots n\},\ \text{if } ce_j = e_j \text{ then } s \vdash e_j \boldsymbol{:} \mathit{Noun}(m_j)/\mathit{Set}(m_j) \text{ and } a_j = \mathit{Term}(m_j) \\ \text{if } ce = c \text{ then } s \vdash c \boldsymbol{:} \mathit{Categ}(a) \\ \text{if } ce = e \text{ then } s \vdash e \boldsymbol{:} \mathit{Noun}(m)/\mathit{Set}(m) \text{ and } a = \mathit{Term}(m) \end{array}}{s \vdash i(c_1, \ldots, c_n) : e \boldsymbol{:} \mathit{Dec}((a_1, \ldots, a_n) \to a)} \text{ DEC}$$

4.2.3.2.3 Typing of descriptions The `Noun` constructor takes a step as argument and the `Adj` takes an expression and a step as arguments. Their step argument describes the characteristics and behavior of the terms that could inhabit them. The adjective description extra expression indicates the domain of applicability of the adjective described. The NOUN rule specifies that each identifier declared or defined within the noun description becomes a character for this noun. Similarly, the ADJ rule specifies that each identifier declared, defined or refined within the adjective description becomes an output character for this adjective. In addition the ADJ rule indicates that the expression argument of `Adj` is the input noun of this adjective.

$$\frac{\begin{array}{c} \vdash s : \mathit{Step} \qquad \{s;\ \texttt{self} : \texttt{term}\} \vdash s' : \mathit{Step} \\ \forall i \in dI(s') \cup DI(s'),\ \{s;\ \texttt{self} : \texttt{term};\ s'\} \vdash i : m(i) \end{array}}{s \vdash \texttt{Noun}\ \{s'\}\ : \mathit{Noun}(m)}\ \text{NOUN}$$

$$\frac{\begin{array}{c} \vdash s : \mathit{Step} \qquad s \vdash e : \mathit{Noun}(m) \\ \{s;\ \texttt{self} : e\} \vdash s' : \mathit{Step} \qquad \forall i \in I(s'),\ \{s;\ \texttt{self} : e;\ s'\} \vdash i : m'(i) \end{array}}{s \vdash \texttt{Adj}\ (e)\ \{s'\}\ : \mathit{Adj}(m, m')}\ \text{ADJ}$$

In both cases the special keyword `self` could be used inside the descriptive step to stand for the generic term characterised. A self marker is introduce indicating the type of this keyword. See Section 4.2.3.2.5 for the rules typing `self`.

Remark 4 (Self marker) *It is important to notice that the self markers* `self` : `term` *and* `self` : e *can not be constructed according to the abstract syntax of Section 4.1,* `self` *being a keyword and not an identifier of* $\mathcal{I}$. *Therefore the rules of Section 4.2.3.1.1 are not applicable to retrieve the type of* `self` *inside the context.*

4.2.3.2.4 Typing of refinements A refinement is the application of an adjective to either a term, a set, a noun or another adjective. The rules TERM-REFINEMENT, SET-REFINEMENT and NOUN-REFINEMENT restrict such refinement to a term, a set or a noun with sufficient characters accordingly to the adjectives input (condition $m_1 \preccurlyeq m_2$). The condition $\forall i \in (\mathrm{dom}(m'_1) \setminus \mathrm{dom}(m_1)) \cap \mathrm{dom}(m_2),\ m_2(i) \bar{\preccurlyeq} m'_1(i)$ makes sure that no character clash occurs if some adjective characters were already present. Note the use of $m'_1 \uplus m_2$ (instead of $m_2 \uplus m'_1$) to give priority to the adjective characters.

The rule ADJ-REFINEMENT enforces the fact that the output of the adjective to be refined is acceptable by the refiner-adjective input (condition $m_1 \preccurlyeq m'_2$).

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad s \vdash e_1 : \mathit{Adj}(m_1, m'_1) \\ s \vdash e_2 : \mathit{Noun}(m_2)/\mathit{Set}(m_2)/\mathit{Term}(m_2) \qquad m_1 \preccurlyeq m_2 \\ \forall i \in (\mathrm{dom}(m'_1) \setminus \mathrm{dom}(m_1)) \cap \mathrm{dom}(m_2),\ m_2(i) \mathrel{\bar{\preccurlyeq}} m'_1(i)\end{array}}{s \vdash e_1\ e_2 : \mathit{Noun}(m'_1 \uplus m_2)/\mathit{Set}(m'_1 \uplus m_2)/\mathit{Term}(m'_1 \uplus m_2)}\ \begin{array}{l}\textsc{term/}\\ \textsc{set/}\\ \textsc{noun}\end{array}\ \textsc{-refinement}$$

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \\ s \vdash e_1 : \mathit{Adj}(m_1, m'_1) \qquad s \vdash e_2 : \mathit{Adj}(m_2, m'_2) \qquad m_1 \preccurlyeq m'_2 \\ \forall i \in (\mathrm{dom}(m'_1) \setminus \mathrm{dom}(m_1)) \cap (\mathrm{dom}(m'_2) \setminus \mathrm{dom}(m_2)),\ m'_2(i) \mathrel{\bar{\preccurlyeq}} m'_1(i)\end{array}}{s \vdash e_1\ e_2 : \mathit{Adj}(m_2, m'_2 \uplus m'_1)}\ \textsc{adj-refinement}$$

4.2.3.2.5 Typing of self The rules SELF-NOUN and SELF-ADJ indicate that the special identifier `self` is typed according to a self marker in the typing context. This marker is disposed by the noun and adjective descriptor typing rules NOUN and ADJ.

$$\frac{\vdash \{\overrightarrow{s}\} : \mathit{Step}}{\{\overrightarrow{s};\ \texttt{self} : \texttt{term}\} \vdash \texttt{self} : \mathit{Term}(\emptyset)}\ \textsc{self-noun}$$

$$\frac{\vdash \{\overrightarrow{s}\} : \mathit{Step} \qquad \{\overrightarrow{s}\} \vdash e : \mathit{Noun}(m)}{\{\overrightarrow{s};\ \texttt{self} : e\} \vdash \texttt{self} : \mathit{Term}(m)}\ \textsc{self-adj}$$

The rule SELF-CHARACTER indicates that all, directly available identifier in the context is a character for self.

$$\frac{\begin{array}{c}\vdash \{\overrightarrow{s}\} : \mathit{Step} \\ \{\overrightarrow{s}\} \vdash \texttt{self} : \mathit{Term}(m) \qquad i \in I(p) \qquad \{\overrightarrow{s};\ p\} \vdash i : t\end{array}}{\{\overrightarrow{s};\ p\} \vdash \texttt{self} : \mathit{Term}((i, t) \uplus m)}\ \textsc{self-character}$$

Similarly to rules IDENT-BASIC, IDENT-LOCAL-SCOPING, IDENT-BLOCK and IDENT-LABEL, the rules SELF-BASIC, SELF-LOCAL-SCOPING, SELF-BLOCK and SELF-LABEL repercuss the typing of `self` through a block element, a local scoping and a step labelling respectively.

$$\frac{\{\overrightarrow{s}\} \vdash \texttt{self} : a \qquad I(s') = \emptyset}{\{\overrightarrow{s}; p\} \vdash \texttt{self} : a} \ \textsc{self-basic}$$

$$\frac{\{\overrightarrow{s}; s_2\} \vdash \texttt{self} : a}{\{\overrightarrow{s}; s_1 \rhd s_2\} \vdash \texttt{self} : a} \ \textsc{self-local-scoping}$$

$$\frac{\{\overrightarrow{s_1}; \overrightarrow{s_2}\} \vdash \texttt{self} : a}{\{\overrightarrow{s_1}; \{\overrightarrow{s_2}\}\} \vdash \texttt{self} : a} \ \textsc{self-block} \qquad \frac{\{\overrightarrow{s}; s'\} \vdash \texttt{self} : a}{\{\overrightarrow{s}; \texttt{label}\ l\ s'\} \vdash \texttt{self} : a} \ \textsc{self-label}$$

4.2.3.2.6 Typing of references The rule REF indicates that if a label exists in a step, it could be referred to in the context of this step. A reference has type *Stat*.

$$\frac{\vdash s : \mathit{Step} \qquad l \in L(s)}{s \vdash \texttt{ref}\ l : \mathit{Stat}} \ \textsc{ref}$$

4.2.3.3 Rules for the phrase level

This section contains the typing rules for the phrase level of Section 4.1.2.4.

4.2.3.3.1 Typing of definitions Definitions introduce new identifiers. For a definition, the parameters could either be identifiers (DEF rule) or expressions for definition by matching cases (DEF-CASE rule). Note that definitions have one restriction in comparison with declarations. Definitions of polymorphic identifiers are not allowed. This is due to the fact that any CGa expression ought to have a concrete category. One could declare `identity (a):a` but can not define it with the same type (see Section 4.3.2.2).

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad i \notin \mathit{DI}(s) \\ \forall j,k \in \{1 \ldots n\},\ j \neq k \Rightarrow i_j \neq i_k \qquad \forall j \in \{1 \ldots n\},\ s \vdash i_j : () \to a_j \\ s \vdash e : a \qquad \text{if}\ \ i \in \mathit{dI}(s)\ \ \text{then}\ \ s \vdash i : (a_1, \ldots, a_n) \to a\end{array}}{s \vdash i(i_1, \ldots, i_n) := e : \mathit{Def}((a_1, \ldots, a_n) \to a)}\ \textsc{def}$$

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad \text{if}\ \ i \in I(s)\ \ \text{then}\ \ s \vdash i : (a_1, \ldots, a_n) \to a \\ \forall j \in \{1 \ldots n\},\ s \vdash e_j : a_j \qquad s \vdash e : a\end{array}}{s \vdash i(e_1, \ldots, e_n) := e : \mathit{Def}((a_1, \ldots, a_n) \to a)}\ \textsc{def-case}$$

4.2.3.3.2 Typing of sub refinements The rule SUB-NOUN lists the conditions to build a correct sub refinement with a term- set- or noun-identifier, and a noun expression. The rule SUB-ADJ does so with an identifier and an adjective expression.

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad s \vdash i : \mathit{Term}(m_1)/\mathit{Set}(m_1)/\mathit{Noun}(m_1) \\ s \vdash e : \mathit{Noun}(m_2) \qquad \forall i' \in \mathrm{dom}(m_1) \cap \mathrm{dom}(m_2),\ m_1(i') \bar{\preccurlyeq} m_2(i')\end{array}}{s \vdash i \ll e : \mathit{Sub}(\mathit{Term}(m_1 \uplus m_2)/\mathit{Noun}(m_1 \uplus m_2)/\mathit{Set}(m_1 \uplus m_2))}\ \textsc{sub-noun}$$

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad s \vdash i : \mathit{Term}(m_1) \\ s \vdash e : \mathit{Adj}(m_2, m_2') \qquad \forall i' \in \mathrm{dom}(m_1) \cap \mathrm{dom}(m_2'),\ m_1(i') \bar{\preccurlyeq} m_2'(i')\end{array}}{s \vdash i \ll e : \mathit{Sub}(\mathit{Term}(m_1 \uplus m_2')/\mathit{Noun}(m_1 \uplus m_2')/\mathit{Set}(m_1 \uplus m_2'))}\ \textsc{sub-adj}$$

4.2.3.4 Rules for the discourse level

This section contains the typing rules for the discourse level of Section 4.1.2.5.

4.2.3.4.1 Typing of basic steps Only well typed statements, declarations, definitions or sub refinements could be considered as phrases or basic steps according to the BASIC-STEP rule.

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash p : \mathit{Stat}/\mathit{Dec}(t)/\mathit{Def}(t)/\mathit{Sub}(t)}{s \vdash p : \mathit{Step}}\ \textsc{basic-step}$$

4.2.3.4.2 Typing of local scopings The rule LOCAL-SCOPING for local scopings indicates that once a step s_2 is valid in the context of a step s_3, it is possible to form a local scoping which puts s_2 as a context for s_3.

$$\frac{\vdash s_1 \mathbin{\mathbf{:}} \mathit{Step} \qquad s_1 \vdash s_2 \mathbin{\mathbf{:}} \mathit{Step} \qquad \{s_1; s_2\} \vdash s_3 \mathbin{\mathbf{:}} \mathit{Step}}{s_1 \vdash s_2 \rhd s_3 \mathbin{\mathbf{:}} \mathit{Step}}\ \textsc{local-scoping}$$

4.2.3.4.3 Typing of blocks The rule BLOCK expresses that if a block of steps $\{\overrightarrow{s}\}$ is valid in a certain context s_1 and if a step s_2 is valid under the context extended with this block ($\{s_1; \{\overrightarrow{s}\}\}$), then extending the block $\{\overrightarrow{s}\}$ with the step s_2 forms a valid block $\{\overrightarrow{s}; s_2\}$. With the rule EMPTY-STEP we state that the empty step is a valid step. The extra rule SELF-MARKER states that the self marker, used in the rules for typing `self` and in the rules NOUN and ADJ, is considered as a normal step by the type system.

$$\frac{\vdash s_1 \mathbin{\mathbf{:}} \mathit{Step} \qquad s_1 \vdash \{\overrightarrow{s}\} \mathbin{\mathbf{:}} \mathit{Step} \qquad \{s_1; \{\overrightarrow{s}\}\} \vdash s_2 \mathbin{\mathbf{:}} \mathit{Step}}{s_1 \vdash \{\overrightarrow{s}; s_2\} \mathbin{\mathbf{:}} \mathit{Step}}\ \textsc{block}$$

$$\frac{}{\vdash \{\} \mathbin{\mathbf{:}} \mathit{Step}}\ \textsc{empty-step} \qquad\qquad \frac{\vdash s \mathbin{\mathbf{:}} \mathit{Step}}{s \vdash \texttt{self} : \texttt{term}/e \mathbin{\mathbf{:}} \mathit{Step}}\ \textsc{self-marker}$$

4.2.3.4.4 Typing of step labels The rule LABEL indicates that if a step is a valid step, then labelling it with an unused label forms a valid step too.

$$\frac{\vdash s_1 \mathbin{\mathbf{:}} \mathit{Step} \qquad l \notin L(s_1) \qquad s_1 \vdash s_2 \mathbin{\mathbf{:}} \mathit{Step}}{s_1 \vdash \texttt{label}\ l\ s_2 \mathbin{\mathbf{:}} \mathit{Step}}\ \textsc{label}$$

4.3 Features of CGa

In this section we discuss in detail some features of MathLang-CGa. We group these features in three groups: the language features (Section 4.3.1), the typing features (Section 4.3.2) and the language's meta-theory (Section 4.3.3).

4.3.1 Language features

4.3.1.1 Binders

Binders play an important role in both MV, WTT and MWTT. Their presence in these languages reflect the wide use of variable binding operators in mathematics and logics. A binding makes an abstraction explicit by introducing a variable used throughout an expression. The lambda binder λ, the universal and existential quantifiers $\forall$ and $\exists$, or the integration sign $\int dx$ are among famous binders. We recommend the reading of [Zin04, § 4.2.2] where C. Zinn discusses the role of variables.

From a grammatical point of view a binder is a constructor which holds a variable declaration and provides the variable declared for an expression. The bindings are effectively defined in WTT and MWTT as the combination of a binder identifier, a declaration and an expression. In WTT the set of binders was assumed to be predefined. In our first year PhD report [Maa03], we considered splitting the set of binders into three families of binders.

1. The "pure abstraction" binders for which the abstraction permits to reason on an abstract object. For these bindings a substitution may occur as for A. Church's λ. We classified also the quantifiers $\forall$, $\exists$ and the operators such as Σ, `min`, *lim* and $\int dx$ among this family of binders.

2. The "description binders" for which the variable introduced stands for the binding expression itself. The body of the binding is therefore a description making some judgment on the variable introduced. B. Russell's ι binder is of this kind.

3. The "comprehension binders" for which the variable introduction is used to describe a complex structure. MV's *Set* binder and WTT's `Abst`, `Set`, `Noun` and `Adj` binders [KN04] are among this family of binders.

The goal of this decomposition was to group binders by their grammatical behavior. For family 1 the variable could be of any grammatical category (term and set more likely but one could imagine a λ-abstraction on a statement). For family 2 the variable bound and the binding itself share the same category as they stand for the same thing. The bounded expression is usually a statement. For family 3, the bounded expression is also usually a statement but the variable is just a medium for describing a notion that can not be described by extension and therefore needs to be described by comprehension. For example, the *golden number* ϕ which could be described as the positive real number such that $\phi^2 - \phi - 1 = 0$.

The problem faced in WTT and MWTT was the strict definition and usage opportunities for binders (all binders have only one declaration and one expression as arguments). These languages did not permit to define new binders nor to get a

Declarations	
Common name	CGa declaration
Set binder	`Set( dec(term), stat ) : set`
$\forall$ quantifier	`forall( dec(‘a), stat ) : stat`
Church's λ	`lambda( dec(‘a), ‘b ) : ‘b`
Russell's definite description ι	`iota( dec(‘a), stat ) : ‘a`

Instances	
Formula	CGa equivalent
$\{x \in \mathbb{R} \mid x > 0\}$	`Set( x:R, >(x,0) )`
$\forall P, P(S) \Rightarrow P(\mathbb{R})$	`forall( P(set):stat , =>(P(S),P(N)) )`
$\lambda x.xx$	`lambda( x:term, app(x,x) )`
$n \in \mathbb{N}$ st. $3 < n < 5$	`iota( n:N, and(<(3,n),<(n,5)) )`

Table 4.1: Examples of binder identifiers

refine checking of their grammatical aspect (such as checking that both the variable and the binding expression share the same category in a ι-expression).

In CGa, we offer the possibility to declare new binders and we therefore give more expressive freedom. See Table 4.1 for some examples of declaration and instances of common binders. This is made possible by the following elements of the language.

- Declarations are first class element of the expression level, see Section 4.1.2.3. In WTT and MWTT they were a subsidiary level used in bindings and contexts.

- Earlier parameters holding declarations extend the typing of a parameter of an identifier's instantiation, see the rule INSTANCE on page 87.

Example 9 (Binders) *Let us see some examples of binder declarations and uses.*

λ-binder *A possible declaration of λ is* `lambda (dec(term),term):term` *if we decide to consider λ-expressions as CGa terms (see Table 4.1 for an alternative declaration). The λ-calculus K operator could therefore correspond to the expression* `lambda(x:term,lambda(y:term,y))`*.*

Quantifiers *One could define the $\forall$ quantifier and state that for an element e of a set S, e is greater or equal to any element of S. This could be done as follows (the prefix ‘ denotes category variables).*

```
{
  forall(dec(‘a),stat):stat;
```

```
    S:set;
    e:S;
    forall(x:S, >=(e,x));
  }
```

Limit operator *We define the limit operator to be the binder:* `lim(dec(term),term,term)`*. The second argument being the value the variable declared by the first argument approaches. The formula* $\lim_{x \to a} f(x)$ *would be expressed in CGa as* `lim(x:term,a,f(x))`*.*

ι-binder *We define here, using the ι-binder, the golden mean aforementioned. Note the repetition of the* `‘a` *category variable in the declaration of ι. We assume here* `R`*,* `=`*,* `-`*,* `sq`*,* `1` *and* `0` *to be predefined.*

```
  {
    iota(dec(‘a),stat):‘a;
    phy := iota( x:R, =(-(-(sq(x),x),1),0) );
  }
```

4.3.1.2 Definition by matching cases

In CGa, definitions define the meaning of a symbol. The language gives two ways to define an identifier.

- Stand-alone definition. For this kind of definition a unique expression is given as definiens. The definiendum is a new denomination for this expression. A list variables plays the role of parameters of the definition. These variables should all be well declared identifiers and be unparameterised (as stated in the rule def on page 91). The types of these parameters and the type of the definiens-expression constitute the type signature of the identifier (as stated by the rules def on page 91 and ident-def on page 85). An instance of the newly defined identifier needs to provide an expression-argument per parameter matching the parameter's type (see the rule instance on page 87). One can envision in later computerisations the in-lining of the identifier's definiens in place of an identifier's instance. It is important to notice that, in contrast to WTT's typing rule, we do not restrict the set of free variables of the context to be equal to the set of parameters. Therefore, some free variables might occur in the definiens without being a parameter. We discuss this issue in Section 4.3.1.5.

Example 10 *The union of sets, $A \cup B = \{x \mid x \in A \vee x \in B\}$, could be defined with two parameters A and B:*

```
{
  in(term,set):stat;   Set(dec(term),stat):set;
  or(stat,stat):stat;   A:set;   B:set;

  union(A,B) := Set( x:term, or(in(x,A),in(x,B)) )
}
```

- Definition by cases. For this kind of definition, the language associates a symbol with one expression per situation. Each particular situation is given by a context and a pattern formed by the definiendum's parameters. The symbol's definition is therefore split into several definition cases.

  This kind of definition is similar to the definitions by pattern matching in ML languages [Pie02, Ch. 4] but with important differences. CGa allows the cases of definition to be disjoint in the document where traditional ML languages imposes the cases to be all defined at once. In CGa, a case of a definition could be redundant or "visibly" in contradiction with a previous one. Matching such as `function x -> true | x -> false` are allowed but usually result in a warning at ML compilation time. Our type system does not verify the exhaustibility of a pattern matching definition. This is mainly due to the fact that CGa types are not constructed.

  The only verification done by our type system (DEF-CASE on page 91) is the coherence of the typing result (i.e. the type signature of the identifier should be identical for each case). Note that the patterns are expressions of the language and that the typing of each pattern is done independently.

Example 11 *In [Lan51] (see Section 5.3.1), E. Landau defines addition as follows:*

$$x + 1 = x' \text{ for every } x$$
$$x + y' = (x + y)' \text{ for every } x \text{ and every } y$$

where x' stands for the successor of x. Here is a CGa encoding of this definition.

```
{
  N:set;   1:N;   S(N):N

  x:N |> {
                  +(x,1) := S(x);
           y:N |> +(x,S(y)) := S(+(x,y))
         }
}
```

Example 12 *The Fibonacci function that defines how to calculate Fibonacci numbers by recursion. The first and second numbers are* 0 *and* 1*. Each new number is the sum of the two previous ones.*

```
{
  N:set;   0:N;   1:N;   +(N,N):N;

  Fibonacci(0) := 0;
  Fibonacci(1) := 1;
  { n:N; >(n,0) } |> Fibonacci(+(n,2)) :=
                       +(Fibonacci(n),
                         Fibonacci(+(n,1)))
}
```

Remark 5 *Example 12 is a recursive definition. Such kind of definition could be expressed with a stand-alone definition or a definition by cases if the type signature of the function is already stated (either with a declaration or a previous case).*

4.3.1.3 One class of identifiers

In WTT and MWTT, identifiers are separated into three disjoint sets: *variables*, *constants*, and *binders*.

The rest of this paragraph briefly describes how identifiers work in WTT and MWTT (see Sections 2.2 and 4.4). All three kinds of identifiers have a weak type, and this is all that variables have. Constants also have a definition and parameters (each parameter being a variable declaration). Each use of a constant is applied to arguments of the right weak type. Binders have parameters like constants, and one additional special parameter for the bound variable. Unlike variables and constants, binders can not be defined inside a document but can only be listed in the *preface*. Binders can not be given definitions; a statement using a binder can act as a definition but there is no way to indicate this.

In encoding texts, we found these restrictions of the different identifier kinds problematic, so CGa instead now has just one kind of identifiers and distinguishes the uses via types. To fit binders in our new scheme and to allow declaring/defining new binders in documents, we replace the old single special parameter of each binder with a new kind of parameter with a *declaration type* usable with any identifier. For example, the binder $\forall$ might be declared as `forall(dec('a), stat):stat`, making it an identifier with output type `stat` and two parameters: a declaration of an identifier of arbitrary type `'a` and an expression of type `stat` (statement). An example using this identifier is the translation `forall( n:N, >=(n,0) )` of the proposition $\forall n \in \mathbb{N}.\, n \geq 0$ (assuming `N`, `>=` and `0` are already declared). See Section 4.3.1.1 and Table 4.1 for further examples.

4.3.1.4 Grouping and scopes

4.3.1.4.1 Grouping A fundamental idea of CGa (inherited from MV) is capturing the grammatical and binding structure of a mathematical text. In MV and WTT, each *line* of a *book* has a context representing the set of assumptions about types of variables ("let x be a natural number") and truths ("suppose $x = y^2$ for some natural number y") used in the definition or statement made by the line. MV allows using flags as a secondary graphical 2-dimensional way of writing the current context in a book; an element repeated in the contexts of consecutive lines can be written as a flag whose *head* contains the repeated element and whose *flagstaff* goes through all the lines repeating the element (see Section 2.1.2.2). MV also has a secondary notion of blocks derived from flag nesting.

Unlike MV, MWTT directly supports flags and blocks rather than treating them as secondary syntax-sugaring notions derived from the contexts, see Section 4.4. Upon careful examination of MWTT's flags and blocks, we found that they overlapped in function. MWTT's blocks allow grouping lines and sub-blocks and limiting to a block the scope of some of the constants defined in the block. MWTT's flag allow identifying a group of lines in which a context element is active.

In CGa, we instead merged similar functionality. A *block*, written $\{step_1, \ldots, step_n\}$, is a sequence of statements. The *local scoping* construct $step_1 \rhd step_2$ makes the declarations, definitions, sub refinements and assertions inside $step_1$

assumptions used by *step*$_2$ and restricts declarations, definitions and sub refinements inside *step*$_1$ to be visible only in *step*$_2$. Both blocks and local scoping constructs are *steps*, as are declarations, definitions, sub refinements and assertions. Steps can be of various sizes, such as the declaration of a variable, the definition of a function, a proof, or an entire book.

4.3.1.4.2 Scopes The scopes of identifiers depend on the location of their declarations or definitions. Declarations could occur anywhere in an expression or could be an atomic step. We explain here the three possible cases: a declaration/definition in a local scoping, a declaration/definition as atomic step in the body of a local scoping and a declaration as a parameter of an identifier. The first two are shared by definitions and declarations. The third one is declaration specific.

Local scoping The first case is the presence of a declaration or a definition in a local scoping. The introduced identifier is available in the step (and its substeps) covered by the local scoping. Here, an identifier `x` is declared in the context part of a local scoping. `x` is available in the part of this context that follows the declaration $\boxed{3}$, and also in the body part $\boxed{4}$ of the local scoping. But `x` is not available prior to being declared: in the preceding steps $\boxed{1}$ and in the preceding part of the context $\boxed{2}$ of the local scoping. Furthermore, `x` is not available in the steps that follow the local scoping $\boxed{5}$.

```
{
  1;
  { 2; x:term; 3; } |> { 4; };
  5;
};
```

Block The second case is a declaration or a definition inside some blocks. The identifier is therefore available is all the steps that follow. A declaration of a triangle is an atomic step of the sub-block of a block. The identifier `triangle` is not available before being declared $\boxed{1}$ and $\boxed{2}$ but is available in all that follows $\boxed{3}$ and $\boxed{4}$. The availability of `triangle` would have been identical if the declaration had been replaced by a definition.

```
{
  1;
  { 2;
```

```
          triangle:noun;
          3; };
        4;
      };
```

Binding The last case is a declaration as argument for an instantiation. If an identifier takes a declaration as parameter, then the declared identifier is available for the following parameters. We declare a binder identifier `b` with a declaration as a second parameter. We also declare an identifier `a` with three parameters. In an expression using these two identifiers, a variable `x` is declared. This identifier `x` is not available before being declared $\boxed{1}$ and $\boxed{2}$. `x` is available in the parameters of the `binder` that follows the declaration of `x` $\boxed{3}$. Finally `x` is neither available in the remaining part of the expression $\boxed{4}$ nor in the steps that follow $\boxed{5}$.

```
      {
        b(term, dec(term), term): term;
        a(term, term, term): stat;
        a( 1 , b(2,x:term,3), 4 );
        5;
      };
```

4.3.1.5 Context and parameters

A major difference between the CML form of a mathematical text and its formalised counterpart is the handling of parameters. When defining a new symbol (definiendum) the mathematician gives firstly a clear list of parameters that one should provide to make use of the newly defined symbol and secondly the actual expression (definiens) for which the new symbol will stand. This expression may contain occurrences of the parameters' variables. The generic representation of a definition is $f(x, y) := E[x, y]$ where f is the definiendum, x and y the parameters, $:=$ the definition symbol, E the definiens, and where the square brackets [and] contain the list of free variables of E. In a mathematical text, the parameters are either introduced implicitly of by a phrase like "let x and y be" but some situations could be more ambiguous when, for example, the list of parameters of a function f are clearly stated in a formula $f(x, y) = \ldots$ but where an extra parameter exists (such as the arbitrary set S of x and y). From a formal point of view, x and y are placeholders in the definiens as well as S. Formally S should be a parameter. We name this kind of parameters *ghost parameters*. To avoid such a situation which breaks the δ-reducibility, MV, WTT and MWTT constrain each variable of the context to be a parameter in a definition (see MWTT-FULL-DEF in MWTT's weak type system Section 4.4.2). This is directly inherited from Automath and assures the existence of a δ-normal-form for every expression. We decided to remove this constraint of having all the context's variables as parameters; this allows parameters to exist on their own and makes the author free to chose his parameters.

4.3.2 Typing features

4.3.2.1 Object orientedness, abstraction with nouns and adjectives

In this section we explain the reasons for CGa's object-orientedness.

4.3.2.1.1 Nouns as classes

> Let us try to compare MV and Automath. In the first place it must be said that MV has been inspired by the structure of Automath as well as by the tradition of writing in Automath. In that tradition elementhood, i.e. the fact that an object belongs to a set, is expressed by the typing mechanism available in Automath. So in order to say that p is an element of the set S, this is coded as $p : S$, so S is the type of p. This is in accordance with the tradition in the standard mathematical language. If we say that p is a demisemitriangle, one does not think of the set or the class of all demisemitriangles in the first place, but rather thinks of "demisemitriangle" as a type of p. It says what kind of things p is.
>
> In order to keep this situation alive, MV does not take sets as the primitive vehicles for describing elementhood but substantives (in the above example semidemitriangle is a substantive). *[dB87, §1.12]*

Nouns are abstractions that classify objects according to their common features as explained in Section 4.1. Nouns have an important place in the representations of mathematics in MV and WTT, see Sections 2.1 and 2.2 respectively.

As already mentioned in Section 3.4, we encountered limitations of the expressiveness of WTT-style nouns when we started translating Euclid's *Elements* [Hea56]. Euclid starts his first chapter by defining basic geometric objects such as points, lines, figures, triangles, angles, etc. The definition of a line is as follows:

Euclid's *Elements*	CGa translation
A point is that which has no parts	`point := Noun`
A line is breadthless length	`line := Noun {length:term}`
A surface is that which has length and breadth only	`surface := Noun {length:term; breadth:term}`

Table 4.2: Examples of noun definitions

> **20.** Of trilateral figures, an equilateral triangle is that which has its three sides equal, an isosceles triangle that which has two of its sides alone equal, and a scalene triangle that which has its three sides unequal.
>
> *Euclid [Hea56, Book I]*

Figure 4.1: Definition of trilateral figures by Euclid

A line is breadthless length. In MWTT, one way to write this is by defining *line* by forming a noun characterized by two statements: one that *line* "has length", the other that *line* is breadthless (does not "have breadth"). This uses a constant "has" which takes two nouns and returns a statement. This constant was unsatisfactory because it is hard to define its semantics precisely and because MWTT could not make any use of it for checking well-formedness. Because "has" deeply characterises the noun *line* and by consequence any concrete *line* — weak typed as "`term`" — created as a *line* instance, we felt it should be replaced by something that informs the language that *lines* have *length*, to allow approving of statements about the length of a line and disapproving of those about nonsense properties like its breadth, angle, weight, etc.

We found a solution in the concept of classes and objects in programming. A *line* is a class with one character *length*. Any instance of *line* is an object with a *length*. We characterise a line as breadthless in our translation with the absence of such a character. Table 4.2 gives more examples.

Consider the first definition in Figure 4.1 which contains the definition of trilateral figures by Euclid. This definition uses the noun *figure*. In the preceding definitions in [Hea56], *figures* (*rectilinear figures*) are defined as *those contained by straight lines*. Therefore we define the noun `figure` with one character being the set of *straight lines* (we shorten it to *lines* in this example) and a statement precising that the figure is contained by this set of lines. The `Noun` descriptor describes the noun with a step. The first unit of this step defines the character `sides`.

Definition 1. A set with an associative law of composition, possessing an identity element and under which every elements is invertible, is called a group. [...] A group G is called finite if the underlying set of G is finite [...]
A group [with operators] G is called *commutative* (or *Abelian*) if its group law is commutative. *N. Bourbaki [Bou74, Chapter I, §4]*

Figure 4.2: Definition of group, finite group and Abelian group by N. Bourbaki

Sides is a set of lines. The second unit of this step is a statement which uses an identifier `contained_by`. This identifier (assumed to be declared earlier) takes a term and a set and returns a statement (`contained_by (term,set): stat`). The two parameters passed to this identifier are the future instance of the figure itself (encoded by the keyword `self`) and by the sides of the figure (character `sides` of `self`).

```
    figure := Noun { sides:set(line);
                     contained_by(self,sides) }
```

Remark 6 *According to the typing rules of Section 4.2, the noun* `triangle` *and the adjective* `isosceles` *have the following types respectively.*

$$\mathit{Noun}(\ \{(\texttt{sides}, \mathit{Set}(\{(\texttt{length}, \mathit{Term})\})\}\ )$$

$$\mathit{Adj}(\{(\texttt{sides}, \mathit{Set}(\{(\texttt{length}, \mathit{Term})\})\},\ \{(\texttt{sides}, \mathit{Set}(\{(\texttt{length}, \mathit{Term})\})\})$$

Considering now the example of Figure 4.2 which is the definition of a *group* by N. Bourbaki. We define `group` as a noun, The characters of this noun are identifiable in the text. The set E, the compositional law $*$ and the neutral element e. Two statements also define a group: the associativity of $*$ and the existence of an inverse of any element of E (we use an infix notation for `=`).

```
    group := Noun { E:set;
                    { a:E; b:E } |> *(a,b):E;
                    e:E;
                    forall (a:E,
                      forall (b:E,
                        forall (c:E,
                          *(*(a,b),c) = *(a,*(b,c)))));
                    forall (x:E, invertible(e,x))
                  }
```

Remark 7 *According to the typing rules of Section 4.2, the* `group` *identifier has type* $Noun(\ \{(\mathtt{E}, Set),\ (*, (Term, Term) \rightarrow Term),\ (\mathtt{e}, Term)\}\ )$.

Remark 8 *The type system prevents any misuse of identifiers' characters. For instance, let* `ABC` *be a declared triangle (*`ABC:triangle`*). This triangle is therefore a term with type* $Term(\ \{(\mathtt{sides}, Set(\{(\mathtt{length}, Term)\})\}\ )$*. According to our definition of triangle, the only defined character is* `sides`*, the set of lines composing a triangle. The expression* `ABC.sides` *refers to the sides of our triangle* `ABC`*. The set* `ABC.sides` *has type* $Set(\{(\mathtt{length}, Term)\})$*.*

4.3.2.1.2 Adjectives as mixins

> An adjective belongs to a substantive, and serves a double purpose: (i) to form a new substantive, and (ii) to form a new sentence.
>
> *[dB87, §1.19]*

According to (i), an adjective is a function from noun to noun. An adjective, like *isosceles*, when applied to a noun like *triangle* creates a new noun *isosceles triangle*. In our system where nouns are classes, the adjectives are therefore mixins [FKF98]. Intuitively, a mixin is a function from class to class. As in mixin calculi, an adjective can also be applied to an adjective to form a new adjective, to a term to form a new term, and to a set to form a new set (mapping the adjective across all members of the set). In CGa, we call these constructions *refinements*. The rules NOUN-REFINEMENT, ADJ-REFINEMENT, TERM-REFINEMENT and ADJ-REFINEMENT dictate the typing of such refinements accordingly.

Following (ii), we also incorporate the possibility that an existing term, noun or set gets the properties held by an adjective or a noun. For example one can describe a triangle ABC and demonstrate that this triangle is isosceles. The last line of this demonstration can be written in CGa as the statement: `ABC << isosceles` (read ABC is isosceles). In our syntax we extend this *sub refinement* to be a cast inheritance. The expression $A \ll B$, given by N.G. de Bruijn in MV, is seen as a statement which states that *"every A is a B"*. For example, `triangle << trilateral figure`. We kept this notation in CGa.

Let us see the use of these notions in some examples. In the example taken from Euclid's *Elements*, several adjectives are defined. The noun *triangle* is defined as a refinement of the noun *figure* using the adjective *trilateral*. We define the adjective `trilateral` with the constructor `Adj`. The `Adj` constructor takes as a parameter the noun to be extended to form the new noun. In the case of our example, trilateral could only be applied to figures as it requires the character sides. The body of Adj is a step (similarly to the Noun descriptor). In this step two specific objects are available. self which refers to the instances of the noun being defined (see section 4.3.2.1.1) and super which refers to the instance of the noun being refined (only needed when a component of the old noun is hidden by a component with the same name of the new noun). After the definition of trilateral, triangle is simply defined as a trilateral figure. We similarly define the adjectives *equilateral*, *isosceles* and *scalene* (We use an infix notation for the identifiers =(term,term):stat and !=(term,term):stat and and(stat,stat):stat).

```
{
  trilateral :=
    Adj (figure) {
                   card(sides) = 3
                 };
  triangle := trilateral figure;
  equilateral :=
    Adj (triangle) {
                     forall(side1:sides,
                       forall(side2:sides,
                         =(side1.length,
                           side2.length))
                   };
  isosceles :=
    Adj (triangle) {
                     exists_one(side1:sides,
                       exists_one(side2:sides,
                         =(side1.length,
                           side2.length)))
                   };
  scalene :=
    Adj (triangle) {
                     forall (side1:sides,
                       forall (side2:sides,
                         !=(side1.length,
                            side2.length)))
                   }
}
```

4.3.2.1.3 Multi adjective refinements With adjectives we have an operation of simple inheritance between nouns. Let us see with this last example how multi adjective refinements work.

The example of Figure 4.2 defines two adjectives. These adjectives for groups are *finite* and *Abelian*. *Finite* states that the set E of the group is finite. *Abelian* (or *commutative*) states that the operator of the group is commutative. In CGa, we write the definitions of these adjectives as follow.

```
{
  finite :=
    Adj (group) {
                  finite_set(E)
                };
  Abelian :=
    Adj (group) {
                  forall(x:E,
                    forall(y:E,
                      =( *(x,y), *(y,x) )))
                 }
}
```

We could combine these two adjectives to obtain either `Abelian finite group` or `finite Abelian group`. In CGa both expressions share the same type. Their meaning may differ as the statements introduced by the adjectives may overlap. It is for instance possible to define an `isosceles equilateral scalene triangle`. This expression is perfectly typable but of course would be considered as inconsistent even by pupils in primary schools. This reflects exactly the purpose of this first layer of MathLang which is to capture the structure of the text and its elements to allow, in a later stage, semantical analysis.

$$\begin{tikzcd}
\texttt{group} \arrow[r, "\texttt{finite}"] \arrow[d, "\texttt{Abelian}"'] & \texttt{finite group} \arrow[d, "\texttt{Abelian}"] \\
\texttt{Abelian group} \arrow[r, "\texttt{finite}"'] & \begin{array}{c}\texttt{Abelian finite group} \\ \texttt{finite Abelian group}\end{array}
\end{tikzcd}$$

4.3.2.2 Weak typing

We discuss in this section the notion of weak typing for MathLang-CGa. We already presented the notion of weak types and weak typing for WTT in Section 2.2.2 on page 26.

Is the CGa typing weak? Yes in the sence that the CGa types are, like WTT ones, atomic types. Unlike most type systems, WTT and CGa's type system do not have a function type. This fact alone creates the weakness of our type system in comparison to the numerous descendants of the Simple Type System [Hin97]. The benefit we get from this weakness is a low level of complexity of the type system. The type system attributes to abstract syntax expressions a concrete type. This gives a static vision of the meaning hosted in the language. This makes CGa authoring accessible but nevertheless checkable as we see in Chapter 5.

Each type in WTT and CGa is inhabited by expressions having little in common. As we saw in Section 3.3.1.2.4, the language does not permit directly to describe a set of sets (an element of such a set being at the same time a term according to its belonging to a set and a set by definition). The solution we saw in Section 3.3.1.2.4, which was proposed by N.G. de Bruijn, is to define a special operator "element of" to downcast a set into a term (`element_of(set):term`). Another solution is to use a special character a term to stand for its "set-selfness". The formula $a \in b \in c$ could therefore be turned into the following expressions where `E` is the character representing this "set-selfness".

```
  {
    c:set( Noun { E:set } );

    b:c;
    a:b.E
  }
```

Could the type signature of identifiers be assimilated to a light arrow type? In Section 4.2.3.1.1, we presented the typing rules for identifiers. They attribute to an identifier a type (element of the set $\mathcal{T}$ from Section 4.2.1.1) composed by a sequence of input atomic types and an output atomic type (i.e. $(a_1, \ldots, a_n) \rightarrow a$). Such type is only attributed to identifiers. No expression could get similar typing (as enforced by the INSTANCE rule on page 87). Functions are not "first class citizens" of the language. Assuming that we declare two identifiers `fun(dec('a),term):term` and `app(term):term`, it is possible to define a function $f(x) = e$ by its λ-term `f := fun(x:term,e)` and later express the application $f(a)$ by `app(f,a)` but the typing does not verify that `f` is a function and that `a`'s type match `f`'s input type. Both `f` and `a` are seen as terms by CGa's type system.

Does the object orientedness make the type system stronger? The object-oriented nature of CGa comes from the object-oriented structure of mathematical texts and particularly nouns and adjectives (see Sections 3.3.1.3 and 4.3.2.1). The language expressiveness is highly extended with this feature but the type system remains a weak type system as the type of any expression or character is atomic.

4.3.3 Meta-theory

In Section 2.2.4 we gave a summary of WTT's meta-theory as developed by F. Kamareddine and R. Nederpelt and presented in [KN04]. We did not attempt to prove the similar properties for CGa for several reasons.

The first reason being that the language kept evolving to fulfill our first requirement of endorsing mathematical authoring. When developing MWTT we could faithfully rely on part of WTT's met-theory as we verified that none of our additional constructions in MWTT were affecting the decidability of weak type checking nor the weak typability. As for the subject reduction and strong normalisation, they are broken by the inclusion of the definition by cases in MWTT. This leads us to our second reason.

The second reason is based on the distance between the goal of CGa and some properties of WTT's meta-theory. The subject reduction and strong normalisation make sense for a language attached to a calculus as these properties are fundamental for the valuation and verification of the reliability of the calculus. CGa is a descriptive language. A typical type system checks the decidability of a calculus where in CGa it simply permits to check the grammatical well-formation. The CGa aspect will certainly be useful for later computerisation which will come with their specific meta-requirements.

The third reason resides in the weakness of the type system. As we explained in Section 4.3.2.2, CGa's types remain weak. The type system did not evolve much since WTT. The major improvement is the extension with object-oriented constructs. This extension and the NOUN and ADJ rules of Section 4.2.3.2.3 require a simple unification for type inference.

4.4 MWTT, a Refinement of WTT

In this section we present MWTT our early work on refining WTT. As explained in Section 2.2.5, the research work for this PhD started with an investigation on the feasibility to translate a mathematical text into WTT. We started to translate the first chapter of E. Landau's *Foundations of Analysis* into WTT. We identify the constructions that makes the specificity of MWTT. We give the reasons from E. Landau's text that lead to these extensions. Most material in this section were published in [KMW04b].

4.4.1 Abstract syntax for MWTT

We present here the abstract syntax of MWTT and the differences with WTT as presented in [NK01, KN04]. The notation we use in this section is a mixture of those used in [NK01, KN04, Maa03, KMW04b].

4.4.1.1 Grammatical categories for MWTT

In MWTT we have five grammatical categories over which g ranges: T for terms, S for sets, N for nouns, A for adjectives and P for statements.

4.4.1.2 Atomic level for MWTT

The identifiers define the atomic level. Variables stand for abstractions over terms, sets or statements. Constants are names for terms, sets, nouns, statements or adjectives. Binders are defined in the language and bind a variable in an expression. Each identifier is part of one grammatical category. In the abstract syntax, the grammatical category of an identifier is indicated by an exponent.

$$\begin{array}{lll} \text{Variables} & v \in \mathcal{V} & ::= \mathcal{V}^T \mid \mathcal{V}^S \mid \mathcal{V}^P \\ \text{Constants} & c \in \mathcal{C} & ::= \mathcal{C}^T \mid \mathcal{C}^S \mid \mathcal{C}^N \mid \mathcal{C}^A \mid \mathcal{C}^P \\ \text{Binders} & b \in \mathcal{B} & ::= \mathcal{B}^T \mid \mathcal{B}^S \mid \mathcal{B}^N \mid \mathcal{B}^A \mid \mathcal{B}^P \end{array}$$

Example 13 (Variables) *In the sentence "Let $\mathfrak{M}$ be a set to which 1 belongs," $\mathfrak{M}$ is a set variable and is denoted by $\mathfrak{M}^S$.*

Example 14 (Constants) *We give here some examples of constants.*

- *Constants of 0-arity: 1^T, $\mathbb{N}^S$, a triangle^N, isosceles^A, false^P.*
- *Constants of greater arity: $+^T$, $\cup^S$, a multiple of^N, divisible by^A, $\wedge^P$.*

Example 15 (Binders) *The expression "a natural number from 1 to 10" describes a noun. We use the $\mathtt{Noun}^N$ (as presented in [KN04], see Section 4.3.1.1) to express it.* $\mathtt{Noun}^N_{n^T:\mathbb{N}^S}(1^T \leqq^T n^T \leqq^T 10^T)$

4.4.1.3 Phrase level for MWTT

The phrase level represents formula-like elements. Expressions are either a variable occurrence, a constant or binder call, or an attribution (i.e. refinement in CGa).

$$
\begin{array}{lll}
\text{Terms} & \mathcal{E}^T & ::= \mathcal{V}^T \mid \mathcal{C}^T(\vec{\mathcal{P}}) \mid \mathcal{B}^T_{\mathcal{Z}}(\mathcal{E}) \\
\text{Sets} & \mathcal{E}^S & ::= \mathcal{V}^S \mid \mathcal{C}^S(\vec{\mathcal{P}}) \mid \mathcal{B}^S_{\mathcal{Z}}(\mathcal{E}) \\
\text{Nouns} & \mathcal{E}^N & ::= \mathcal{C}^N(\vec{\mathcal{P}}) \mid \mathcal{B}^N_{\mathcal{Z}}(\mathcal{E}) \mid \mathcal{E}^A\ \mathcal{E}^N \\
\text{Adjectives} & \mathcal{E}^A & ::= \mathcal{C}^A(\vec{\mathcal{P}}) \mid \mathcal{B}^A_{\mathcal{Z}}(\mathcal{E})
\end{array}
$$

$\mathcal{P}$ and $\mathcal{E}$ are introduced in section 4.4.1.6. $\vec{\mathcal{P}}$stands for a sequence of $\mathcal{P}$.

4.4.1.4 Sentence level for MWTT

The sentence level describes irreducible reasoning components. MWTT has two kinds of definitions similarly to CGa (see Section 4.3.1.2). The case definition (right column) did not exist in WTT.

$$
\begin{array}{llcll}
\text{Statements} & \mathcal{E}^P & ::= & \mathcal{V}^P \mid \mathcal{C}^P(\vec{\mathcal{P}}) \mid \mathcal{B}^P_{\mathcal{Z}}(\mathcal{E}) & \\
\text{Definitions} & \mathcal{D} & ::= & \mathcal{C}^T(\vec{\mathcal{V}}) := \mathcal{E}^T & \mid \mathcal{C}^T(\vec{\mathcal{P}}) := \mathcal{E}^T \\
 & & \mid & \mathcal{C}^S(\vec{\mathcal{V}}) := \mathcal{E}^S & \mid \mathcal{C}^S(\vec{\mathcal{P}}) := \mathcal{E}^S \\
 & & \mid & \mathcal{C}^N(\vec{\mathcal{V}}) := \mathcal{E}^N & \mid \mathcal{C}^N(\vec{\mathcal{P}}) := \mathcal{E}^N \\
 & & \mid & \mathcal{C}^A(\vec{\mathcal{V}}) := \mathcal{E}^A & \mid \mathcal{C}^A(\vec{\mathcal{P}}) := \mathcal{E}^A \\
 & & \mid & \mathcal{C}^P(\vec{\mathcal{V}}) := \mathcal{E}^P & \mid \mathcal{C}^P(\vec{\mathcal{P}}) := \mathcal{E}^P
\end{array}
$$

$\vec{\mathcal{V}}$ stands for a sequence of $\mathcal{V}$.

4.4.1.5 Discourse level for MWTT

The discourse level defines the constructions to structure mathematical texts. Contexts, lines and books were the only elements defined in WTT's discourse level.

$$
\begin{array}{lcll}
\text{Contexts} & \gamma \in \Gamma & ::= & \Gamma_F \mid \Gamma, \mathcal{Z} \mid \Gamma, \mathcal{E}^P \\
\text{Flags} & \Gamma_F & ::= & \Gamma_{FS} \mid \Gamma_F, [\mathcal{Z}] \mid \Gamma_F, [\mathcal{E}^P] \\
\text{Flagstaffs} & \Gamma_{FS} & ::= & \emptyset^{\mathcal{C}} \mid \Gamma_{FS}, \bullet \\
\text{Lines} & l \in \mathcal{L} & ::= & \Gamma \rhd \mathcal{E}^P \mid \Gamma \rhd \mathcal{D} \\
\text{Blocks} & k \in \mathcal{K} & ::= & \emptyset^{\mathcal{K}} \mid \mathcal{K} \circ \mathcal{L} \mid \mathcal{K} \circ \{\mathcal{K}\}_{\vec{\mathcal{C}}} \\
\text{Books} & \mathbf{b} \in \mathbf{B} & ::= & \emptyset^{\mathbf{B}} \mid \mathbf{B} \circ \mathcal{L} \mid \mathbf{B} \circ \{\mathcal{K}\}_{\vec{\mathcal{C}}}
\end{array}
$$

$\vec{\mathcal{C}}$ stands for a sequence of $\mathcal{C}$.

In MWTT flags are composed by a head (a statement or a variable declaration in [ ]) and a flagstaff (several $\bullet$). MWTT included flags as full-fledged element in the abstract syntax. In WTT flags are only a sugared notion. Flags allow to introduce a variable or to make an assumption on several consecutive lines.

Normal notation		Flag notation
$[e_1], [e_2], e_3, e_4$	$\rhd b_1$	$\boxed{e_1}$
$\bullet, \bullet, e_5$	$\rhd b_2$	$\vert\ \boxed{e_2}$
$\bullet, [e_6]$	$\rhd b_3$	$\vert\ \vert\ e_3, e_4 \rhd b_1$
$\bullet, \bullet$	$\rhd b_4$	$\vert\ \vert\ e_5 \rhd b_2$
		$\vert\ \boxed{e_6}$
		$\vert\ \vert\ \rhd b_3$
		$\vert\ \vert\ \rhd b_4$

In MWTT, a block represents a group of consecutive lines and/or blocks. The main feature of blocks is to restrain the scope of some constants (subscript list). In WTT, blocks were not defined.

4.4.1.6 Subsidiary sets

$$\begin{array}{lccl} \text{Declarations} & z \in \mathcal{Z} & ::= & \mathcal{V}^S : \mathrm{SET} \mid \mathcal{V}^P : \mathrm{STAT} \mid \mathcal{V}^T : \mathcal{E}^S \mid \mathcal{V}^T : \mathcal{E}^N \\ \text{Parameters} & \mathcal{P} & ::= & \mathcal{E}^T \mid \mathcal{E}^S \mid \mathcal{E}^P \\ \text{Expressions} & e \in \mathcal{E} & ::= & \mathcal{E}^T \mid \mathcal{E}^S \mid \mathcal{E}^N \mid \mathcal{E}^P \end{array}$$

4.4.2 Weak type system

We define the following weak types (for terms, sets, nouns, adjectives, statements, declarations, definitions, contexts, lines, blocks and books respectively) over which w ranges.

$$\mathtt{T}, \mathtt{S}, \mathtt{N}, \mathtt{A}, \mathtt{P}, \mathtt{Z}, \mathtt{D}, \mathtt{G}, \mathtt{L}, \mathtt{K}, \mathtt{B}.$$

The weak types Z, L and K are MWTT specific. In this section we use a similar notation for typing rules as described in Section 4.2.1.2 where the typing context is a book.

4.4.2.1 Flattening flags

The flag construction is important to enhance the expressiveness of MWTT in comparison with WTT but the flattening of each flag does not change its typing.

Each flagstaff element is replaced by the content of the corresponding flag's head. By flattening we only lose the boundaries of the flag.

4.4.2.2 Functions

We use in the derivation rules some specific functions.

- `dvars` returns the set of variables declared in a given context ($\texttt{dvars} : \Gamma \to \wp(\mathcal{V})$).

$$\begin{array}{rcl}
\texttt{dvars} : \Gamma & \to & \wp(\mathcal{V}) \\
\texttt{dvars}(\emptyset_\Gamma) & = & \emptyset \\
\texttt{dvars}(\gamma, v : e) & = & \texttt{dvars}(\gamma) \cup \{v\} \\
\texttt{dvars}(\gamma, v : \mathrm{SET}) & = & \texttt{dvars}(\gamma) \cup \{v\} \\
\texttt{dvars}(\gamma, v : \mathrm{STAT}) & = & \texttt{dvars}(\gamma) \cup \{v\} \\
\texttt{dvars}(\gamma, \mathrm{p}) & = & \texttt{dvars}(\gamma)
\end{array}$$

- $\texttt{dcons}_{\mathbf{B}}$ returns the set of defined constant in a given book. This excludes local constants ($\texttt{dcons}_{\mathbf{B}} : \mathbf{B} \to \wp(\mathcal{C})$).

$$\begin{array}{rcl}
\texttt{dcons}_{\mathbf{B}} : \mathbf{B} & \to & \wp(\mathcal{C}) \\
\texttt{dcons}_{\mathbf{B}}(\emptyset_{\mathbf{B}}) & = & \emptyset \\
\texttt{dcons}_{\mathbf{B}}(\mathbf{b} \circ \mathrm{p}) & = & \texttt{dcons}_{\mathbf{B}}(\mathbf{b}) \\
\texttt{dcons}_{\mathbf{B}}(\mathbf{b} \circ \mathrm{d}) & = & \texttt{dcons}_{\mathbf{B}}(\mathbf{b}) \cup \texttt{dcons}_{\texttt{D}}(\texttt{d}) \\
\texttt{dcons}_{\mathbf{B}}(\mathbf{b} \circ \{k\}_{c_1,\ldots,c_n}) & = & \texttt{dcons}_{\mathbf{B}}(\mathbf{b}) \cup (\texttt{dcons}_{\mathcal{K}}(k) \ - \ \cup_{i=1}^{n} c_i)
\end{array}$$

- $\texttt{dcons}_{\mathcal{K}}$ returns the set of defined constant in a given block. This excludes constants defined locally in inner blocks ($\texttt{dcons}_{\mathcal{K}} : \mathcal{K} \rightarrow \wp(\mathcal{C})$).

$$\begin{array}{rcl}
\texttt{dcons}_{\mathcal{K}} : \mathcal{K} & \rightarrow & \wp(\mathcal{C}) \\
\texttt{dcons}_{\mathcal{K}}(\emptyset_{\mathcal{K}}) & = & \emptyset \\
\texttt{dcons}_{\mathcal{K}}(k \circ \texttt{p}) & = & \texttt{dcons}_{\mathcal{K}}(k) \\
\texttt{dcons}_{\mathcal{K}}(k \circ \texttt{d}) & = & \texttt{dcons}_{\mathcal{K}}(k) \cup \texttt{dcons}_{\texttt{D}}(\texttt{d}) \\
\texttt{dcons}_{\mathcal{K}}(k \circ \{k'\}_{c_1,\ldots,c_n}) & = & \texttt{dcons}_{\mathcal{K}}(\mathcal{K}) \cup (\texttt{dcons}_{\mathcal{K}}(k') \; - \; \cup_{i=1}^{n} c_i)
\end{array}$$

- $\texttt{dcons}_{\texttt{D}}$ returns the set composed by the constant defined in a given definition construction.

$$\begin{array}{rcl}
\texttt{dcons}_{\texttt{D}} : \texttt{D} & \rightarrow & \wp(\mathcal{C}) \\
\texttt{dcons}_{\texttt{D}}(\gamma \rhd c(v_1, \ldots, v_n) := e) & = & \{c\} \\
\texttt{dcons}_{\texttt{D}}(\gamma \rhd c(e_1, \ldots, e_n) := e) & = & \{c\}
\end{array}$$

- $\texttt{in}_{\mathcal{C}}(i, c, \mathbf{b})$ gives the type of the i^{th} argument of constant c as defined in the book $\mathbf{b}$.
- $\texttt{in}_{\mathcal{B}}(i, b, \mathbf{b})$ represents the type expected by the binder b.
- $\texttt{fvars}$ returns the set of free variables of a given expression ($\texttt{fvars} : \mathcal{E} \rightarrow \wp(\mathcal{V})$).

$$\begin{array}{rcl}
\texttt{fvars} : \mathcal{E} & \rightarrow & \wp(\mathcal{V}) \\
\texttt{fvars}(v) & = & \{v\} \\
\texttt{fvars}(c(e_1, \ldots, e_n)) & = & \cup_{i=1}^{n} \texttt{fvars}(e_i) \\
\texttt{fvars}(b_{v:e'} e) & = & (\texttt{fvars}(e) \backslash v) \cup \texttt{fvars}(e') \\
\texttt{fvars}(b_{v:\mathrm{SET}} e) & = & \texttt{fvars}(e) \backslash v \\
\texttt{fvars}(b_{v:\mathrm{STAT}} e) & = & \texttt{fvars}(e) \backslash v \\
\texttt{fvars}(e_1 \; e_2) & = & \texttt{fvars}(e_1) \cup \texttt{fvars}(e_2)
\end{array}$$

- $\texttt{casedcons}_{\mathbf{B}}$ and $\texttt{casedcons}_{\mathcal{K}}$ return the set of constants defined by cases in a given book, respectively block, excluding local definitions ($\texttt{casedcons}_{\mathbf{B}}$:

$\mathbf{B} \to \wp(\mathcal{C})$ and $\texttt{casedcons}_{\mathcal{K}} : \mathcal{K} \to \wp(\mathcal{C})$). Their definitions follow the definition of $\texttt{dcons}_{\mathbf{B}}$ (respectively $\texttt{dcons}_{\mathcal{K}}$) but differ by returning only the constants defined by cases. Both $\texttt{casedcons}_{\mathbf{B}}$ and $\texttt{casedcons}_{\mathcal{K}}$ make use of $\texttt{casedcons}_{\texttt{D}}$.

$$\begin{array}{rcl}
\texttt{casedcons}_{\mathbf{B}} : \mathbf{B} & \to & \wp(\mathcal{C}) \\
\texttt{casedcons}_{\mathcal{K}} : \mathcal{K} & \to & \wp(\mathcal{C}) \\
\texttt{casedcons}_{\texttt{D}} : \texttt{d} & \to & \wp(\mathcal{C}) \\
\texttt{casedcons}_{\texttt{D}}(\gamma \rhd c(v_1, \ldots, v_n) := e) & = & \emptyset \\
\texttt{casedcons}_{\texttt{D}}(\gamma \rhd c(e_1, \ldots, e_n) := e) & = & \{c\}
\end{array}$$

- OK($\mathbf{b}$; γ) is an abbreviation for $\vdash \mathbf{b} : \texttt{B}$ and $\mathbf{b} \vdash \gamma : \texttt{G}$.

4.4.2.3 Weak typing rules

4.4.2.3.1 Rules for MWTT expressions The rule MWTT-VAR assigns the type corresponding to the variable's grammatical category. The constant rule MWTT-CONS does the same after checking the coherence of the arguments' weak typings. The MWTT-BIND rule, in addition, introduces the new variable in the typing environment of the inner expression. The rule MWTT-ATTR describes how to construct a new noun by attributing an adjective to an existing noun.

$$\frac{\mathrm{OK}(\mathbf{b}\,;\,\gamma) \qquad v \in \mathcal{V}^{T/S/P} \qquad v \in \texttt{dvars}(\gamma)}{\mathbf{b}\,;\,\gamma \vdash v : \texttt{T/S/P}}\ \textsc{mwtt-var}$$

$$\frac{\begin{array}{c}\mathrm{OK}(\mathbf{b}\,;\,\gamma) \qquad c \in \mathcal{C}^{T/S/N/A/P} \\ c \in \texttt{dcons}_{\mathbf{B}}(\mathbf{b}) \qquad \forall i \in \{1, \ldots, n\},\ \mathbf{b}\,;\,\gamma \vdash e_i : \texttt{in}_{\mathcal{C}}(i, c, \mathbf{b})\end{array}}{\mathbf{b}\,;\,\gamma \vdash c(e_1, \ldots, e_n) : \texttt{T/S/N/A/P}}\ \textsc{mwtt-cons}$$

$$\frac{\mathrm{OK}(\mathbf{b}\,;\,\gamma) \qquad b \in \mathcal{B}^{T/S/N/A/P} \qquad \mathbf{b}\,;\,\gamma, z \vdash e : \texttt{in}_{\mathcal{B}}(b)}{\mathbf{b}\,;\,\gamma \vdash b_z(e) : \texttt{T/S/N/A/P}}\ \textsc{mwtt-bind}$$

$$\frac{\mathbf{b}\,;\,\gamma \vdash e_1 : \texttt{A} \qquad \mathbf{b}\,;\,\gamma \vdash e_2 : \texttt{N}}{\mathbf{b}\,;\,\gamma \vdash e_1 e_2 : \texttt{N}}\ \textsc{mwtt-attr}$$

4.4.2.3.2 Rules for MWTT contexts These rules check the coherence of contexts. The first one MWTT-EMPTY-CONT states that an empty context is a valid context. The last one MWTT-ASSUMP checks if an assumption in the context is a well formed statement expression. The three remaining rules MWTT-SET-VAR-DECL, MWTT-STAT-VAR-DECL and MWTT-TERM-VAR-DECL correspond to the three constructions of variable introduction.

$$\frac{\vdash \mathbf{b} \mathbf{:} \mathtt{B}}{\mathbf{b} \vdash \emptyset_\Gamma \mathbf{:} \mathtt{G}}\ \textsc{mwtt-empty-cont}$$

$$\frac{\mathrm{OK}(\mathbf{b}\,;\,\gamma) \qquad v \in \mathcal{V}^S \qquad v \notin \mathtt{dvars}(\gamma)}{\mathbf{b} \vdash \gamma, v : \mathrm{SET} \mathbf{:} \mathtt{G}}\ \textsc{mwtt-set-var-decl}$$

$$\frac{\mathrm{OK}(\mathbf{b}\,;\,\gamma) \qquad v \in \mathcal{V}^P \qquad v \notin \mathtt{dvars}(\gamma)}{\mathbf{b} \vdash \gamma, v : \mathrm{STAT} \mathbf{:} \mathtt{G}}\ \textsc{mwtt-stat-var-decl}$$

$$\frac{\begin{array}{c}\mathrm{OK}(\mathbf{b}\,;\,\gamma)\\ v \in \mathcal{V}^T \qquad v \notin \mathtt{dvars}(\gamma) \qquad \mathbf{b}\,;\,\gamma \vdash e \mathbf{:} \mathtt{S/N}\end{array}}{\mathbf{b} \vdash \gamma, v : e \mathbf{:} \mathtt{G}}\ \textsc{mwtt-term-var-decl}$$

$$\frac{\mathrm{OK}(\mathbf{b}\,;\,\gamma) \qquad \mathbf{b}\,;\,\gamma \vdash \mathtt{p} \mathbf{:} \mathtt{P}}{\mathbf{b} \vdash \gamma, \mathtt{p} \mathbf{:} \mathtt{G}}\ \textsc{mwtt-assump}$$

4.4.2.3.3 Rules for MWTT definitions There are two kinds of definitions in the abstract syntax with three typing rules. One rule MWTT-FULL-DEF is for the basic constant definition which provides a unique expression as definiens for the constant. The two remaining rules MWTT-CASE-DEF-FIRST and MWTT-CASE-DEF-ALTER are for definitions by cases and are MWTT specific.

$$\dfrac{\begin{array}{c}\mathrm{OK}(\mathbf{b}\,;\,\gamma)\\ c \in \mathcal{C}^{T/S/N/A/P} \qquad c \notin \mathtt{dcons}_{\mathbf{B}}(\mathbf{b}) \qquad \mathtt{dvars}(\gamma) = \cup_{i=1}^{n} v_i\\ \forall i \in \{1,\ldots,n\},\ v_i \in \mathcal{V}^{g_i} \qquad \mathbf{b}\,;\,\gamma \vdash e \mathbin{\mathbf{:}} \mathtt{T/S/N/A/P}\end{array}}{\mathbf{b}\,;\,\gamma \vdash c(v_1,\ldots,v_n) := e \mathbin{\mathbf{:}} \mathtt{D}}\ \textsc{mwtt-full-def}$$

$$\dfrac{\begin{array}{c}\mathrm{OK}(\mathbf{b}\,;\,\gamma) \qquad c \in \mathcal{C}^{T/S/N/A/P}\\ c \notin \mathtt{casedcons}_{\mathbf{B}}(\mathbf{b}) \qquad \mathtt{dvars}(\gamma) = \cup_{i=1}^{n} \mathtt{fvars}(e_i)\\ \forall i \in \{1,\ldots,n\},\ \mathbf{b}\,;\,\gamma \vdash e_i \mathbin{\mathbf{:}} w_i \qquad \mathbf{b}\,;\,\gamma \vdash e \mathbin{\mathbf{:}} \mathtt{T/S/N/A/P}\end{array}}{\mathbf{b}\,;\,\gamma \vdash c(e_1,\ldots,e_n) := e \mathbin{\mathbf{:}} \mathtt{D}}\ \textsc{mwtt-case-def-first}$$

$$\dfrac{\begin{array}{c}\mathrm{OK}(\mathbf{b}\,;\,\gamma) \qquad c \in \mathcal{C}^{T/S/N/A/P}\\ c \in \mathtt{casedcons}_{\mathbf{B}}(\mathbf{b}) \qquad \mathtt{dvars}(\gamma) = \cup_{i=1}^{n} \mathtt{fvars}(e_i)\\ \forall i \in \{1,\ldots,n\},\ \mathbf{b}\,;\,\gamma \vdash e_i \mathbin{\mathbf{:}} \mathtt{in}_{\mathcal{C}}(i, c, \mathbf{b})\\ \mathbf{b}\,;\,\gamma \vdash e \mathbin{\mathbf{:}} \mathtt{T/S/N/A/P}\end{array}}{\mathbf{b}\,;\,\gamma \vdash c(e_1,\ldots,e_n) := e \mathbin{\mathbf{:}} \mathtt{D}}\ \textsc{mwtt-case-def-alter}$$

4.4.2.3.4 Rules for MWTT blocks An empty block is a valid according to rule MWTT-EMPTY-BLOCK. Additionally, the rules MWTT-EMPTY-LINE-IN-BLOCK and MWTT-EMPTY-BLOCK-IN-BLOCK indicate how a valid block can be extended with a line and a block respectively.

$$\dfrac{\vdash \mathbf{b} \mathbin{\mathbf{:}} \mathtt{B}}{\mathbf{b} \vdash \emptyset_{\mathcal{K}} \mathbin{\mathbf{:}} \mathtt{K}}\ \textsc{mwtt-empty-block}$$

$$\dfrac{\mathbf{b} \vdash k \mathbin{\mathbf{:}} \mathtt{K} \qquad \mathbf{b} \circ \{k\}_{\emptyset}\,;\,\gamma \vdash \mathtt{p/d} \mathbin{\mathbf{:}} \mathtt{P/D}}{\mathbf{b} \vdash k \circ \gamma \rhd \mathtt{p/d} \mathbin{\mathbf{:}} \mathtt{K}}\ \textsc{mwtt-line-in-block}$$

$$\dfrac{\begin{array}{c}\vdash \mathbf{b} \mathbin{\mathbf{:}} \mathtt{B} \qquad \mathbf{b} \vdash k \mathbin{\mathbf{:}} \mathtt{K}\\ \{c_1,\ldots,c_n\} \subseteq \mathtt{dcons}_{\mathcal{K}}(k') \qquad \mathbf{b} \circ \{k\}_{\emptyset} \vdash k' \mathbin{\mathbf{:}} \mathtt{K}\end{array}}{\mathbf{b} \vdash k \circ \{k'\}_{c_1,\ldots,c_n} \mathbin{\mathbf{:}} \mathtt{K}}\ \textsc{mwtt-block-in-block}$$

4.4.2.3.5 Rules for MWTT books The rules for books MWTT-EMPTY-BOOK, MWTT-EMPTY-LINE-IN-BOOK and MWTT-EMPTY-BLOCK-IN-BOOK are similar to the rules for blocks. A book could be seen as the outermost block.

$$\frac{}{\vdash \emptyset_{\mathbf{B}} \mathbin{:} \mathtt{B}}\ \textsc{mwtt-empty-book}$$

$$\frac{\vdash \mathbf{b} \mathbin{:} \mathtt{B} \qquad \mathbf{b}\,;\,\gamma \vdash \mathtt{p/d} \mathbin{:} \mathtt{P/D}}{\vdash \mathbf{b} \circ \gamma \rhd \mathtt{p/d} \mathbin{:} \mathtt{B}}\ \textsc{mwtt-line-in-book}$$

$$\frac{\vdash \mathbf{b} \mathbin{:} \mathtt{B} \qquad \{c_1, \ldots, c_n\} \subseteq \mathtt{dcons}_{\mathcal{K}}(k) \qquad \mathbf{b} \vdash k \mathbin{:} \mathtt{K}}{\vdash \mathbf{b} \circ \{k\}_{c_1,\ldots,c_n} \mathbin{:} \mathtt{B}}\ \textsc{mwtt-block-in-book}$$

Conclusion

In this chapter we defined the theoretical CGa system that is to say an object-oriented language for CGa and a weak type system to check CGa documents' well formation. We also presented MWTT which is a refinement of WTT developed during this PhD early studies.

Chapter 5

MathLang-TSa, Authoring Method and Experience

This chapter focuses on describing the interface for encoding mathematics with MathLang support. We firstly describes in Section 5.1 our platform-independent method to create or turn a natural language text into a TSa-CGa MathLang document (following MathLang-CGa defined in Chapter 4). We then extend in Section 5.2 this method to provide a manner to explicate the morphology of some specific natural language abbreviations and notations. A number of notations such as aggregated inequations (i.e., $a = b < c$), joint declarations (i.e., "let x and y be natural numbers") or alternative styles (i.e., $a \in R$ and R contains a) can be input and their meaning attached easily with this method. We illustrate in Section 5.3 the use of this method for encoding several mathematical texts. Most material in this chapter were published in [KLMW07].

5.1 TSa's Natural Language Annotation

We propose an authoring technique which uses mathematical natural language as primary input instead of trying to replace it. As an author composes a document on computer, it is desirable to truly derive any formal or symbolic version from the natural way the mathematicians sees his document which is a succession of arranged symbols and text chunks. We propose to decorate this natural original text with extra information. This extra content has to be more precise, complete and computation-friendly than natural language. With such extra information intermingle with the original text, it is possible to ensure subsequent translations to be consistent and faithful to natural language text.

term	Common mathematical objects.	"$a + b$"
set	Sets of mathematical objects.	"$\mathbb{N}$"
noun	Families of **terms**.	"ring"
adjective	**Noun** refiners.	"Abelian"
statement	Affirmations, arguments, properties, assertions, ...	"$a + 0 = a$"
declaration	Introductions of new symbols or notions.	"Let a be ..."
definition	Explanations of the meaning of new symbols notions.	"A ring is ..."
step	A group of mathematical assertions.	"..., therefore ..."
context	Preliminary assertions prior to a **step**.	"Assume ..."

Table 5.1: TSa box annotations' colour coding system

5.1.1 Box annotation

Our approach augment natural language text with supplementary information. We do so by wrapping pieces of text with *annotation boxes*. To each box is associated some meta-information which describes the grammatical role played by the wrapped text in the document's argumentation. The background colour of an annotation box gives the CGa grammatical category to which the wrapped text belongs. Table 5.1 contains the colour coding system we use. It is important to notice that once we remove these annotation boxes we find the text completely unchanged. Let us illustrate the use of annotation boxes with the sentence "There is an element 0 in R such that $a + 0 = a$" extracted from a textbook in abstract algebra [Gal02, Chapter 12]. The grammatical information, in terms of CGa's grammatical constructions, that we can easily infer from the original text is shown by the following annotation boxes. The boxes surrounding "an element 0", "a", "0" and "$a+0$" indicate that these expressions are **terms**. "R" is wrapped in a **set** box and "an element 0 in R" in a **declaration** box. The box surrounding "$a + 0 = a$" indicates that this equation is a **statement**. The whole sentence is put in a **step** box.

$$\boxed{\text{There is } \boxed{\boxed{\text{an element } 0} \text{ in } \boxed{R}} \text{ such that } \boxed{\boxed{\boxed{a} + \boxed{0}} = \boxed{a}}}$$

Such box-annotation is similar to an explicit typing of values in programming if we consider our grammatical categories equivalent to types. The equation representing this example would be written as follows in a pseudo programming language.

```
(eq((plus(a:term,0:term )):term,a:term)):statement
```

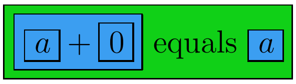
It differs from traditional programming with type inference where the user gives the information of the type signatures of identifiers only once. In such language with type inference our example would be written as follows.

```
eq(plus(a,0),a)
```

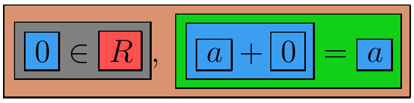
The type of `eq`, `plus`, `a` and `0` would therefore be inferred or retrieved from the context by a type inference system. Explicit typing of values differs also from explicit typing of identifiers where the type of an identifier is provided for each instance of this identifier. Our example would be written as follows in an explicit typing of identifiers.

```
(eq:term -> term -> statement)
     ((plus:term -> term -> term) (a:term) (0:term))
     (a:term)
```

If we compare the pseudo-language code `eq(plus(a,0),a)` and its box-annotated natural language equivalent, we can see that their namespaces differ. The symbol "+" corresponds to the identifier `plus`. One might argue that the symbols `=` and `+` could have been used with infix notation and relevant symbols' precedence. We would have obtained an expression `a+0=a` very similar to the natural language sentence's equation. But imagine a situation where, instead of stating the equality between $a + 0$ and a by an equation, the authors prefers to use of the verb "equal". The sentence would be printed differently but would still mean that `a+0=a`.

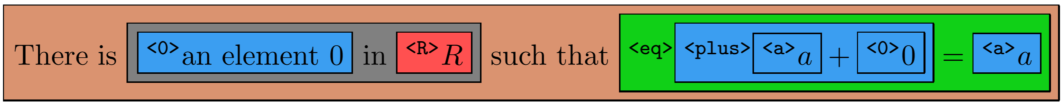


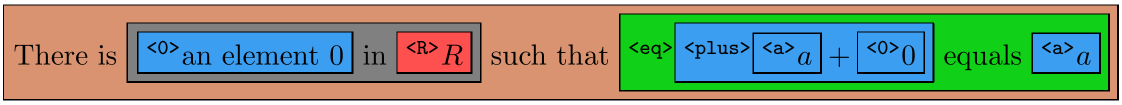
An equation and its natural language equivalent should reflect the same meaning (`a+0=a` in our example). Similarly a natural language sentence and its equivalent formula should get similar box annotations. Our sentence could look as follows and still be annotated with the same boxes.

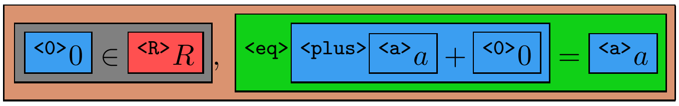


5.1.2 Interpretation

To establish the meaning of the text contained in each annotation box, we attribute to each box its *interpretation* in terms of CGa grammatical expressions. For example the boxes surrounding "an element 0" and "0" get `0` as an interpretation attribute which means that they should be interpreted as an identifier named `0`. The box surrounding "R" (respectively a, $a+0$ and the equation) get `R` (respectively `a`, `plus` and `eq`) as an interpretation attribute. Each interpretation attribute is printed in a typewriter typeface on the left hand side of the annotation box between the `<` and `>` angle brackets. Immediately below are the versions of our three sentences with interpretation attributes.



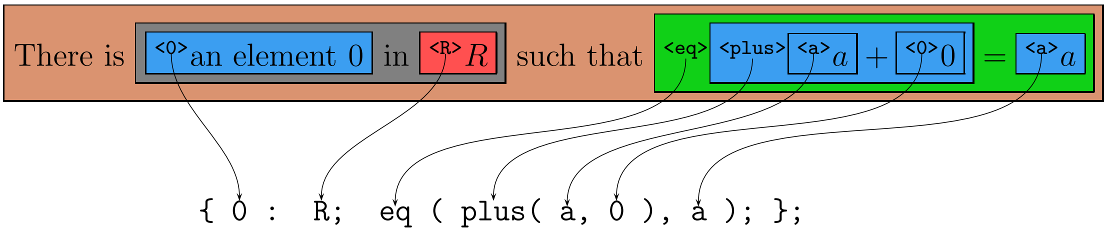




With these examples we see that CGa's grammar is not a natural language grammar but a grammar for mathematical *justifications* (following N.G. de Bruijn's terminology [dB91a]) or argumentations. To make the editing and update of the interpretation attribute of box annotations easier. Its is recommended to have coherent naming of identifiers.

5.1.3 Nested annotations

In our example we see also that some boxes are inside other boxes. In the case of our equation, each inner box is interpreted as an argument for its surrounding box. The nesting of boxes indicates that some annotated expressions are subexpressions of others. It is therefore a straightforward automatic process to create a CGa grammatical expression out of a text with box annotations. We illustrate this with our sentence-example.



```
{ 0 :  R;  eq ( plus( a, 0 ), a ); };
```

We show here the CGa grammatical expression corresponding to our box-annotated text. This expression is written using the abstract syntax presented in Section 4.1. Note that this syntax is not meant to be used by the end-user of CGa, it is only designed for theoretical discussion on CGa's grammar. The CGa end-user edits his natural language text with annotation boxes. The internal syntax used in our implementations follows XML recommendations (see Section 6.1.1).

5.1.4 Annotations, a syntax for MathLang-CGa

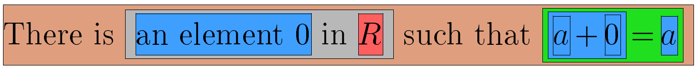
The annotation boxes play the role of a user-oriented input syntax for CGa. We show here that this annotation language could be used as a concrete syntax for input in CGa. Table 5.2 contains the definition of denotation semantic $\llbracket\ \rrbracket^a$ which expresses the meaning of a annotation document in terms of documents written in the CGa's abstract syntax presented in Section 4.1. Table 5.3 presents the reverse denotational semantic $\llbracket\ \rrbracket^b$ which expresses the meaning of abstract syntax documents in terms of annotation boxes.

Remark 9 *We do not have an isomorphism between these two syntaxes. For every annotation document* $d, \llbracket \llbracket d \rrbracket^a \rrbracket^b \neq d$ *and for every abstract syntax document* $a, \llbracket \llbracket a \rrbracket^b \rrbracket^a \neq a$. *This is due to the fact that the annotation syntax is not meant to be complete but is designed to satisfy the mathematicians needs. One of the restriction that does not permit to define such isomorphism is pointed out by foot note 2 of Table 5.3.*

5.1.5 Automatic grammatical analysis

We implemented this authoring method with annotation boxes as a plugin for the scientific text editor $\mathrm{T_EX_{MACS}}$, see Section 6.3.1 for technical description of this plugin and Appendix C for the integrated documentation of the plugin. During or after the editing of the natural language text, the mathematician is asked to wrap relevant pieces of text in TSa's annotation boxes. Continuing with our sentence-example, the user of the MathLang plugin for $\mathrm{T_EX_{MACS}}$ easily obtains, among others, the following views.

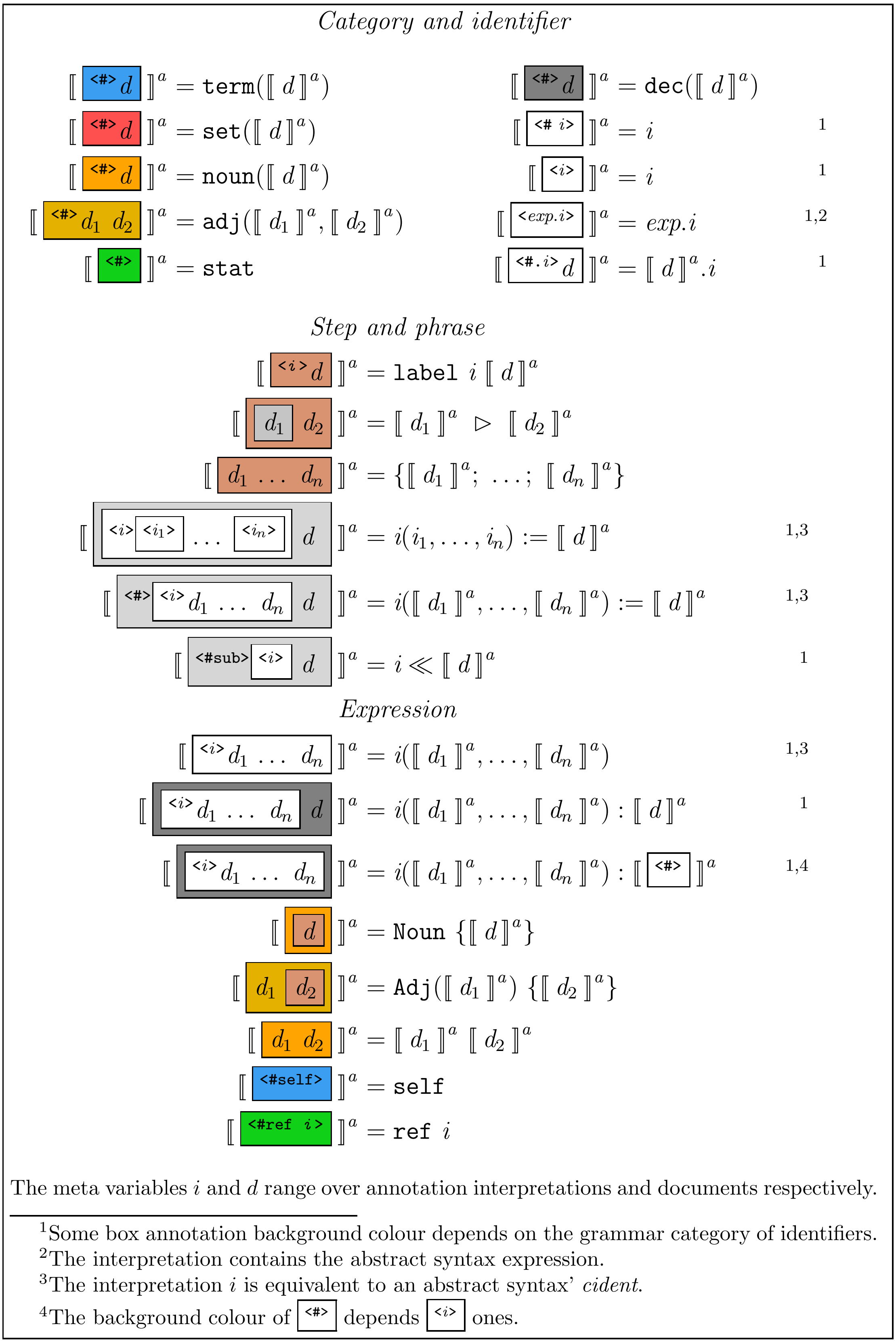
An annotation view with annotation boxes printed as coloured boxes:



There is an element 0 in R such that $a + 0 = a$

A standard view without annotation boxes:

There is an element 0 in R such that $a + 0 = a$



Category and identifier

🟦 $[\![\, \boxed{\texttt{<\#>}\, d} \,]\!]^a = \texttt{term}([\![\, d \,]\!]^a)$	⬛ $[\![\, \boxed{\texttt{<\#>}\, d} \,]\!]^a = \texttt{dec}([\![\, d \,]\!]^a)$	
🟥 $[\![\, \boxed{\texttt{<\#>}\, d} \,]\!]^a = \texttt{set}([\![\, d \,]\!]^a)$	$[\![\, \boxed{\texttt{<\#}\ i\texttt{>}} \,]\!]^a = i$	1
🟧 $[\![\, \boxed{\texttt{<\#>}\, d} \,]\!]^a = \texttt{noun}([\![\, d \,]\!]^a)$	$[\![\, \boxed{\texttt{<}i\texttt{>}} \,]\!]^a = i$	1
🟨 $[\![\, \boxed{\texttt{<\#>}\, d_1\ d_2} \,]\!]^a = \texttt{adj}([\![\, d_1 \,]\!]^a, [\![\, d_2 \,]\!]^a)$	$[\![\, \boxed{\texttt{<}exp.i\texttt{>}} \,]\!]^a = exp.i$	1,2
🟩 $[\![\, \boxed{\texttt{<\#>}} \,]\!]^a = \texttt{stat}$	$[\![\, \boxed{\texttt{<\#.}i\texttt{>}\, d} \,]\!]^a = [\![\, d \,]\!]^a.i$	1

Step and phrase

$$[\![\, \boxed{\texttt{<}i\texttt{>}\, d} \,]\!]^a = \texttt{label}\ i\ [\![\, d \,]\!]^a$$

$$[\![\, \boxed{\boxed{d_1}\ d_2} \,]\!]^a = [\![\, d_1 \,]\!]^a \ \triangleright\ [\![\, d_2 \,]\!]^a$$

$$[\![\, \boxed{d_1\ \ldots\ d_n} \,]\!]^a = \{[\![\, d_1 \,]\!]^a;\ \ldots;\ [\![\, d_n \,]\!]^a\}$$

$$[\![\, \boxed{\boxed{\texttt{<}i\texttt{>}\boxed{\texttt{<}i_1\texttt{>}}\ \ldots\ \boxed{\texttt{<}i_n\texttt{>}}}\ d} \,]\!]^a = i(i_1, \ldots, i_n) := [\![\, d \,]\!]^a$$ 1,3

$$[\![\, \boxed{\texttt{<\#>}\boxed{\texttt{<}i\texttt{>}\, d_1\ \ldots\ d_n}\ d} \,]\!]^a = i([\![\, d_1 \,]\!]^a, \ldots, [\![\, d_n \,]\!]^a) := [\![\, d \,]\!]^a$$ 1,3

$$[\![\, \boxed{\texttt{<\#sub>}\boxed{\texttt{<}i\texttt{>}}\ d} \,]\!]^a = i \ll [\![\, d \,]\!]^a$$ 1

Expression

$$[\![\, \boxed{\texttt{<}i\texttt{>}\, d_1\ \ldots\ d_n} \,]\!]^a = i([\![\, d_1 \,]\!]^a, \ldots, [\![\, d_n \,]\!]^a)$$ 1,3

$$[\![\, \boxed{\boxed{\texttt{<}i\texttt{>}\, d_1\ \ldots\ d_n}\ d} \,]\!]^a = i([\![\, d_1 \,]\!]^a, \ldots, [\![\, d_n \,]\!]^a) : [\![\, d \,]\!]^a$$ 1

$$[\![\, \boxed{\boxed{\texttt{<}i\texttt{>}\, d_1\ \ldots\ d_n}} \,]\!]^a = i([\![\, d_1 \,]\!]^a, \ldots, [\![\, d_n \,]\!]^a) : [\![\, \boxed{\texttt{<\#>}} \,]\!]^a$$ 1,4

$$[\![\, \boxed{\boxed{d}} \,]\!]^a = \texttt{Noun}\ \{[\![\, d \,]\!]^a\}$$

$$[\![\, \boxed{d_1\ \boxed{d_2}} \,]\!]^a = \texttt{Adj}([\![\, d_1 \,]\!]^a)\ \{[\![\, d_2 \,]\!]^a\}$$

$$[\![\, \boxed{d_1\ d_2} \,]\!]^a = [\![\, d_1 \,]\!]^a\ [\![\, d_2 \,]\!]^a$$

$$[\![\, \boxed{\texttt{<\#self>}} \,]\!]^a = \texttt{self}$$

$$[\![\, \boxed{\texttt{<\#ref}\ i\texttt{>}} \,]\!]^a = \texttt{ref}\ i$$

The meta variables i and d range over annotation interpretations and documents respectively.

[1] Some box annotation background colour depends on the grammar category of identifiers.
[2] The interpretation contains the abstract syntax expression.
[3] The interpretation i is equivalent to an abstract syntax' *cident*.
[4] The background colour of $\boxed{\texttt{<\#>}}$ depends $\boxed{\texttt{<}i\texttt{>}}$ ones.

Table 5.2: TSa's annotation boxes denoted in terms of CGa's abstract syntax. Denotation function $[\![\ ]\!]^a$ transforming annotation boxes into abstract syntax expressions.

Category and identifier

$$\llbracket\, \texttt{term}(e) \,\rrbracket^b = \boxed{\texttt{<\#>}\llbracket\, e \,\rrbracket^b}$$

$$\llbracket\, \texttt{set}(e) \,\rrbracket^b = \boxed{\texttt{<\#>}\llbracket\, e \,\rrbracket^b}$$

$$\llbracket\, \texttt{noun}(e) \,\rrbracket^b = \boxed{\texttt{<\#>}\llbracket\, e \,\rrbracket^b}$$

$$\llbracket\, \texttt{adj}(e_1, e_2) \,\rrbracket^b = \boxed{\texttt{<\#>}\llbracket\, e_1 \,\rrbracket^b \llbracket\, e_2 \,\rrbracket^b}$$

$$\llbracket\, \texttt{stat} \,\rrbracket^b = \boxed{\texttt{<\#>}}$$

$$\llbracket\, \texttt{dec}(c) \,\rrbracket^b = \boxed{\texttt{<\#>}\llbracket\, c \,\rrbracket^b}$$

$$\llbracket\, v \,\rrbracket^b = \boxed{\texttt{<\#}\ v\texttt{>}} \qquad 1$$

$$\llbracket\, i \,\rrbracket^b = \boxed{\texttt{<}i\texttt{>}} \qquad 1$$

$$\llbracket\, e.i \,\rrbracket^b = \begin{cases} \boxed{\texttt{<}e.i\texttt{>}} \\ \boxed{\texttt{<\#.}i\texttt{>}\llbracket\, e \,\rrbracket^b} \end{cases} \qquad 1,2$$

Step and phrase

$$\llbracket\, \texttt{label}\ l\ s \,\rrbracket^b = \boxed{\texttt{<}l\texttt{>}\llbracket\, s \,\rrbracket^b}$$

$$\llbracket\, s_1 \ \triangleright\ s_2 \,\rrbracket^b = \boxed{\boxed{\llbracket\, s_1 \,\rrbracket^b} \llbracket\, s_2 \,\rrbracket^b}$$

$$\llbracket\, \{s_1;\ \ldots;\ s_n\} \,\rrbracket^b = \boxed{\llbracket\, s_1 \,\rrbracket^b \ \ldots\ \llbracket\, s_n \,\rrbracket^b}$$

$$\llbracket\, ci(i_1, \ldots, i_n) := e \,\rrbracket^b = \boxed{\boxed{\texttt{<}ci\texttt{>}\boxed{\texttt{<}i_1\texttt{>}}\ \ldots\ \boxed{\texttt{<}i_n\texttt{>}}} \llbracket\, e \,\rrbracket^b} \qquad 1$$

$$\llbracket\, ci(e_1, \ldots, e_n) := e \,\rrbracket^b = \boxed{\texttt{<\#>}\boxed{\texttt{<}ci\texttt{>}\llbracket\, e_1 \,\rrbracket^b\ \ldots\ \llbracket\, e_n \,\rrbracket^b} \llbracket\, e \,\rrbracket^b} \qquad 1$$

$$\llbracket\, i \ll e \,\rrbracket^b = \boxed{\texttt{<\#sub>}\boxed{\texttt{<}i\texttt{>}}\ \llbracket\, e \,\rrbracket^b} \qquad 1$$

Expression

$$\llbracket\, ci(e_1, \ldots, e_n) \,\rrbracket^b = \boxed{\texttt{<}ci\texttt{>}\llbracket\, e_1 \,\rrbracket^b\ \ldots\ \llbracket\, e_n \,\rrbracket^b} \qquad 1$$

$$\llbracket\, i(ce_1, \ldots, ce_n) : e \,\rrbracket^b = \boxed{\boxed{\texttt{<}i\texttt{>}\llbracket\, ce_1 \,\rrbracket^b\ \ldots\ \llbracket\, ce_n \,\rrbracket^b} \llbracket\, e \,\rrbracket^b} \qquad 1$$

$$\llbracket\, i(ce_1, \ldots, ce_n) : c \,\rrbracket^b = \boxed{\boxed{\texttt{<}i\texttt{>}\llbracket\, ce_1 \,\rrbracket^b\ \ldots\ \llbracket\, ce_n \,\rrbracket^b}} \qquad 1,3$$

$$\llbracket\, \texttt{Noun}\ \{s\} \,\rrbracket^b = \boxed{\boxed{\llbracket\, s \,\rrbracket^b}}$$

$$\llbracket\, e_1\ e_2 \,\rrbracket^b = \boxed{\llbracket\, e_1 \,\rrbracket^b \llbracket\, e_2 \,\rrbracket^b}$$

$$\llbracket\, \texttt{self} \,\rrbracket^b = \boxed{\texttt{<\#self>}}$$

$$\llbracket\, \texttt{Adj}(e)\ \{s\} \,\rrbracket^b = \boxed{\llbracket\, e \,\rrbracket^b \boxed{\llbracket\, s \,\rrbracket^b}}$$

$$\llbracket\, \texttt{ref}\ l \,\rrbracket^b = \boxed{\texttt{<\#ref}\ l\texttt{>}}$$

We use the meta variables e, c, v, i, l, s and ci defined in Section 4.1. The meta variable ce ranges over both categories and expressions.

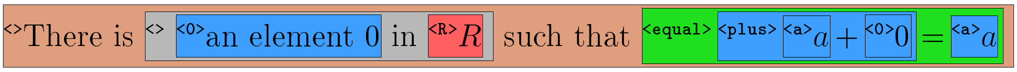
[1] Some box annotation background colour depends on the grammar category of identifiers.

[2] The interpretation contains the abstract syntax expression.

[3] The background colour of $\boxed{\texttt{<}i\texttt{>}}$ depends the category c. The annotation syntax permits only the situation where c is a simple category.

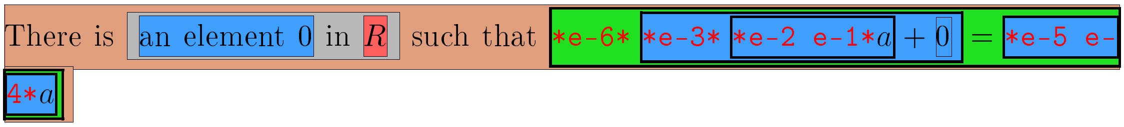
Table 5.3: CGa's abstract syntax denoted in terms TSa's annotation boxes. Denotation function $\llbracket\ \rrbracket^b$ transforming abstract syntax expressions into annotation boxes.

An interpretation view with annotation boxes printed as coloured boxes and interpretations printed on the top left corner in between the angle brackets < and >:

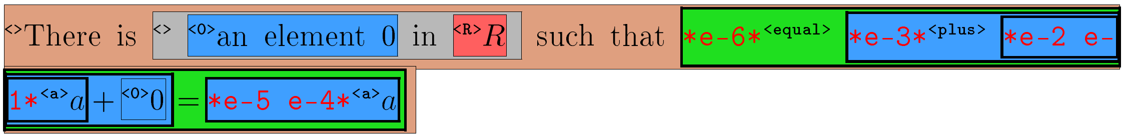


<>There is <> <0>an element 0 in <R>R such that <equal><plus><a>a + <0>0 = <a>a

The MathLang plugin for TeXmacs communicates the content of the document to CGa's checker we present Section 6.2 and based on CGa's type system defined in Section 4.2. Let us assume that R, $=$ and $+$ were properly introduced in the larger document. When the user is satisfied with his annotation of the sentence, the TeXmacs plugin is instructed to send the entire document to the type checker. The checker analyses the grammatical structure of the CGa document and finds out that a has not been properly introduced in our sentence-example. A set of errors[1] with their locations in the TeXmacs document are sent back to the plugin to be shown to the user. Here are two views of the text with the errors' labels printed in between stars $*$.



There is an element 0 in R such that *e-6* *e-3* *e-2 e-1*a + 0 = *e-5 e-4*a



<>There is <> <0>an element 0 in <R>R such that *e-6*<equal> *e-3*<plus> *e-2 e-1*<a>a + <0>0 = *e-5 e-4*<a>a

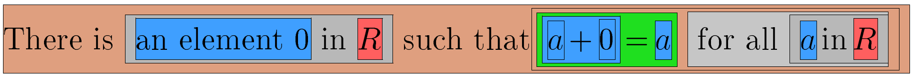
The description of each error label is also provided by the plugin in the MathLang plugin menu as shown here.

```
Error (e-1):
  Anticipated instance of "a"
Error (e-2):
  Categories mismatch, Unspecified expected, not term.
Error (e-3):
  Types mismatch for "plus",
   (term,term):term expected, not (Unspecified,term):term.
Error (e-4):
  Anticipated instance of "a"
```

[1] The high number of errors is due to the fact that the checking of the document does not stop after one error is found but analyses the entire document. An error may point at several locations in the document, in order to cover all the expressions involved in a typing error.

```
Error (e-5):
  Categories mismatch, Unspecified expected, not term.
Error (e-6):
  Types mismatch for "equal",
   (term,term):stat expected, not (term,Unspecified):stat.
```

To fix these errors we simply define a. The extra "for all a in R" text is wrapped in a **context** box annotation which indicates that it forms the context of the equation (this is one possible way to annotate this extra expression).



There is an element 0 in R such that $a+0=a$ for all a in R

The sentence we obtain is now valid in accordance to CGa grammar and type checker. This sentence also corresponds to the original sentence we found in [Gal02, Chapter 12].

5.2 TSa's Annotation Shuffling

The grammatical box annotation as presented in Section 5.1 is guided by the style in which the original natural-language sentences were written. One can easily find a piece of mathematical text that does not permit such simplistic annotations. The writing style used by mathematicians is usually *uneven* in regard to the structure of the underlying meaning described, and by extent uneven in regard to the grammatical box annotations. Authors usually simplify some expressions or rearrange some explanations for the sake of facilitating and also entertaining the reader. Such writing style helps any reader to focus on the actual knowledge contained in the text. To adapt to any style, it is therefore necessary to extend our annotation language with some extra indication on how the author's style has to be interpreted. The author's writing style does not dramatically change the structure of the mathematical reasoning contained in the text which means that the grammatical role of each piece of text remains clear, especially to the author himself.

5.2.1 Syntax souring

These grammatical annotations presented in Section 5.1 are applied to a text written in the mathematical natural language. This natural language text is the language in which the mathematician is used to communicate mathematical knowledge (see Sections 2.4.1.1 and 3.2.1.1). But this language is highly automation-unfriendly for computer software. We showed in Section 5.1.1, and in [KMW04a] that CGa has constructions that correspond to the way common mathematical justifications are structured. The CGa language defined in Chapter 4 is automation-friendly and mimics the mathematical natural language structure of justification. Therefore CGa's authoring does not require the user to tweak, alter, or translate the document's knowledge for computerisation, although there is a need to tweak the writing style when encoding text directly into the core CGa language. Because we regard our starting material, natural language, to be the *sweetest* for human readers, we call this tweaking *syntax souring.* This term describes the process of rewriting natural language into a syntactically formalised language. We call *souring annotations*, the additives needed to describe how to perform a transformation of natural language to a core formalised language.

5.2.1.1 Syntax sugaring

The notion of *syntax sugaring* is well known by programmers. It was coined by P. J. Landin. Syntactic sugar is added to the syntax of programming languages to make it easier to use by humans. Syntax sugaring does not affect the expressiveness of the language but lightens its syntax. Syntactic sugar is usually an additive for the syntax of a formal language. This formal language is the target language that needs to be lightened for a human to input it. *De-sugaring* is the process of getting rid of the sugared bits by replacing them with proper expressions written in the core language syntax.

$$\boxed{\begin{array}{c}\text{Programming language}\\ +\\ \text{Syntactic sugar}\end{array} \quad \xrightarrow{\textit{de-sugaring}} \quad \begin{array}{c}\text{Core programming}\\ \text{language}\end{array}}$$

5.2.1.2 Souring: dual of de-sugaring

In our case the primary input is mathematical natural language. This language needs to be extended for computer software to understand it. *Souring* is the process of rewriting the sour bits to produce a *sour document*, i.e. a document which is formal enough to be understood by computer software. The input document and the sour document do not belong to the same type of document. A sour document is not a proper natural language text because it has been reorganised according to its core mathematical knowledge content. In that sense it is no longer a natural language text but more a core CGa document with local natural languages formatting elements. As a consequence, syntax souring is not the opposite of syntax sugaring.

$$\boxed{\begin{array}{c}\text{Natural language}\\ + \\ \text{Core annotations}\\ + \\ \text{Syntactic sour bits}\end{array} \quad \xrightarrow{\mathit{souring}} \quad \text{Sour core language}}$$

The *duality* between syntax sugaring and syntax souring resides in the fact that both are methods to humanise the authoring of rigid languages. These two methods are not equivalent because sugaring has a programming language as input. Syntax sugaring is a technique to adapt rigid languages to human uses. De-sugaring corresponds to rigidifying natural language. Syntax souring indicates how to transform natural language into a rigid language (this natural language which is altered by the souring process).

5.2.1.3 Syntax souring and natural language grammars

It is important to notice that the syntax souring system is by no means a natural language grammar. Syntax souring and the TSa annotation language do not stand at the natural language level but at the level of the mathematical natural language semantic. Natural language grammars describe the syntactic structure of natural language. The structure obtained is then interpreted to provide a natural language semantic. Such semantics reside at the speech level and identify a role or a meaning to every token of the syntax. There have been several grammars that used similar constructs as the syntax souring transformation (we presented in this Section 5.2). Nevertheless they are natural language grammars and do not provide this separation between the original human-medium (natural language) and the software-medium (CGa core language). We do not enter in the details of computational linguistic nor of natural language processing but we mention here Transformational-Generative Grammar [Cho57] [Far05, Chapter 5] and Combinatory Categorial Grammar [Ste96].

The TSa annotation language is an authoring method which permits the direct identification of the semantical role of parts of a natural language text (independently of its natural language semantic). The syntax souring annotations facilitates the use of this method by accommodating some natural language morphisms. Our authoring method—joining both TSa grammatical annotation and syntax souring—would benefits from automatic recognition and natural language processing. We discuss a possible outlook in Section 7.1.2.

5.2.2 Denotational representation

We give here the denotational representation of the TSa annotation language which we introduced in Section 5.1 and which is formalised in Section 6.4.1. Table 5.4 is a summary of the denotational representation.

5.2.2.1 Document

Our starting point is the mathematician's text (as he wrote it on paper) which is composed by a mixture of natural language text and formulas formed by symbols. This primary input corresponds to $\mathcal{D}_{\mathcal{F}}$ (formed by $\mathcal{F}$ individuals) in the abstract syntax of Section 6.4.1. We add to this primary input, grammatical and souring annotations that wrap portions of text. We already saw in Section 5.1 how we represent grammatical annotations. In this section we explain how we represent the souring annotations discussed in Section 5.2.1. We denote by T a portion of text which may include formulas, grammatical annotations and souring annotations. We denote by A an arbitrary annotation.

5.2.2.2 Grammatical annotations

A grammatical annotation is an instance of one of the grammatical categories **term**, **set**, **noun**, **adjective**, **statement**, **declaration**, **definition**, **context**, or **step** (see Table 5.1). Each instance of a grammatical annotation may get an attribute which corresponds to the grammatical annotation's interpretation as introduced in Section 5.1.2. We represent grammatical annotations by a box whose background colour—according to the colour coding of Table 5.1—informs the grammatical category and whose interpretation is printed on the upper left-hand side of the box in between the angle brackets `<` and `>`. Here is for instance the term a annotated with a **term**-box with `"a"` as interpretation: $\boxed{{}^{\texttt{<a>}}a}$. We use G, G', G_1, etc., to range over grammatical interpretations. Grammatical annotations correspond to $\mathcal{G}$ labels in the formal system presented in Section 6.4.1.

5.2.2.3 Souring annotations

Sour bits correspond to souring annotations. We denote them by a distinguishable border colour and a thicker box for the annotation they describe (i.e., $\boxed{{}^{\text{<list>}}a, b, c}$).

We define in Section 5.2.3 the following syntax souring annotations (which correspond to the elements souring labels $\mathcal{S}_u$ of Section 6.4.1): `position` i, `fold-right`, `fold-left`, `base`, `list`, `hook`, `loop`, `shared` and `map` (where i is a natural number).

5.2.2.4 Patterns

To describe the souring rules, we need to reason about the annotation boxes contained in a text. To do so, we add parameters to a text T to identify the text patterns that could be transformed. We use three different notations (as shown in Table 5.4) for these parametrised texts.

***In-order* notation** The arguments should appear in T in the same order as they appear in the pattern.

***Un-ordered* notation** The order of arguments is unimportant.

***Named* notation** The arguments are named. The names $n_1, \ldots, n_k$ are used as markers to determine the argument's location in the text.

The behaviour of a parametrised text is reflected in the de-formatting function (see Definition 14) and the compatibility property (see Definition 15) stated in Section 6.4.

5.2.3 Souring transformations

In this section we define the souring annotations and indicate how they can be used. We describe the result of each souring transformation where souring annotations are unfolded to obtain a text containing grammatical annotations, similarly to those of Section 5.1. Such a document could then be checked according to the CGa grammatical checker defined in Section 4.2.

We give in Section 6.4 a formal definition of a syntax souring operational system implementing these transformations. The set of syntax souring transformations we define in this section are summarised in Appendix A.3.

5.2.3.1 Reordering

<position i> When dealing with a natural language mathematical text, one regularly faces situations where two expressions holding similar knowledge are ordered differently. The `position` souring annotation is meant for reordering inner-annotations.

Grammatical annotation	
<x>x <Q>the set $\mathbb{Q}$	A chunk of text that belongs to a particular grammatical category is annotated with the corresponding grammatical annotation. The background colour informs the grammatical category (i.e., x is a term and could therefore be denoted <x>x), see Table 5.1. Grammatical annotations correspond to grammatical labels $\mathcal{G}$ of Section 6.4.1.

Souring annotation	
$\boxed{{}^{<\text{list}>}x, y, z}$	Souring annotation boxes are printed with a thicker border than other annotation boxes for differentiation. They correspond to souring labels $\mathcal{S}_u$ of Section 6.4.1.

Patterns	
In-order notation	$T(\boxed{{}^{<A_1>}T_1}, \dots, \boxed{{}^{<A_k>}T_k})$ or $T\begin{pmatrix}\boxed{{}^{<A_1>}T_1}\\ \vdots \\ \boxed{{}^{<A_k>}T_k}\end{pmatrix}$
Un-ordered notation	$T\left[\ \boxed{{}^{<A_1>}T_1}, \dots, \boxed{{}^{<A_k>}T_k}\ \right]$ or $T\left[\begin{array}{l}\boxed{{}^{<A_1>}T_1}\\ \vdots \\ \boxed{{}^{<A_k>}T_k}\end{array}\right]$
Named notation	$T\left[\ n_1 : \boxed{{}^{<A_1>}T_1}, \dots, n_k : \boxed{{}^{<A_k>}T_k}\ \right]$ or $T\left[\begin{array}{l}n_1 : \boxed{{}^{<A_1>}T_1}\\ \vdots \\ n_k : \boxed{{}^{<A_k>}T_k}\end{array}\right]$
$\boxed{{}^{<A_1>}T_1}, \dots, \boxed{{}^{<A_k>}T_k}$ being the arguments for T.	

Texts and symbols: **Formatting**
As we focus mainly on grammatical and souring nodes we omit, in this representation, the formatting elements (texts and symbols). In our examples, we represent these formating elements as they would be rendered by typesetting software. These formating elements correspond to formatting labels $\mathcal{F}$ of Section 6.4.1.

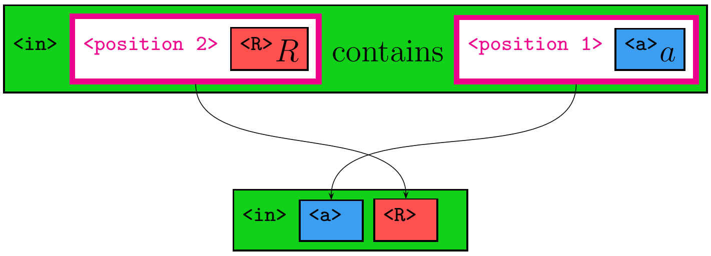
Table 5.4: TSa annotation representations summary

Each *direct* sub-expression gets a position number as attribute to indicate its position in the interpretation. The souring rewriting function reorders the elements according to their position indices. The reordering transformation corresponds to $\rightarrow_{pos}$ of Section 6.4.2.1.6.

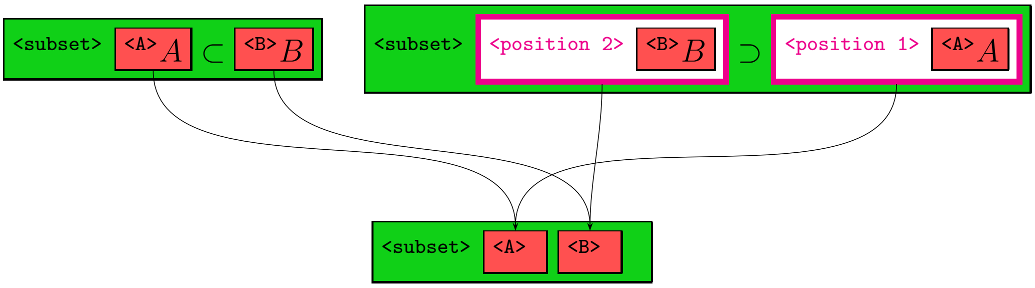
Definition 1 (Reordering transformation)

$$T\left[\begin{array}{l}\boxed{\textit{<position 1>}T_1}\\ \vdots \\ \boxed{\textit{<position n>}T_n}\end{array}\right] \xrightarrow{souring} T\left(T_1,\ldots,T_n\right)$$

Example 16 (Reordering) *Considering the expression "a in R", one can easily imagine the author using the equivalent expression "R contains a". The expressions "a in R" and "R contains a" should both be interpreted as* `in(a,R)` *if* `in` *is the set membership relation. To indicate in the second expression that the order of the argument is not the "reading" order, we annotate R and a with* `position 2` *and* `position 1`*, respectively.*



Example 17 (Mirror symbols) *It is common for binary symbols like $\subset$ to have a mirror twin like $\supset$. The* `position` *souring annotation usefully gives the same interpretation to twin symbols. The formulas "$A \subset B$" and "B contains A" are both interpreted as* `subset(A,B)`.



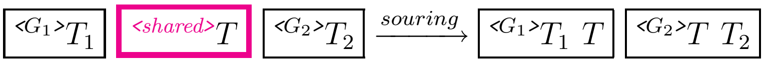
5.2.3.2 Sharing and chaining

<shared> <hook> <loop> Mathematicians have the habit of aggregating equations which follow one another. This creates reading difficulties for novices yet

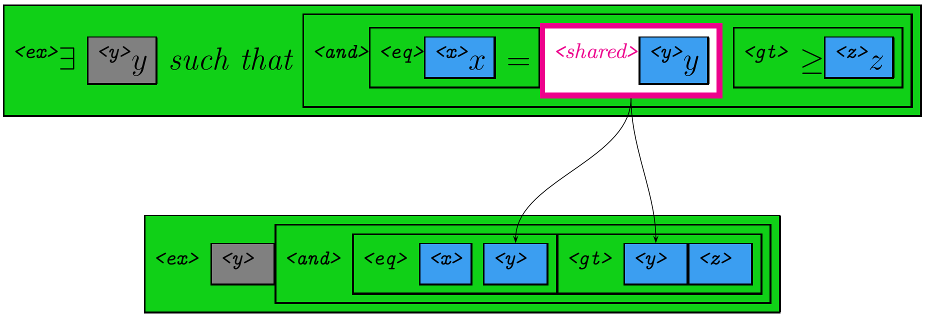
contributes to the aesthetic of mathematical writing. The `shared` and `hook`/`loop` souring annotations are solutions which elucidate such expressions.

5.2.3.2.1 Sharing The `shared` annotation indicates that an expression is to be used by both its preceding and following expressions. The shared expression is inlined at the end of the preceding expression and at the beginning of the following one. This transformation corresponds to $\twoheadrightarrow_{share}$ of Section 6.4.2.1.2.

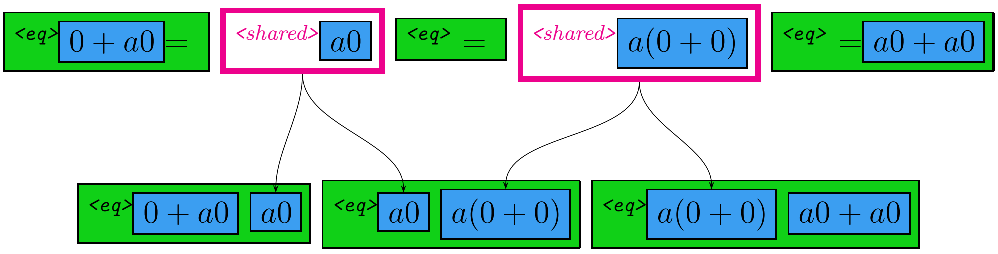
Definition 2 (Sharing transformation)



Example 18 (Simple sharing) *The formula "∃y such that $x = y > z$" could be annotated as follows and therefore be interpreted as* *`ex(y:term, and(eq(x,y),gt(y,z)))`.*



Example 19 (Sharing) *Let us take an example form [Gal02]'s chapter on ring theory. The following sentence is made easier to computerise by the use of sharing. The multiple equation "0+a0 = a0 = a(0+0) = a0+a0" requires the use of two* `shared` *annotations. We can see that $a0$ and $a(0+0)$ are shared by two equations each. We annotate them as being shared to obtain an unfolded result equivalent to "0+a0 = $a0$, a0 = a(0+0), a(0+0) = a0+a0".*



The full interpretation of this expression being:

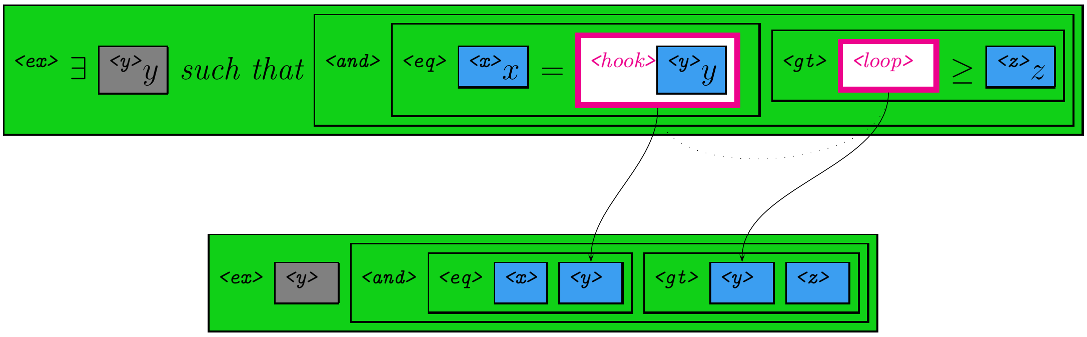
```
eq(plus(0,times(a,0)),times(a,0));
eq(times(a,0),times(a,plus(0,0)));
eq(times(a,plus(0,0)),plus(times(a,0),times(a,0)))
```

5.2.3.2.2 Chaining The tuple of souring annotations `hook`/`loop` indicates the expression contained in the hook should be repeated in the loop. We named this concept chaining because it permits the separation of two expressions which are effectively printed as one in a natural language text. Chaining provides results similar to sharing (any sharing could be expressed in terms of chaining), but is more expressive. This transformation corresponds to $\twoheadrightarrow_{chain}$ of Section 6.4.2.1.3.

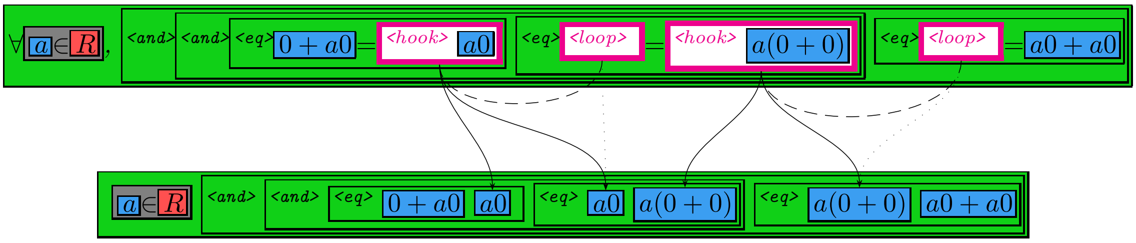
Definition 3 (Chaining transformation)

$$T\left(\begin{array}{c}\boxed{\texttt{<hook>}T'}\\ \boxed{\texttt{<loop>}}\end{array}\right)\xrightarrow{souring} T\left(\begin{array}{c}T'\\ T'\end{array}\right)$$

Example 20 (Simple chaining) *Let us see how the expression we used to illustrate a simple sharing annotation (see Example 18) could be annotated using chained sub-expressions.*


<ex> ∃ <y> y such that <and> <eq> <x> x = <hook> <y> y <gt> <loop> ≥ <z> z
<ex> <y> <and> <eq> <x> <y> <gt> <y> <z>

Example 21 (Chaining) *Let us see an example where a* `shared` *souring annotation could not have been used. If we consider the equation we used in the sharing example and decide to quantify this equation over a, we would obtain "∀a ∈ R, 0+a0 = a0 = a(0+0) = a0+a0" which is effectively a shortcut for "∀a ∈ R, 0+a0=a0 ∧ a0=a(0+0) ∧ a(0+0)=a0+a0". We can see that in this example the individual equations are combined using two binary operators* ***and****, the combination of whose annotation boxes disallows the use of* `shared`.


∀ a ∈ R, <and> <and> <eq> 0 + a0 = <hook> a0 <eq> <loop> = <hook> a(0 + 0) <eq> <loop> = a0 + a0
a ∈ R <and> <and> <eq> 0 + a0 a0 <eq> a0 a(0 + 0) <eq> a(0 + 0) a0 + a0

The full interpretation of this expression being:

```
forall(a:R,
       and( and( eq(plus(0,times(a,0)),times(a,0)),
                 eq(times(a,0),times(a,plus(0,0))) ),
            eq(times(a,plus(0,0)),plus(times(a,0),times(a,0)))
           ))
```

5.2.3.3 List manipulations

$\boxed{\texttt{<fold-right>}\ \boxed{\texttt{<base>}}\ \boxed{\texttt{<list>}}}$ $\boxed{\texttt{<fold-left>}\ \boxed{\texttt{<base>}}\ \boxed{\texttt{<list>}}}$ $\boxed{\texttt{<map>}\ \boxed{\texttt{<list>}}}$ The list souring annotations indicate how lists of expressions have to be unfolded into. We define two list folding annotations, `fold-right` and `fold-left`, and a mapping annotation, `map`.

5.2.3.3.1 Folding The `fold-right` souring annotation defines a pattern which is repeated for each element of the list argument. For each repeated pattern, the `list` inner annotation is replaced by one element of the list and the `base` inner annotation is replaced by the pattern with the next element of the list. A major use of the `fold-right` souring annotation is to handle quantification over multiple variables. `fold-left` works similarly but starts with the last element of the list. These transformations correspond to $\rightarrow_{fold}$ of Section 6.4.2.1.5.

Definition 4 (Right folding transformation)

$$\boxed{\texttt{<fold-right>}\ T_f \begin{bmatrix} b : \boxed{\texttt{<base>}\ T_b} \\ l : \boxed{\texttt{<list>}\ T_1 \ldots T_k} \end{bmatrix}} \xrightarrow{\textit{souring}}$$

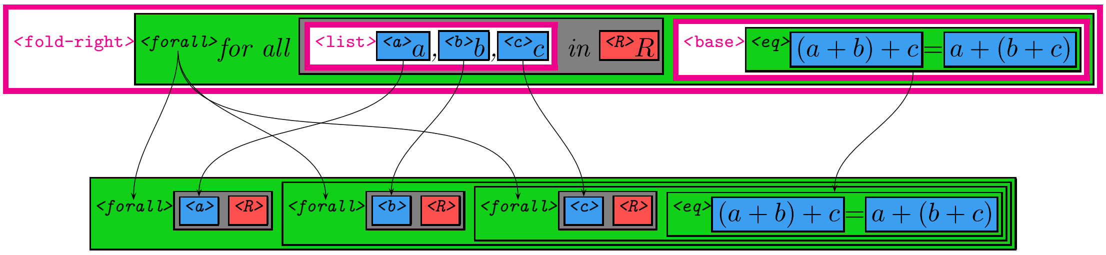
$$T_f \begin{bmatrix} b : T_f \begin{bmatrix} b : T_f \left[\cdots T_f \begin{bmatrix} b : T_b \\ l : T_k \end{bmatrix} \cdots \right] \\ l : T_2 \end{bmatrix} \\ l : T_1 \end{bmatrix}$$

Definition 5 (Left folding transformation)

$$\boxed{\texttt{<fold-left>}\ T_f \begin{bmatrix} b: \boxed{\texttt{<base>}\ T_b} \\ l: \boxed{\texttt{<list>}\ T_1\ldots T_k}\end{bmatrix}} \xrightarrow{souring}$$

$$T_f\begin{bmatrix} b: T_f\begin{bmatrix} b:T_f\begin{bmatrix}\cdots T_f\begin{bmatrix} b:T_b\\ l:T_1\end{bmatrix}\cdots\end{bmatrix} \\ l:T_{k-1}\end{bmatrix} \\ l:T_k\end{bmatrix}$$

Example 22 (Folding) *Considering the sentence "for all a, b, c in R [...] (a + b) + c = a + (b + c)", we would like to use one single* **`forall`** *instance for each variable a, b and c. We simply annotate the list of variables as such and the base equation as* `base` *and the souring unfolding creates a fully expanded interpretation on our behalf.*


<fold-right>
<forall> for all
<list> <a> a, <b> b, <c> c
in <R> R
<base> <eq> (a + b) + c = a + (b + c)
<forall> <a> <R>
<forall> <b> <R>
<forall> <c> <R>
<eq> (a + b) + c = a + (b + c)

The full interpretation of this expression being:

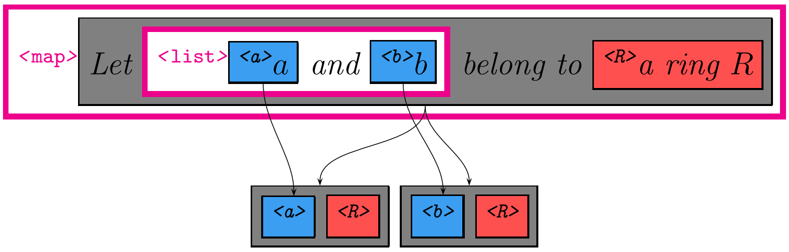
```
forall(a:R,
         forall(b:R,
                  forall(c:R,
                           eq(plus(plus(a,b),c),
                               plus(a,plus(b,c))) ) ) )
```

5.2.3.3.2 Mapping The `map` souring annotation also defines a pattern but with only one argument being `list`. This pattern is also repeated for each element of the list. The resulting expression is a sequence. Similarly to folding, this souring annotation is useful for declarations, definitions or statements over several arguments. It corresponds to $\rightarrow_{map}$ defined in Section 6.4.2.1.4.

Definition 6 (Mapping transformation)

$$\boxed{\texttt{<map>}\ T_f\left(\boxed{\texttt{<list>}\ T_1\ldots T_n}\right)} \xrightarrow{souring} T_f(T_1)\ldots T_f(T_n)$$

Example 23 (Mapping) *In the case of the sentence "Let a and b belong to a ring R" taken from our supplement example, the variables a and b are declared simultaneously.*



5.3 Authoring Experience

In this section we illustrate the use of MathLang's CGa-TSa on concrete examples taken from the mathematical literature. All examples presented here were input using the MathLang-TSa plugin for TeXmacs. The customised views (i.e., without annotations, with annotations and with printed interpretation) are instantaneously accessible via the plugin menu. See Appendix C for the plugin's documentation. See Section 6.3.1 for a presentation of the plugin.

We start in Section 5.3.1 with examples from E. Landau's *Foundations of Analysis* [Lan51]. Section 5.3.2 continues with the annotations of examples taken from Euclid's *Elements* [Hea56]. We finish in Section 5.3.3 with another encoding example, namely the proof of irrationality of $\sqrt{2}$ used by F. Wiedijk in his comparison of theorem provers [Wie06].

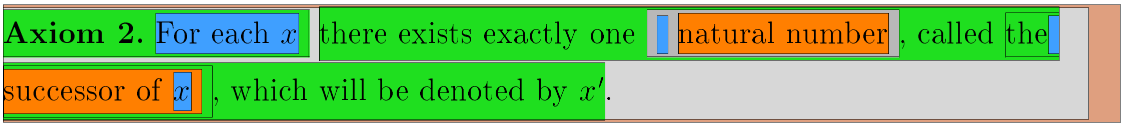
5.3.1 Examples from E. Landau's *Foundations of Analysis*

In 1930, E. Landau's *Foundations of Analysis* [Lan51] was published in its original German version [Lan30]. The whole book was encoded by L.S. van Benthem Jutting [vBJ77a] in Automath (see Section 2.1.1). We present here the TSa-CGa encoding of some passages of the first chapter of *Foundations of Analysis*.

First example We start with the definition of the Axiom 2 [Lan51, Chapter 1]. The original version is as follows.

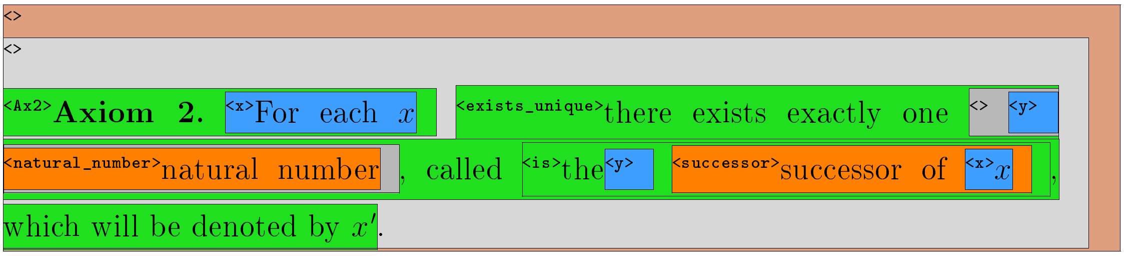
> **Axiom 2.** For each x there exists exactly one natural number , called the successor of x , which will be denoted by x'.
>
> *[Lan51, Chapter 1]*

This passage defines an axiom which could be interpreted as a CGa definition of a statement identifier (we name this identifier `Ax2`). This axiom is applied to any x (as the first phrase states), we decided therefore to parametrise the `Ax2` identifier by x (we assume this variable to be declared in the context of this example). This choice depends on the manner one wants to refer to the axiom later in the document. The rest of the definition is a statement that, for such x, there exists a unique natural number called the "successor of x". The TSa annotation version of this example is as follows.



Axiom 2. For each x there exists exactly one natural number, called the successor of x, which will be denoted by x'.

And the TSa annotation version with printed CGa interpretations is as follows.



<> <> <Ax2>**Axiom 2.** <x>For each x <exists_unique>there exists exactly one <> <y> <natural_number>natural number, called <is>the<y> <successor>successor of <x>x, which will be denoted by x'.

[Lan51, Chapter 1]

We note the parametrisation of `Ax2` and the use of an existential quantifier (`exists_unique`) to reason about the existence of a successor of x. The identifiers `exists_unique`, `natural_number` and `successor` were declared prior to the axiom's definition. The following CGa plain syntax code is the interpretation of the whole example.

```
{
  Ax2 (x) := exists_unique(y:natural_number,
                           is(y,successor(x)))
}
```

It is important to notice that this encoding is one possible encoding of this example. One can argue that the "Axiom" label commonly means that the statement is admitted throughout the document, a line `forall(x:natural_number,Ax2(x))` might therefore be added. Such precision is left to the author's decision and their logical correctness is to be analysed by future aspects (see Section 7.1). The concern of the CGa aspect is only to capture the grammatical meaning of the document's argumentation.

Second example Let us now consider the definition of "plus" taken from the same chapter. The definition is as follows.

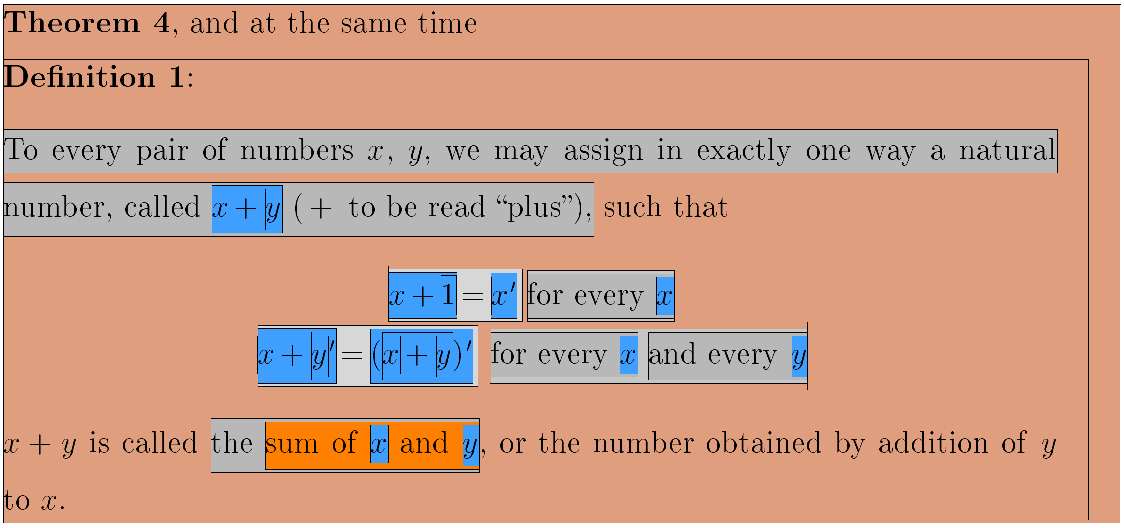
> **Theorem 4**, and at the same time
>
> **Definition 1**:
>
> To every pair of numbers x, y, we may assign in exactly one way a natural number, called $x + y$ ($+$ to be read "plus"), such that
>
> $$x + 1 = x' \text{ for every } x$$
>
> $$x + y' = (x + y)' \text{ for every } x \text{ and every } y$$
>
> $x + y$ is called the sum of x and y, or the number obtained by addition of y to x.
>
> *[Lan51, Chapter 1]*

We already used this example in Example 11 of Section 4.3.1.2 for the same purpose as here which is to illustrate the use of definition by cases. This definition is split in two cases: $x+1 = x'$ and $x+y' = (x+y)'$. We represent each one by a definition case in a step. A case of a definition by cases differs in TSa from a traditional definition by the presence of the symbol `#` as interpretation for the definition box.

Additionally, one can consider the first sentence of the definition as a preliminary declaration of the symbol "+". We therefore annotated this sentence as a declaration.

In this particular example, E. Landau uses two labels "Theorem" and "Definition" for the same paragraph. In our encoding of this example, we make use of the CGa labelling to identify both (i.e., labels `Th4` and `Def1` in the following version with printed interpretations of the example).

The TSa annotation version of this example is as follows.



Theorem 4, and at the same time

Definition 1:

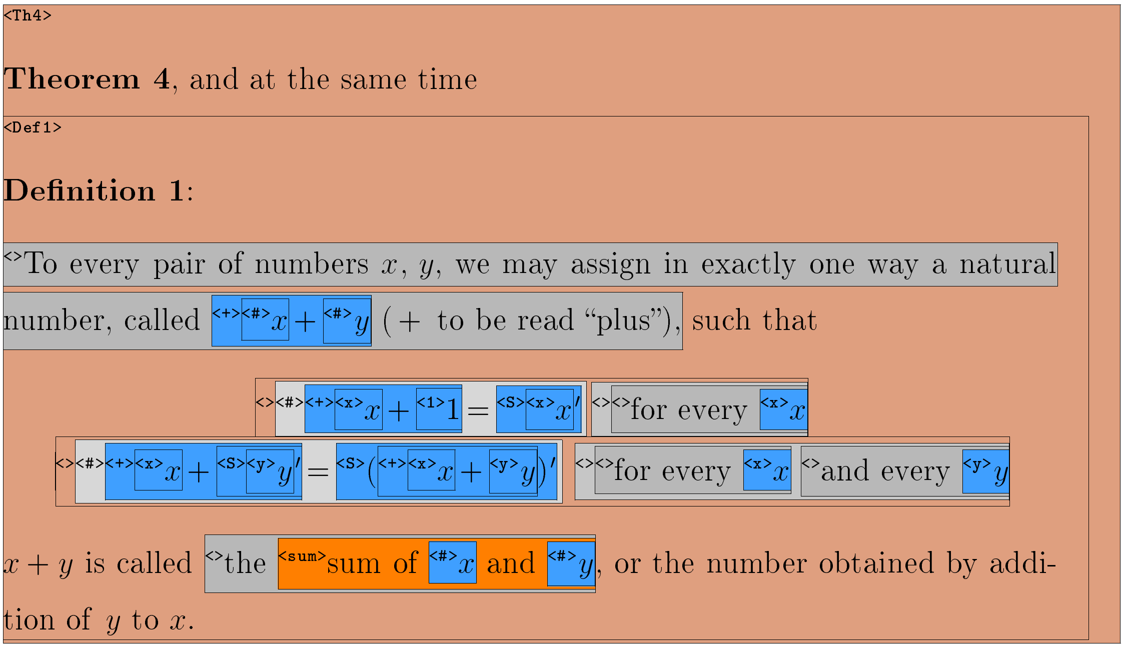
To every pair of numbers x, y, we may assign in exactly one way a natural number, called $x + y$ ($+$ to be read "plus"), such that

$$x + 1 = x' \text{ for every } x$$

$$x + y' = (x + y)' \text{ for every } x \text{ and every } y$$

$x + y$ is called the sum of x and y, or the number obtained by addition of y to x.

The following is the annotation version with printed interpretations.



<Th4>

Theorem 4, and at the same time

<Def1>

Definition 1:

<>To every pair of numbers x, y, we may assign in exactly one way a natural number, called <+><#>x + <#>y ($+$ to be read "plus"), such that

<><#><+><x>x + <1>1 = <S><x>x' <><>for every <x>x

<><#><+><x>x + <S><y>y' = <S>(<+><x>x + <y>y)$'$ <><>for every <x>x <>and every <y>y

$x + y$ is called <>the <sum>sum of <#>x and <#>y, or the number obtained by addition of y to x.

Third example We consider here a larger example with the definition and proof of Theorem 29 taken from the first chapter of E. Landau's *Foundations of Analysis*. The Theorem 29 defines the commutative law of multiplication. The original version and the TSa annotation version of this example are as follows.

Theorem 29 (Commutative Law of Multiplication):

$$x\,y = y\,x\ .$$

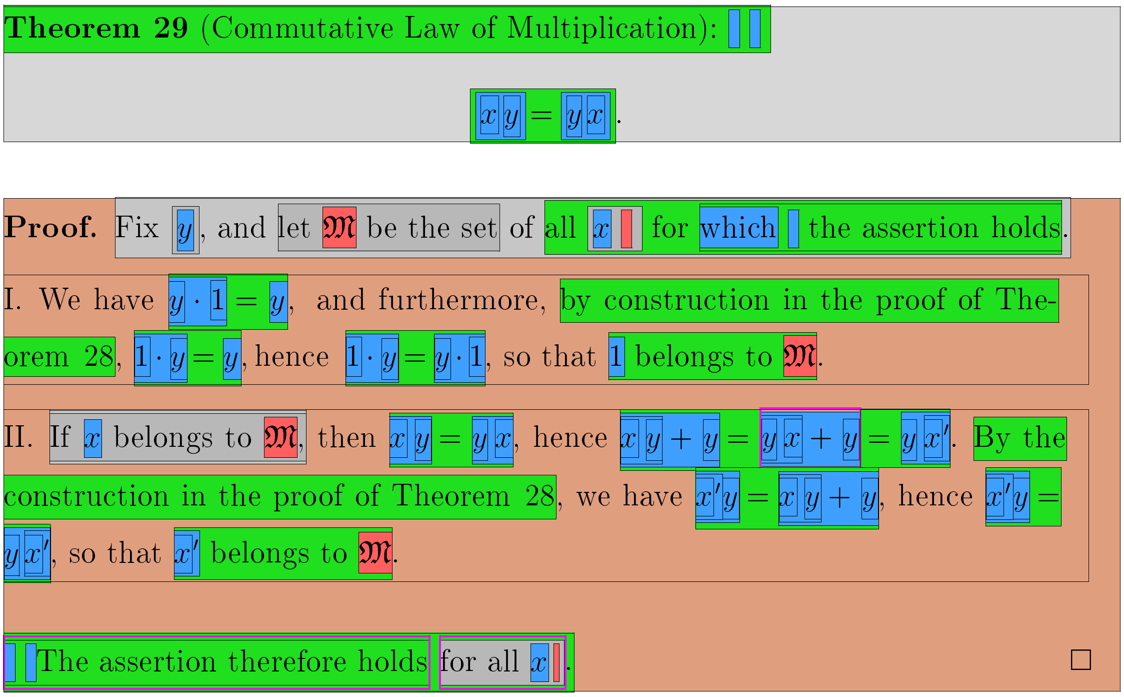
Proof. Fix y, and let $\mathfrak{M}$ be the set of all x for which the assertion holds.

I. We have $y \cdot 1 = y$, and furthermore, by construction in the proof of Theorem 28, $1 \cdot y = y$, hence $1 \cdot y = y \cdot 1$, so that 1 belongs to $\mathfrak{M}$.

II. If x belongs to $\mathfrak{M}$, then $x\,y = y\,x$, hence $x\,y + y = y\,x + y = y\,x'$. By the construction in the proof of Theorem 28, we have $x'y = x\,y + y$, hence $x'y = y\,x'$, so that x' belongs to $\mathfrak{M}$.

The assertion therefore holds for all x . □

[Lan51, Chapter 1]



Theorem 29 (Commutative Law of Multiplication):

$$x\,y = y\,x.$$

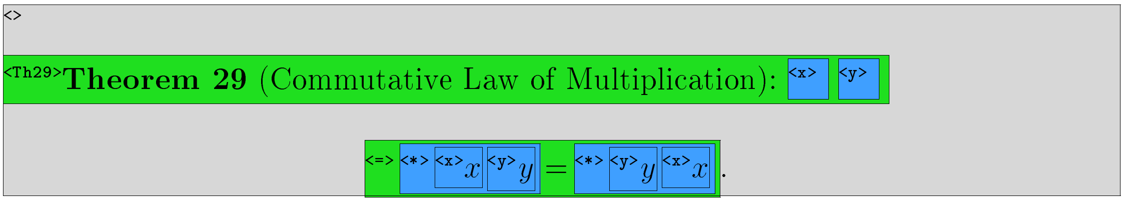
Proof. Fix y, and let $\mathfrak{M}$ be the set of all x for which the assertion holds.

I. We have $y \cdot 1 = y$, and furthermore, by construction in the proof of Theorem 28, $1 \cdot y = y$, hence $1 \cdot y = y \cdot 1$, so that 1 belongs to $\mathfrak{M}$.

II. If x belongs to $\mathfrak{M}$, then $x\,y = y\,x$, hence $x\,y + y = y\,x + y = y\,x'$. By the construction in the proof of Theorem 28, we have $x'y = x\,y + y$, hence $x'y = y\,x'$, so that x' belongs to $\mathfrak{M}$.

The assertion therefore holds for all x. □

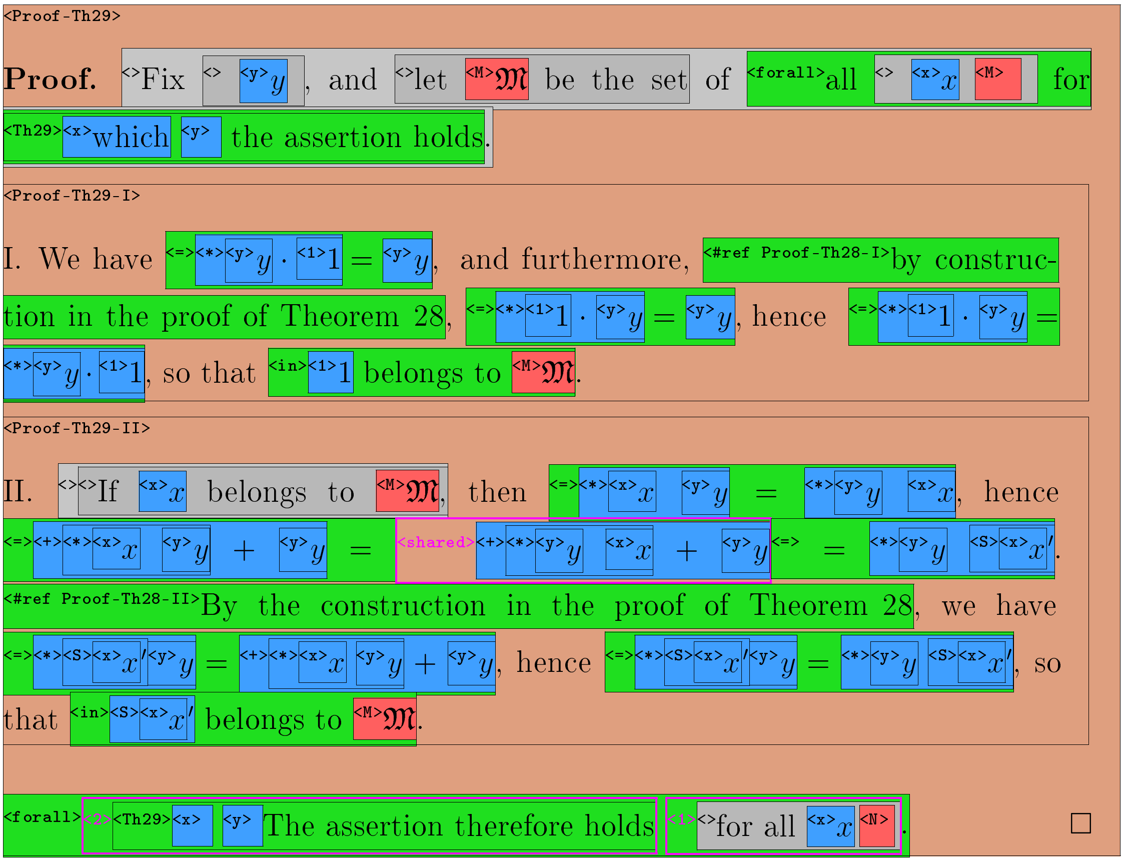
The following is the annotation version with printed interpretations.



<>

<Th29>**Theorem 29** (Commutative Law of Multiplication): <x> <y>

<=><*><x>x<y>y = <*><y>y<x>x.



<Proof-Th29>

Proof. <>Fix <> <y>y, and <>let <M>$\mathfrak{M}$ be the set of <forall>all <> <x>x <M> for <Th29><x>which <y> the assertion holds.

<Proof-Th29-I>

I. We have <=><*><y>y · <1>1 = <y>y, and furthermore, <#ref Proof-Th28-I>by construction in the proof of Theorem 28, <=><*><1>1 · <y>y = <y>y, hence <=><*><1>1 · <y>y = <*><y>y · <1>1, so that <in><1>1 belongs to <M>$\mathfrak{M}$.

<Proof-Th29-II>

II. <><>If <x>x belongs to <M>$\mathfrak{M}$, then <=><*><x>x<y>y = <*><y>y<x>x, hence <=><+><*><x>x<y>y + <y>y = <shared><+><*><y>y<x>x + <y>y<=> = <*><y>y<S><x>x'. <#ref Proof-Th28-II>By the construction in the proof of Theorem 28, we have <=><*><S><x>x'<y>y = <+><*><x>x<y>y + <y>y, hence <=><*><S><x>x'<y>y = <*><y>y<S><x>x', so that <in><S><x>x' belongs to <M>$\mathfrak{M}$.

<forall><2><Th29><x> <y>The assertion therefore holds <1><>for all <x>x<N>. □

We mention four particular exposures of this encoding.

1. Our annotation of Theorem 29's definition uses parameters similarly to the annotation of Axiom 2 in our previous example.

2. In this example, E. Landau uses two notations representing multiplication: a notation without any symbol to mark the multiplication (e.g., xy) and a dot "·" notation (e.g., $x \cdot y$). In the annotated example, each use of multiplication is consistently associated with a unique interpretation symbol "∗".

3. The proof of the Theorem 29 as written by E. Landau contains an aggregated equation: $xy + y = yx + y = yx'$. We use the sharing souring annotation we presented in Section 5.2.3.2.1 to annotate this equation. As a result the

interpretation of this equation duplicates the middle element. The result of the de-souring of this equation is equivalent to having both $xy + y = yx + y$ and $yx + y = yx'$.

4. Finally, E. Landau refers to the construction of the proof of Theorem 28 because his manner to prove Theorem 29 is identical to the manner he already proved Theorem 28. We indicate this relation by a reference to the *step*-labels `Proof-Th28-I` and `Proof-Th28-II` respectively. These labels were attributed to the sub-parts of the proof of Theorem 28 in the same fashion as the `Proof-Th29-I` and `Proof-Th29-II` labelling here.

Former encoding During our first year of PhD studies, we translated the whole first chapter of [Lan51] into MWTT (see Section 3.4.1). The MWTT encoding of the whole first chapter of E. Landau's *Foundations of Analysis* [Lan51] is presented in an appendix [2] to [KMW04b].

5.3.2 Examples from *The 13 Books of Euclid's Elements*

In this section we give some examples taken from the first book of Euclid's *Elements* [Hea56]. In contrast with the examples from Section 5.3.1, these ones are written in a less formal style. We have used these examples in Section 4.3 but we then only gave their CGa's plain syntax version.

First example We present here the TSa-CGa encoding of the six first definitions of the first book of Euclid's *Elements*. For this purpose, we use the object-oriented nature of CGa language (see Section 4.3.2.1). The definitions are as follows.

[2] Available at `http://www.macs.hw.ac.uk/~mm20/papers/Kamareddine+Maarek+Wells:mkm_symposium-entcs-appendix-2004.ps.gz`, last visited 2007–04–22.

1. A **point** is that which has no part.

2. A **line** is breadthless length.

3. The extremities of a line are points.

4. A **straight line** is a line which lies evenly with the points on itself.

5. A **surface** is that which has length and breadth only.

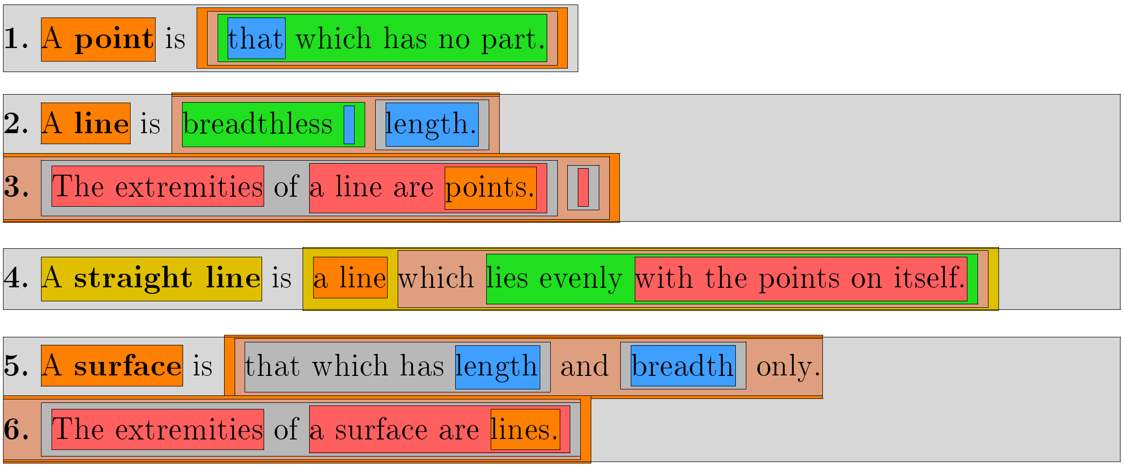
6. The extremities of a surface are lines.

[Hea56, Book I]

Among those six definitions, Euclid defines three kinds of objects: the points, the lines and the surfaces. The three other so-called definitions introduce specific features of points and surfaces for the third and sixth definitions respectively, and they introduce a specific kind (or sub-kind) of lines (fourth definition). We interpret and annotate these definitions as follows:

- The kinds of objects are CGa nouns,
- The specific features are CGa characters,
- And the sub-kinds are CGa adjectives.

The TSa annotation version of this example is as follows.



1. A **point** is that which has no part.

2. A **line** is breadthless length.

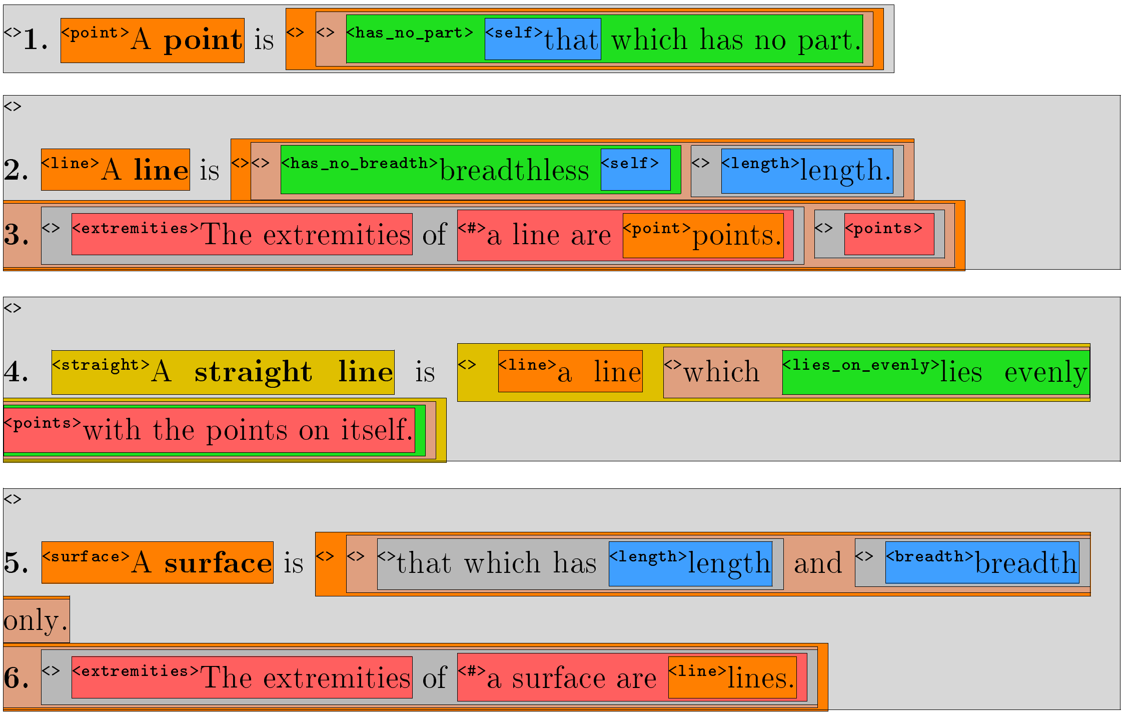
3. The extremities of a line are points.

4. A **straight line** is a line which lies evenly with the points on itself.

5. A **surface** is that which has length and breadth only.

6. The extremities of a surface are lines.

And the TSa annotation version with printed CGa interpretations is as follows.



<>**1.** <point>A **point** is <> <> <has_no_part> <self>that which has no part.

<>

2. <line>A **line** is <><> <has_no_breadth>breadthless <self> <> <length>length.

3. <> <extremities>The extremities of <#>a line are <point>points. <> <points>

<>

4. <straight>A **straight line** is <> <line>a line <>which <lies_on_evenly>lies evenly <points>with the points on itself.

<>

5. <surface>A **surface** is <> <> <>that which has <length>length and <> <breadth>breadth only.

6. <> <extremities>The extremities of <#>a surface are <line>lines.

In these annotation versions we see that the second and third definitions and the fifth and sixth are merged two-by-two as they define the “line” and “surface” nouns respectively. In both cases we use the noun description construction (a step annotation box inside a noun annotation box which corresponds to the CGa's abstract syntax `Noun` construction, see Table 5.2). Alternatively, we could use a CGa sub refinement for the definitions which add new characters (third and sixth definitions).

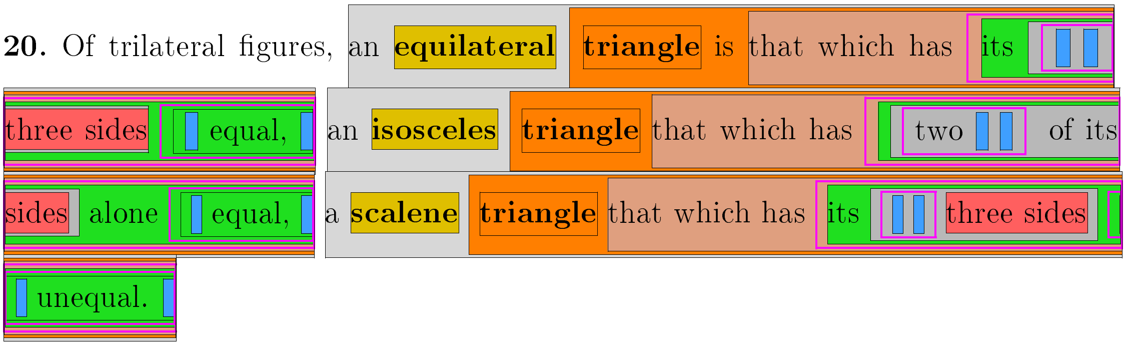
Summarising, the first definition states the absence of part[3] for “points”. The second and third definitions define three characters for “lines”: length, extremities and points. The fourth definition defines that the “straight” adjective characterise a line by stating that its points (character defined in the third definition) lies on evenly (we used specific statement identifier, which we named `lies_on_evenly`, to represent this characteristic). The fifth and sixth definitions defines the following characters for the “surface” noun: length, breadth and extremities.

Second example This example is concerned with the twentieth definition in Book I of Euclid's *Elements*. We illustrated the use of adjectives in CGa's plain syntax with this definition as example. The listing on page 105 is the CGa interpretation of the twentieth definition. The original version, the TSa annotation

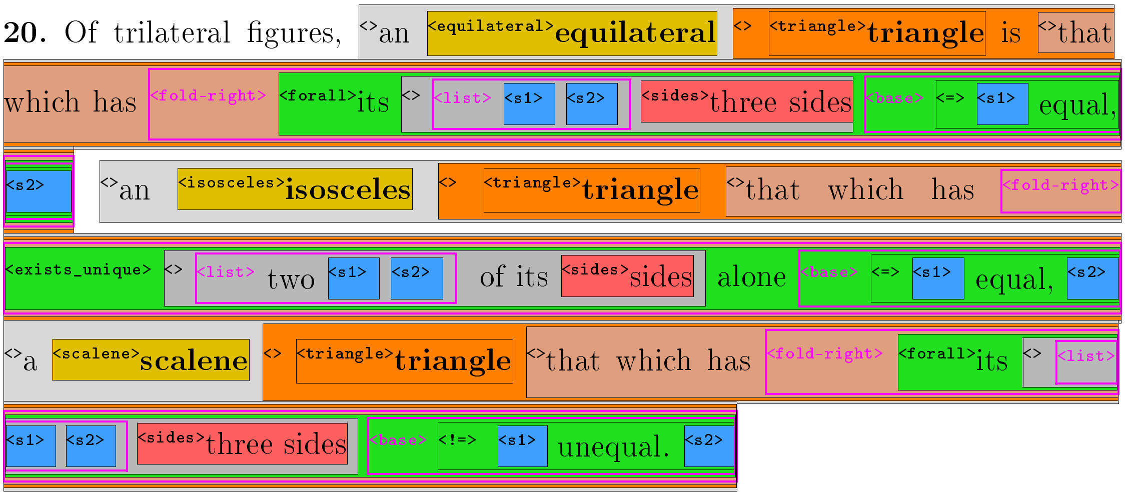
[3] The CGa language does not support to explicitly state the absence of a character; alternatively we used a statement expression `has_no_part`.

version and the TSa annotation version with printed interpretations are as follows.

> **20.** Of trilateral figures, an **equilateral** **triangle** is that which has its three sides equal, an **isosceles** **triangle** that which has two of its sides alone equal, a **scalene** **triangle** that which has its three sides unequal.
>
> *[Hea56, Book I]*



20. Of trilateral figures, an **equilateral** **triangle** is that which has its three sides equal, an **isosceles** **triangle** that which has two of its sides alone equal, a **scalene** **triangle** that which has its three sides unequal.



20. Of trilateral figures, <>an <equilateral>**equilateral** <> <triangle>**triangle** is <>that which has <fold-right> <forall>its <> <list> <s1> <s2> <sides>three sides <base> <=> <s1> equal, <s2> <>an <isosceles>**isosceles** <> <triangle>**triangle** <>that which has <fold-right> <exists_unique> <> <list> two <s1> <s2> of its <sides>sides alone <base> <=> <s1> equal, <s2> <>a <scalene>**scalene** <> <triangle>**triangle** <>that which has <fold-right> <forall>its <> <list> <s1> <s2> <sides>three sides <base> <!=> <s1> unequal. <s2>

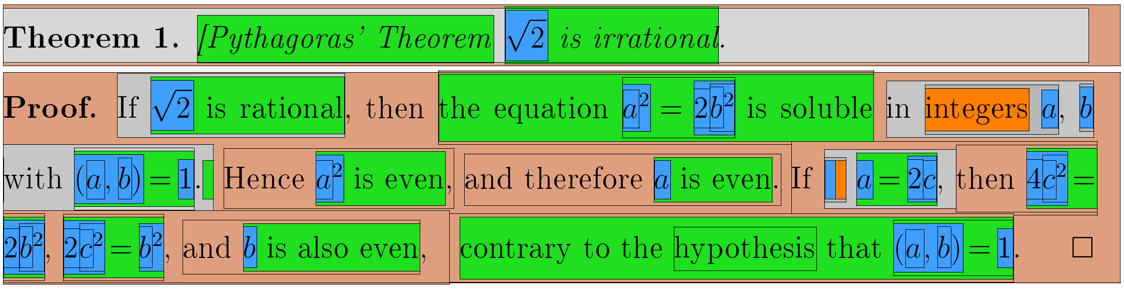
5.3.3 *Pythagoras' proof of irrationality of* $\sqrt{2}$

This section contains a TSa-CGa encoding of the Pythagoras' proof of irrationality of $\sqrt{2}$. The proof we use is due to G.H. Hardy and E.M. Wright [HW80]. It was used by F. Wiedijk in [Wie06] for a comparison of theorem provers. We used this example in [KMW04b] to illustrate our MWTT-based system to customise rendering of MWTT documents (see Section 6.3.2). The annotation of this example causes no particular difficulty. The original version, the TSa annotation version and the TSa annotation version with printed interpretations are as follows.

Theorem 1. *[Pythagoras' Theorem $\sqrt{2}$ is irrational.*

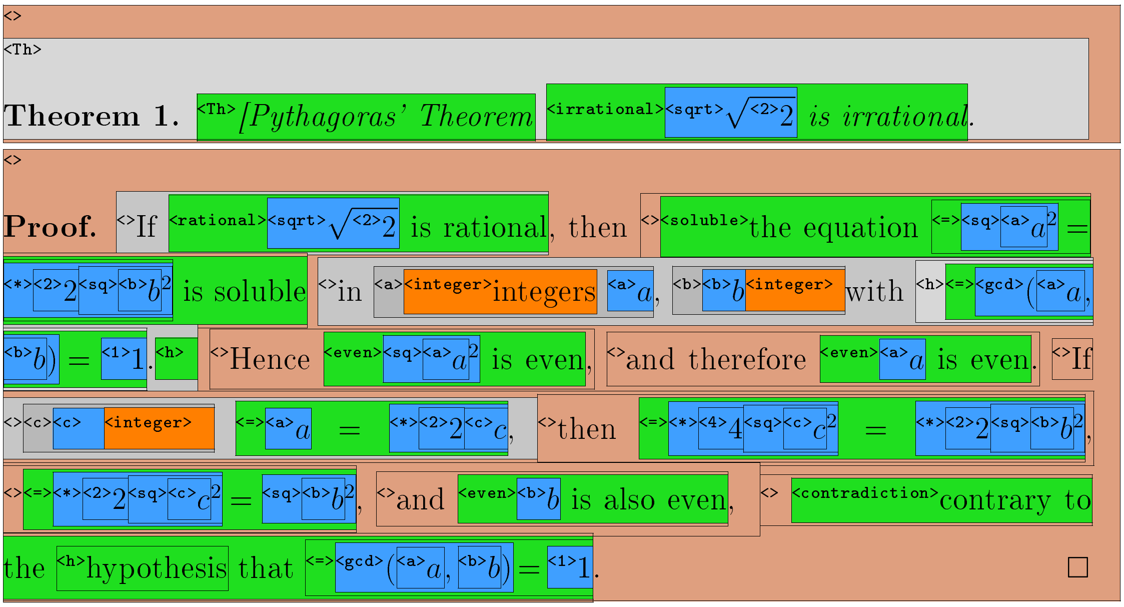
Proof. If $\sqrt{2}$ is rational, then the equation $a^2 = 2b^2$ is soluble in integers a, b with $(a,b)=1$. Hence a^2 is even, and therefore a is even. If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$, and b is also even, contrary to the hypothesis that $(a,b)=1$. □

[HW80]



Theorem 1. *[Pythagoras' Theorem $\sqrt{2}$ is irrational.*

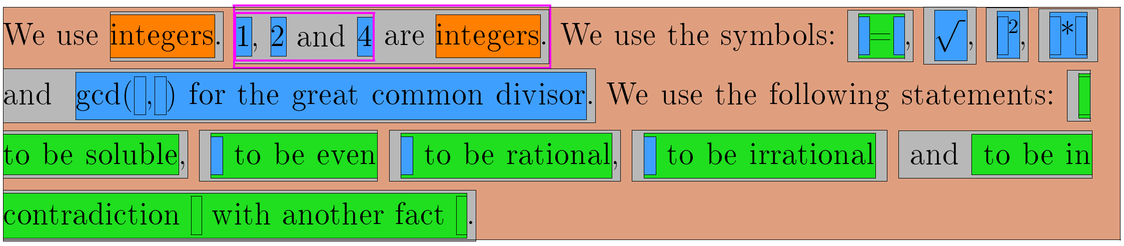
Proof. If $\sqrt{2}$ is rational, then the equation $a^2 = 2b^2$ is soluble in integers a, b with $(a,b)=1$. Hence a^2 is even, and therefore a is even. If $a = 2c$, then $4c^2 = 2b^2$, $2c^2 = b^2$, and b is also even, contrary to the hypothesis that $(a,b)=1$. □



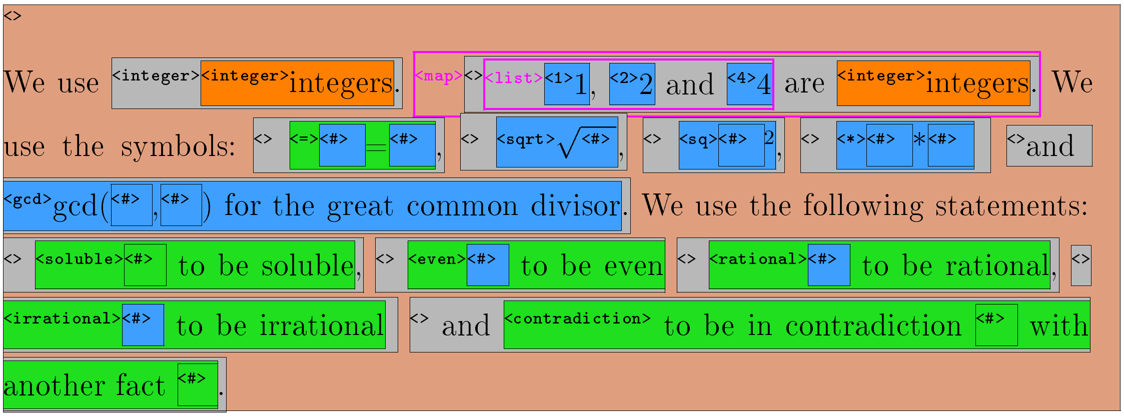
<>
<Th>
Theorem 1. <Th>*[Pythagoras' Theorem* <irrational><sqrt>$\sqrt{}$<2>2 *is irrational.*

<>
Proof. <>If <rational><sqrt>$\sqrt{}$<2>2 is rational, then <><soluble>the equation <=><sq><a>a^2 = <*><2>2<sq><b>b^2 is soluble <>in <a><integer>integers <a>a, <b><b>b<integer> with <h><=><gcd>(<a>a, <b>b) = <1>1. <h> <>Hence <even><sq><a>a^2 is even, <>and therefore <even><a>a is even. <>If <><c><c> <integer> <=><a>a = <*><2>2<c>c, <>then <=><*><4>4<sq><c>c^2 = <*><2>2<sq><b>b^2, <><=><*><2>2<sq><c>c^2 = <sq><b>b^2, <>and <even><b>b is also even, <> <contradiction>contrary to the <h>hypothesis that <=><gcd>(<a>a, <b>b) = <1>1. □

Let us use this example to illustrate another facet of TSa-CGa encodings with the definition of a context for the whole example. We sometime name this context a *preface*. A preface contains all the identifiers used but not declared nor defined in a document. in this practical example we need to declare "integer", "1", "2" and other symbols an notions. We show here the TSa annotation version and the TSa annotation version with printed interpretations of our example's preface. We added some explanatory text in this preface which is not required. A common preface is usually a succession of annotation boxes empty of texts.



We use integers. 1, 2 and 4 are integers. We use the symbols: =, √, 2, *, and gcd(,) for the great common divisor. We use the following statements: to be soluble, to be even, to be rational, to be irrational and to be in contradiction with another fact.



<> We use <integer><integer>integers. <map><><list><1>1, <2>2 and <4>4 are <integer>integers. We use the symbols: <> <=><#> = <#>, <> <sqrt>√<#>, <> <sq><#> 2, <> <*><#> * <#> <>and <gcd>gcd(<#>, <#>) for the great common divisor. We use the following statements: <> <soluble><#> to be soluble, <> <even><#> to be even <> <rational><#> to be rational, <> <irrational><#> to be irrational <> and <contradiction> to be in contradiction <#> with another fact <#>.

Conclusion

This chapter contains the presentation and definition of the MathLang-TSa system which provides a robust mechanism to interface natural language mathematical texts with the CGa language. This method includes a text annotation system and a set of syntax souring transformations. We illustrated the TSa authoring method on several concrete examples.

In Computational linguistics, transformational grammars [Far05, Chapter 5] provide a morphism method similar to souring. Nevertheless these grammars are a natural language grammars and do not provide the TSa separation between the original human-medium (natural language) and the software-medium (CGa language).

Chapter 6

Implementations, CGa and TSa

Since the start of the MathLang project, the focus has always been to combine theoretical specification and working implementation. This is part of our wish to base MathLang's design on encoding experiences (Section 3.4 describes how we managed so far to be driven by experience). Another focus has always been the extensive use of existing techniques and standards. The development was driven by the key characteristics we aimed for our system.

1. *Flexibility.* A computer language to be widely used by mathematicians should adapt to their needs and not the reverse. A language like LaTeX gives full liberty to the user with its macro system or its lower level language TeX. One of the main reasons theorem provers are not yet in the standard toolkit of any mathematician working on computer is certainly their rigidity. The users of such systems often need to adapt their works to the granularity level of the system and to the system's manner to organise knowledge. A system that wants to put its user ahead and let him free to use his editing habits would need to be fully flexible in its concrete encoding. Our choice was therefore an *XML-based implementation* and an attempt to make use of XML's flexibility and extensibility.

2. *Standalone.* The traditional programming process (editing – compiling – evaluating feedback – error fixing) is understood by computer scientists and adapted for most of the programmers' needs where the execution of the program is the final goal. But this editing process could be awkward for most mathematicians. Mathematical texts are meant to play different roles depending whether they are a book, a journal article, a school textbook, a formal proof, an engineering specification, a simple note or a student's as-

signment. The use one can make out of such a text and its content vary according to these roles. Some might lead to a complete proof likely to be checked by a computer and some might result in an implemented algorithm and therefore executed similarly to a computer program. Because the formalism level of a mathematical writing is not always known in advance, it is important to keep the editing process open. We consider as a requirement for the system to leave to the author the choice of the level of validation. To give this liberty to the author we have *embedded typing and analysis results* inside the document. By standalone document we mean the document as edited by the author plus explicit typing, plus environment information, plus errors found by the checker. Some external programs may make use of this embedded information for their computations. Therefore these embedded results and errors are to be independent from the program that created them. Here, standalone means a full reading of “edited” and “generated” content is possible without the need to run the program that created this “generated” content.

3. *Faithful to CML.* Centuries of mathematical practices have forged a tradition of mathematical writings made of standard symbols and representations, proof structure and abstractions. All the tacit rules that compose this CML make it a trustworthy communication medium for mathematics. Logicians, mathematicians and computer scientists have regularly wanted to elaborate and spread a more formal version or formal replacement to CML. These attempts were successful in the sense that they created new forms of mathematical works and that they helped to better understand mathematical deduction itself. Nevertheless they failed to overthrow CML as a universal language for mathematics. This failure is certainly due to the reduced space given to intuition in a formal language. Intuition is irreplaceable in mathematicians’ works, communications and teachings. This is why our input is CML and computerisation is facilitated with user-friendly authoring methods.

4. *Cost effective.* Any encoding of a document on computer requires some time and effort which could be reduced by good quality editing facilities. It is important for the feedback given by this automatic checking to be proportional to the encoding effort. For the design and implementation of our language we have focused on a clear distinction between different aspects of knowledge. This decomposition requires a systematic definition of the requirements and

the expected automated feedbacks for each aspect. Our language became *aspect-oriented* [KLM^{+}97] to facilitate this stratification.

In this chapter we go into the details of our implementation choices. All our choices were driven by these four aims. Section 6.1 is concerned with the definition of concrete syntax for CGa. Section 6.2 describes how the XML DOM-based implementation of the checker is populating the XML document with checking results. In Section 6.3 we describe the manner we handle natural language in our user interface. Finally, we present in Section 6.4 the operation system for the TSa syntax souring transformation of Section 5.2.

6.1 CGa's Concrete Syntax

CGa's concrete syntax follows the XML recommendations of the World Wide Web Consortium (W3C) [Worb]. The CGa XML syntax is the *standard* syntax for CGa. We have also developed a *programming-code* syntax for CGa which we refer to as the CGa's *plain syntax* (see Section 6.1.2) and an extension for the TEXMACS scientific editor which we refer to as the TSa-CGa macros for TEXMACS (see Section 6.1.3). These three syntaxes are not meant to be directly employed by the end-user. The CGa XML syntax is the internal software representation. The CGa's plain syntax is used for testing and theoretical discussions as it follows the abstract syntax of Section 4.1. The mathematician uses the TSa-CGa macros for TEXMACS via a TEXMACS GUI (see Section 2.4.1.4) and the TSa-CGa plugin for TEXMACS (see Section 6.3.1).

A CGa document embraces the actual author's input as well as automatic feedbacks generated by the checker. We explain how these automatic feedbacks are embedded in the XML document in Section 6.2. The reason why we decided not to have one plain syntax as a main user-interface was to avoid some notable disadvantages.

- *Expert-oriented.* A plain syntax as unique input and as standard document representation restrict the language users to a small set of initiates. Any person wanting to use the language would have been required to learn this unique plain syntax of the language in order to enjoy any advantages that the language features would offer.

- *Parser-dependent.* A type-inferred syntax facilitates the authoring but also makes any *intelligent* reading or use of the document tied up to the availability of an inference software to make explicit the meta-information (such as typing and environment). It is notable that standard programming (or formal proof editing) toolkits (Integrated Development Environment) either contain such an inference system or communicate with the expert system (compiler or theorem-prover). For example, Proof General [Asp00, WAL05, ALW06] which is a generic IDE for theorem provers, asks the interfaced theorem-provers for meta-information (such as typing, tactics available or proof obligations).

- *Uniqueness of human-oriented representation.* The choice of having one standard syntax for human input constrains any human user of the system to use this syntax (see Coq and Mizar, for example). Pre-processors, such as Camlp4 [dR03, Pou06], permit one to extend the flexibility of text-based input syntax. Nevertheless, text-based input are more difficult to interface with. A fixed software-oriented XML syntax could accommodate several human-oriented representations with the help of dedicated editors and GUI.

6.1.1 CGa XML syntax

As mentioned previously, CGa XML syntax encodes the author's input document as well as the checker's output such as typing, errors and environment. These contents are differentiated in the XML by different namespaces.

6.1.1.1 Core Grammatical

Namespace URI (usual prefix: `cga`)
`http://www.macs.hw.ac.uk/ultra/mathlang/grammatical-core`

This namespace provides XML elements to construct the steps and expressions of the CGa's abstract syntax described in section 4.1. Table 6.1 defines a transformation function from the CGa's abstract syntax to the CGa XML syntax. Table 6.2 contains an overview of the CGa elements of the CGa XML syntax.

6.1.1.2 Core Grammatical Meta

Namespace URI (usual prefix: `cga-meta`)
`http://www.macs.hw.ac.uk/ultra/mathlang/grammatical-core-meta`

This namespace groups the XML elements we use to enclose the meta-information automatically generated by CGa checker (see Section 6.2). This includes typing results, typing errors and environment information.

Category and identifier

$$\llbracket \mathtt{term}(e) \rrbracket^x = \begin{array}{l}\texttt{<cterm>}\\ \quad \llbracket e \rrbracket^x\end{array}$$

$$\llbracket \mathtt{set}(e) \rrbracket^x = \begin{array}{l}\texttt{<cset>}\\ \quad \llbracket e \rrbracket^x\end{array}$$

$$\llbracket \mathtt{noun}(e) \rrbracket^x = \begin{array}{l}\texttt{<cnoun>}\\ \quad \llbracket e \rrbracket^x\end{array}$$

$$\llbracket \mathtt{adj}(e_1, e_2) \rrbracket^x = \begin{array}{l}\texttt{<cadj>}\\ \quad \llbracket e_1 \rrbracket^x\\ \quad \llbracket e_2 \rrbracket^x\end{array}$$

$$\llbracket \mathtt{stat} \rrbracket^x = \texttt{<cstat>}$$

$$\llbracket \mathtt{dec}(c) \rrbracket^x = \begin{array}{l}\texttt{<cdec>}\\ \quad \llbracket c \rrbracket^x\end{array}$$

$$\llbracket v \rrbracket^x = \begin{array}{l}\texttt{<cvar>}\\ \quad \texttt{<name>}\ v\end{array} \qquad {}^{1}$$

$$\llbracket i \rrbracket^x = \begin{array}{l}\texttt{<ident>}\\ \quad \texttt{<name>}\ i\end{array} \qquad {}^{1}$$

$$\llbracket e.i \rrbracket^x = \begin{array}{l}\texttt{<holder>}\\ \quad e\\ \llbracket i \rrbracket^x\end{array}$$

Step and phrase

$$\llbracket s_p \rrbracket^x = \begin{array}{l}\texttt{<step>}\\ \quad \texttt{<phrase>}\\ \quad\quad \llbracket p \rrbracket^x\end{array} \qquad {}^{2}$$

$$\llbracket s_1 \ \rhd \ s_2 \rrbracket^x = \begin{array}{l}\texttt{<step>}\\ \quad \texttt{<local>}\\ \quad\quad \llbracket s_1 \rrbracket^x\\ \quad \llbracket s_2 \rrbracket^x\end{array}$$

$$\llbracket \{s_1; \ldots; s_n\} \rrbracket^x = \begin{array}{l}\texttt{<step>}\\ \quad \llbracket s_1 \rrbracket^x\\ \quad \vdots\\ \quad \llbracket s_n \rrbracket^x\end{array}$$

$$\llbracket \mathtt{label}\ l\ s \rrbracket^x = \begin{array}{l}\texttt{<label>}\\ \quad\quad \texttt{<name>}\ l\\ \quad \llbracket s \rrbracket^x\end{array} \qquad {}^{1}$$

$$\llbracket ci(i_1, \ldots, i_n) := e \rrbracket^x = \begin{array}{l}\texttt{<definition>}\\ \quad \llbracket ci \rrbracket^x\\ \quad \llbracket i_1 \rrbracket^x\\ \quad \vdots\\ \quad \llbracket i_n \rrbracket^x\\ \quad \llbracket e \rrbracket^x\end{array}$$

$$\llbracket ci(e_1, \ldots, e_n) := e \rrbracket^x = \begin{array}{l}\texttt{<definition>}\\ \quad \llbracket ci \rrbracket^x\\ \quad \llbracket e_1 \rrbracket^x\\ \quad \vdots\\ \quad \llbracket e_n \rrbracket^x\\ \quad \llbracket e \rrbracket^x\end{array}$$

$$\llbracket i \ll e \rrbracket^x = \begin{array}{l}\texttt{<sub>}\\ \quad \llbracket i \rrbracket^x\\ \quad \llbracket e \rrbracket^x\end{array}$$

Expression

$$\llbracket ci(e_1, \ldots, e_n) \rrbracket^x = \begin{array}{l}\texttt{<instance>}\\ \quad \llbracket ci \rrbracket^x\\ \quad \llbracket e_1 \rrbracket^x\\ \quad \vdots\\ \quad \llbracket e_n \rrbracket^x\end{array}$$

$$\llbracket i(ce_1, \ldots, ce_n) : ce \rrbracket^x = \begin{array}{l}\texttt{<declaration>}\\ \quad \llbracket i \rrbracket^x\\ \quad \llbracket ce_1 \rrbracket^x\\ \quad \vdots\\ \quad \llbracket ce_n \rrbracket^x\\ \quad \llbracket ce \rrbracket^x\end{array}$$

$$\llbracket \mathtt{Noun}\{s\} \rrbracket^x = \begin{array}{l}\texttt{<noun>}\\ \quad \llbracket s \rrbracket^x\end{array}$$

$$\llbracket \mathtt{Adj}(e)\{s\} \rrbracket^x = \begin{array}{l}\texttt{<adjective>}\\ \quad \llbracket e \rrbracket^x\\ \quad \llbracket s \rrbracket^x\end{array}$$

$$\llbracket e_1\ e_2 \rrbracket^x = \begin{array}{l}\texttt{<refinement>}\\ \quad \llbracket e_1 \rrbracket^x\\ \quad \llbracket e_2 \rrbracket^x\end{array}$$

$$\llbracket \mathtt{self} \rrbracket^x = \texttt{<self>}$$

$$\llbracket \mathtt{ref}\ l \rrbracket^x = \begin{array}{l}\texttt{<reference>}\\ \quad \texttt{<name>}\ l\end{array} \qquad {}^{1}$$

For readability, we show only the opening tag of each XML element; instead we use indentation to express nesting. In the $\llbracket e.i \rrbracket^x$ *case, it is important to notice that* `<holder>` and i are similarly indented and therefore are nested in the XML tree.

[1] The element `<name>` contains the row raw identifier, variable and label name respectively.
[2] The phrase p is the equivalent of the basic step s_p at the phrase level.

Table 6.1: CGa's abstract syntax denoted in terms of CGa's XML syntax. Denotation function $\llbracket\ \rrbracket^x$ transforming abstract syntax expressions into XML syntax expressions.

XML elements	XML children	Abstract syntax
`<mathlang>`	`<step>`	Whole document
Steps		
`<step>`	`<label>`?, (`<phrase>` | `<step>`* | `<local>`, `<step>`)	`label` *label step* *phrase* $\{\overrightarrow{step}\}$ *step* $\rhd$ *step*
`<label>`	`<name>`	
`<local>`	`<step>`	
Phrases		
`<phrase>`	(expression | `<definition>` | `<sub>`)	*phrase*
`<definition>`	(`<id>`, `<id>`*, expression | `<id>`, expression*, expression)	$cident(\overrightarrow{ident}) := exp$ $cident(\overrightarrow{exp}) := exp$
`<sub>`	`<id>`, expression	$ident \ll exp$
Expressions		
`<adjective>`	expression, `<step>`	`Adj`$(exp)\ \{step\}$
`<declaration>`	`<id>`, (category | expression)*, (category | expression)	*ident* $\overrightarrow{(category \mid exp)}$: $category \mid exp$
`<instance>`	`<holder>`?, `<id>`, expression*	$cident(\overrightarrow{exp})$
`<noun>`	`<step>`	`Noun` $\{step\}$
`<refinement>`	expression, expression	*exp exp*
`<self>`		`self`
`<reference>`	`<name>`	`ref` *label*
`<id>`	`<name>`	*ident*
Categories		
`<cterm>`	expression	`term`(exp)
`<cset>`	expression	`set`(exp)
`<cnoun>`	expression	`noun`(exp)
`<cadj>`	expression, expression	`adj`(exp, exp)
`<cstat>`		`stat`
`<cdec>`	category	`dec`$(category)$
`<cvar>`	`<name>`	*cvar*
`<name>`	Chain of characters	

expression = `<adjective>` | `<declaration>` | `<instance>` | `<noun>` | `<refinement>` | `<reference>` | `<self>`

category = `<cterm>` | `<cset>` | `<cnoun>` | `<cadj>` | `<cstat>` | `<cdec>` | `<cvar>`

Table 6.2: Elements of the CGa XML syntax

We give the possible children and the corresponding abstract syntax expression of each CGa XML element. The symbol "|" denote the alternative, "?" the optional occurrence and "*" the zero or more occurrence.

6.1.1.2.1 Typing results Typing results are presented in two kinds of positions in the XML document. Firstly, as child of each language construction, the output type of the construction is printed in an `<cga-meta:type>` element. Secondly, the type of each identifier is informed in an `<cga-meta:env-type>` element. See Section 6.1.1.2.3 for explanation on where this typing is positioned in relation to the identifier itself and to the typing environment.

Each `<cga-meta:type>` and `<cga-meta:env-type>` element contains one atomic type data and one schema type data, respectively. The representation with the CGa's abstract syntax types defined in Section 4.2.1.1, is described in Table 6.3.

CGa's terms, sets, nouns and adjectives are particularised by a mapping of character names and types (represented by $\mathcal{T}$ in section 4.2.1.1). A mapping is basically a noun description. In CGa's abstract syntax, mappings are expressed by means of noun and adjective descriptions (`Noun` and `Adj` constructions). As we saw in Section 4.1, these descriptions use a step as medium to define the list of characters that compose the mapping. Later on, these mappings can be combined to obtain new nouns and adjectives, or transformed terms and sets. The representation of these mappings in CGa XML syntax is related to the way the mapping has been obtained. Table 6.3 defines the `cga-meta` XML elements to build such mappings. A mapping is either:

- An *empty mapping* if it comes from an empty noun or adjective description.

```
{
  n := Noun;
  n' : noun;
}
```

  Here `n` and `n'` are defined by empty noun descriptions.

- A mapping *described by a step.*

```
{
  Noun { sides:set };
  Adj (figure) { card(sides)=3 }
}
```

- Obtained by a *refinement.*

```
trilateral figure
```

- A noun or adjective identifier which is the *comprehensive name* for a mapping.

```
{
  figure := Noun { sides:set };
  trilateral := Adj (figure) { card(sides)=3 };
}
```

Names of mappings are essential for human comprehension. Formally, a mapping is no more than a set of character names associated with their types. Only this abstract representation matters for the typing of an expression involving mappings, names themselves do not hold any formal information. Nevertheless when an error is detected, the user is entitled to get from the checker a message that can be easily understood. For example, if we define group and ring as two nouns. Group is defined with its set and operation as characters, and ring with its set and two operations as characters. For brevity we omit here the definition of the group and ring neutral elements and axioms.

```
{
  group := Noun { E : set;
                  {a:E; b:E} |> add (a,b) : E
                };
  ring  := Noun { E : set;
                  {a:E; b:E} |> add (a,b) : E;
                  {a:E; b:E} |> mul (a,b) : E
                }
}
```

Let us then declare a property `P` on rings and apply it wrongly on a group `G`.

```
{
  P (ring) : stat;
  G : group;
  P (G)
}
```

Here the type checker would notice the wrong argument's type for the instance of `P`. The identifier `P` expects an argument of type

$$\mathit{Term}(\ \{\ (\mathtt{E}, \mathit{Set}),\ (\mathtt{add}, (\mathit{Term}, \mathit{Term}) \rightarrow \mathit{Term}),\ (\mathtt{mul}, (\mathit{Term}, \mathit{Term}) \rightarrow \mathit{Term})\ \}\ )$$

but is provided with an expression of type

$$\mathit{Term}(\ \{\ (\mathtt{E}, \mathit{Set}),\ (\mathtt{add}, (\mathit{Term}, \mathit{Term}) \to \mathit{Term})\ \}\ ).$$

CGa checker makes use of the mappings' names to describe the error with more readability.

```
Error (e-1):
  Types mismatch for "P",
  (term[ring]):stat expected, not (term[group]):stat.
```

It is nevertheless possible to obtain from the XML elements the full list of characters and types for each noun.

6.1.1.2.2 Typing errors The errors detected by the checker are reported directly inside the document. See section 4.2 for a description of CGa's type system and section 6.2 for an explanation on the way this type system is implemented. CGa's type checker analyses the whole document. It works in a non-stop manner reporting all typing-errors found. The documents that do not comply to the XML recommendations are rejected by the checker. In the rest of this section, we mean typing-error when we use the word "error".

The errors found by the checker are listed as children of the root XML element of the document. Each error is described in one `<cga-meta:error>` with a unique `xml:id` attribute and a description in terms of `cga-meta` elements listed in Appendix B.1.

There are two peculiarities in the way errors are handled in our CGa implementation. Firstly the representation of errors and secondly the location of errors inside the document.

Error description Each error has a unique representation (see table 6.3). For example, if an identifier is used with arguments of the wrong types (according to the identifier's definition), then a new error is created. Its description takes into account the error's name (`Types_mismatch`) and the error's parameters (`xml:id` of the misused parameters). This makes the error description unique.

On the implementation side, this unique representation reduces the listing of errors. If similar errors occur several times they are identified as such and therefore a unique error description points at several locations where the typing rules failed to validate an expression.

	Atomic type
$\llbracket \mathit{Term}(m) \rrbracket^x =$	`<cga-meta:Term>` $\llbracket m \rrbracket^x$
$\llbracket \mathit{Set}(m) \rrbracket^x =$	`<cga-meta:Set>` $\llbracket m \rrbracket^x$
$\llbracket \mathit{Noun}(m) \rrbracket^x =$	`<cga-meta:Noun>` $\llbracket m \rrbracket^x$
$\llbracket \mathit{Adj}(m_1, m_2) \rrbracket^x =$	`<cga-meta:Adj>` $\llbracket m_1 \rrbracket^x$ $\llbracket m_2 \rrbracket^x$
$\llbracket \mathit{Stat} \rrbracket^x =$	`<cga-meta:Stat>`
$\llbracket v \rrbracket^x =$	`<cga-meta:Var>` v
$\llbracket \mathit{Dec}(t) \rrbracket^x =$	`<cga-meta:Dec>` $\llbracket t \rrbracket^x$
$\llbracket \mathit{Def}(t) \rrbracket^x =$	`<cga-meta:Def>` $\llbracket t \rrbracket^x$
$\llbracket \mathit{Sub}(t) \rrbracket^x =$	`<cga-meta:Sub>` $\llbracket t \rrbracket^x$
$\llbracket \mathit{Step} \rrbracket^x =$	`<cga-meta:Step>`
$\llbracket \mathit{Categ}(a) \rrbracket^x =$	`<cga-meta:Categ>` $\llbracket a \rrbracket^x$
	Type signature

$\llbracket (a_1, \ldots, a_n) \rightarrow a \rrbracket^x =$	`<cga-meta:Schema>`
	   `<cga-meta:In>`
	      $\llbracket a_1 \rrbracket^x$
	      $\vdots$
	      $\llbracket a_n \rrbracket^x$
	   `<cga-meta:Out>`
	      $\llbracket a \rrbracket^x$

	Type mapping
Empty mapping	`<cga-meta:Empty>`

Described by a step	`<cga-meta:Description>`
	   [`step's xml:id`]

Refinement of two mappings m_1 and m_2	`<cga-meta:Refinement>` $\llbracket m_1 \rrbracket^x$ $\llbracket m_2 \rrbracket^x$

Mapping m named n	`<cga-meta:Named>`
	   `<cga-meta:name>` n
	   $\llbracket m \rrbracket^x$

For readability, we show only the opening tag of each XML element; instead we use indentation to express nesting.

Table 6.3: CGa types and their corresponding XML elements. Denotation function $\llbracket\ \rrbracket^x$ transforming CGa types into XML syntax expressions.

For example, if the addition is misused several times in a file `err.mathlang.plain` containing the following lines:

```
    {
      x : term;
      0 : term;
      S : set;
      + (term, term) : term;
      = (term, term) : stat;

      = ( + (0,x), +(S,0) );
      = ( + (S,0), +(0,x) )
    }
```

then, the two typing mistakes of line 7 and 8 are pointed out as being a unique error `e-1`:

```
Error (e-1): Types mismatch for "+",
             (term,term):term expected, not (set,term):term.
  err.mathlang.plain: L5 C2-23
  err.mathlang.plain: L8 C15-21
  err.mathlang.plain: L9 C6-13
```

Error pointers The type checker identifies the wrongly formed expressions (resulting of a failure to apply the typing rules of Section 4.2). This does not necessarily mean that the authoring mistake occurred at the location of the typing failure. The system points at the locations involved in the error. This feature is a simplistic approach of type error slicing in typed programming languages [HW04]. In CGa, we opted to associate to each error all the different locations that could have played a role in the typing failure. Each position of error in the XML document is identified by an element `<cga-meta:error>` with an attribute `ref` referring to the error descriptor (via its `xml:id`).

For example, if we change slightly our previous example by fixing the misuse of the addition but also by introducing an error in the declaration of `=` we get the following.

```
    {
      x : term;
      0 : term;
      + (term, term) : term;
      = (stat, term) : stat;

      = ( + (0,x), +(x,0) );
      = ( + (x,0), +(0,x) )
    }
```

The checker finds an error related to the typing of = and points at the type declaration of = and at the instances of this identifier not complying with this type.

```
Error (e-1): Types mismatch for "=",
             (stat,term):stat expected, not (term,term):stat.
  err.mathlang.plain: L5 C2-23
  err.mathlang.plain: L7 C2-23
  err.mathlang.plain: L8 C2-23
```

6.1.1.2.3 Environment The typing environment is composed by the steps preceding the document's element to be checked. It is used in every typing rule of Section 4.2. The IDENT-* rules of Section 4.2.3.1.1 explicate how to retrieve the type of an identifier inside the typing environment. We decided to build this environment inside the XML tree to permit an access to this meta-information. This concrete environment consists of two kinds of `cga-meta` XML elements, the *environment modifiers* described in table 6.4 and the *environment referees* described in table 6.5.

Environment modifiers These are the actions that a local expression has on its immediate followers. This could be, for example, the introduction of a new identifier for declarations and definitions.

Environment referees These are pointers that one should follow to rebuild the environment at each position in the XML tree. The algorithms for rebuilding the environment are explained in detail in Section 6.2.2.

6.1.1.3 Location

Namespace URI (usual prefix: `loc`)
`http://www.macs.hw.ac.uk/ultra/mathlang/location`

CGa XML syntax is the standard representation for CGa documents. Since our implementation of CGa's type system is XML DOM based, the tree representation

`<env-identifier>`	To announce a new identifier. Attribute `xml:id` is the identifier's unique XML identifier (prefixed `i-` by the checker). Its attribute `name-ref` refers to the XML element holding the identifier's name.
`<env-instance>`	To announce the instance of an identifier. Its attribute `for` indicates the `xml:id` of the identifier.
`<env-status>`	To inform about a new status of an identifier. Its attribute `for` indicates the `xml:id` of the identifier. Its attribute `xml:id` is the status modifier's unique XML identifier (prefixed `st-` by the checker). If the status modifier brings a change from previous status, the attributes `extends` contains the `xml:id` of the previous status modifier. This element could have one or more `<env-holder>` as optional children.
`<env-holder>`	Possible child element for `<env-status>`. It is used when a status modifier concerns a character of a term. Its attribute `ref` points at the term identifier's `xml:id`. An element `<env-status>` could contain several `<env-holder>` if the identifier in question is the character of a character... of a term.
`<env-type>`	To inform about a new type of an identifier. Its attribute `for` indicates the `xml:id` of the identifier. Its attribute `xml:id` is the type modifier's unique identifier (prefixed `ty-` by the checker). If the type modifier brings a change from previous type, the attribute `extends` contains the `xml:id` of the previous type modifier.

Table 6.4: Environment modifiers `cga-meta` elements

The environment referees point at the accessible environment modifiers. Each environment modifier has an attribute `ref` containing the `xml:id` of the element to which it points. The algorithms presented in Section 6.2 explain how to follow them.	
`<env-pre-external>`	Points at a preceding content which is external to the current position.
`<env-pre-contained>`	Points at the direct container of the current position.
`<env-pre-local>`	Points at the preceding content which is in the local scoping of the current position.
`<env-post-internal>`	Points at the last internal (sub) content of the current position.

Table 6.5: Environment referees `cga-meta` elements

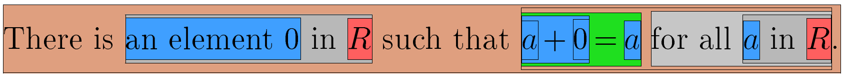
of the XML syntax is the abstract syntax on which the checker bases its analysis. But the input from users could have a different form. CGa's plain syntax and TSa-CGa macros for TeXmacs are examples of alternative input. In both cases, the first task of the checker is to transform these concrete representations into a CGa XML document. Note that this task corresponds to the simple parsing of the XML source document in the case of CGa XML syntax. The checker produces meta-information that are or could be useful for the human-user (error list, typing, structure). In the case of the CGa XML syntax, this kind of meta-information is directly added in the relevant position in the XML tree. In the case of other syntaxes, the additional `<loc:location>` elements inform about the location in the original syntax which corresponds to each XML node. For CGa's plain syntax, the `<loc:location>` element has three attributes namely the file name (`file`), the line number (`line`), the character number of the beginning (`begin`) and the ending (`end`) characters of the expression. In the case of TeXmacs, each TSa annotation (see Section 6.1.3) gets a unique identifier and this identifier is included as an `id` attribute of a `<loc:location>`.

6.1.2 CGa's plain syntax

For testing purposes and theoretical discussions, we developed a plain syntax for CGa (see Section 4.1). This syntax is a developer-oriented alternative to the CGa XML syntax. During the period of December 2005 to February 2006, P. van Tilburg worked in the ULTRA group for his internship [vT06]. He adapted and enhanced a Camlp4 parser we developed for MWTT [Maa03]. Thanks to this contribution to the project, we are able to provide an input syntax which follows exactly the CGa's abstract syntax of Section 4.1. Additionally, he developed some printers for this CGa's plain syntax representation. We refer to his report [vT06] for an exhaustive description of this implementation.

6.1.3 TSa-CGa macros for TeXmacs

In Chapter 5 we saw how we are using TeXmacs with the MathLang-TSa plugin as an input for CGa. In Section 5.1 we used one sentence as example, we repeat it here.



There is an element 0 in R such that $a+0=a$ for all a in R.

The TEXMACS tree corresponding to the internal encoding by TEXMACS could be printed as follows (using TEXMACS presentation facilities).

⟨MathLang-step|||There is ⟨MathLang-declaration|||⟨MathLang-term|0||an element ⟨with|*mode*|math|0⟩⟩ in ⟨MathLang-set|R||⟨with|*mode*|math|R⟩⟩⟩ such that ⟨MathLang-step|||⟨with|*mode*|math|⟨MathLang-statement|eq||⟨MathLang-term|plus|| ⟨MathLang-term|a||a⟩+⟨MathLang-term|0||0⟩⟩=⟨MathLang-term|a||a⟩⟩⟩ ⟨MathLang-context|||for all ⟨MathLang-declaration|||⟨with|*mode*|math|⟨MathLang-term|a||a⟩⟩ in ⟨with|*mode*|math|⟨MathLang-set|R||R⟩⟩⟩⟩⟩.⟩

This example illustrates our choice to represent each TSa annotation by a TEXMACS element with three arguments.

⟨MathLang-term|*interpretation*|*feedback*|*text*⟩

The first argument is the interpretation that the user inputs. The second argument contains the CGa checking feedback. These arguments are empty on input and are filled with error labels (see example in Section 5.1.5) by the plugin. The third argument contains the text annotated. Tables 5.2 and 5.3 describe the correlation between grammatical annotations and CGa's abstract syntax.

6.2 CGa's Type Checker

6.2.1 XML DOM

CGa is a language for computerising mathematical text. Its manner of representing the data is central in its design and therefore in its implementation. Figure 6.1 represents the overall architecture of the current CGa-TSa framework. The central element is the "DOM graph" which is the internal representation of the XML document. We implemented the CGa checker using a XML DOM interface [Wora], which means that the DOM graph is altered in place.

We see in Section 6.2.2 how we incorporate inside the DOM graph the environment, typing and typing error information respectively. This corresponds to the "Type checker" node of Figure 6.1.

We currently provide three different input formats (see Section 6.1). Here is the way we denoted them in Figure 6.1:

1. The CGa XML syntax is represented by the left hand side "XML document" node and the "DOM XML Parser" node.

2. The CGa's plain syntax is represented by the "Plain document" node and the Camlp4 "Plain Parser" node (right hand side).

3. The TSa-CGa macros for $\TeX_{\text{MACS}}$ permit one to input a TSa annotated document which is represented by the "$\TeX_{\text{MACS}}$ document" node (bottom right corner). We implemented a client-server communication between $\TeX_{\text{MACS}}$ and the CGa checker via our plugin for $\TeX_{\text{MACS}}$ (see Section 6.3.1). H. Lesourd from the Ωmega group at Universität des Saarlandes gave a practical help in developing the $\TeX_{\text{MACS}}$ plugin's client.

Additionally, a set of XSL transformations produce different overviews of a CGa document such as errors, types, environment, document's skeleton ("XSLT" grouping node in the figure). They use the typing result embedded in the document by the type checker.

6.2.2 Typing environment

CGa checker creates inside the XML DOM graph the typing environment. This is done prior to applying each typing rule. The typing environment is constructed with environment modifiers and environment referees (see Section 6.1.1.2.3). The environment referees give the route to follow to map the documents pieces in scoping.

6.2.2.1 Retrieving the typing environment

The search for environment items works as follows. It first aggregates the location points that are visible and then gets the content needed (list of identifiers, status of an identifier, type of an identifier).

Definition 7 (Containers) *We define the containers of an element e to be the sequence composed by:*

1. *The elements pointed at by the* `pre_contained` *sub-elements of e,*

2. *The containers of the elements pointed at by the* `pre_local` *sub-elements of e,*

3. *And the containers of the elements pointed at by the* `pre_external` *sub-elements of e.*

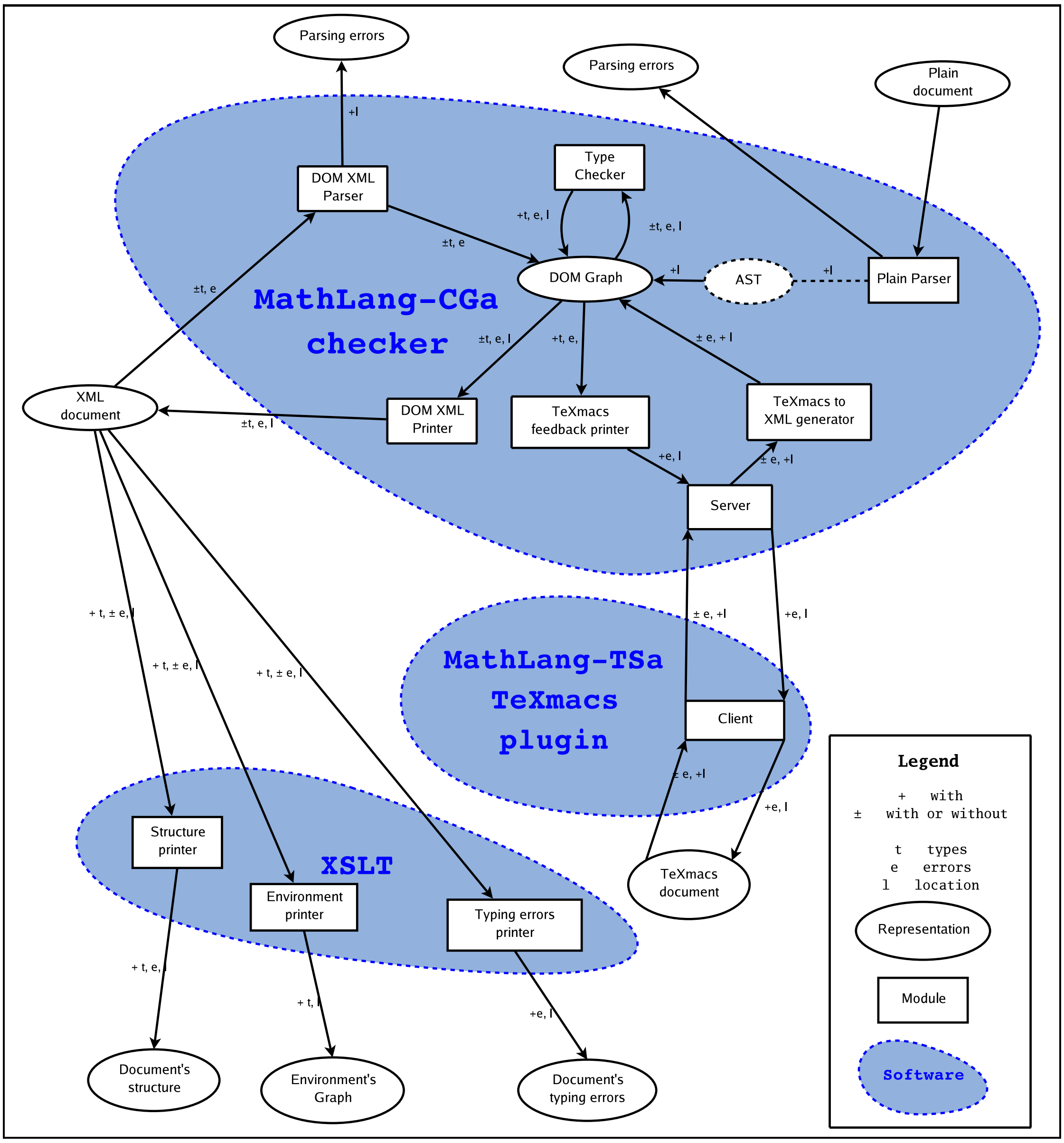


Figure 6.1: CGa-TSa system architecture

Diagram[a] describing the role of the different pieces of software in the current CGa-TSa framework. See Section 6.2.1 for additional explanations.

[a] This diagram stems from P. van Tilburg [vT06]. We present here an updated version of the diagram.

Definition 8 (Internal visibility) *We define the elements visible from an internal point of view at an element e to be the sequence composed by:*

1. *The elements internally visible at each container of e,*
2. *The elements externally visible at each element pointed at by the* `pre_local` *sub-elements of e,*
3. *And the elements externally visible at each element pointed at by the* `pre_external` *sub-elements of e.*

Definition 9 (External visibility) *We define the elements visible from an external point of view at an element e to be the sequence composed by:*

1. *The elements externally visible at each element pointed at by the* `pre_external` *sub-elements of e,*
2. *The elements externally visible at each element pointed at by the* `post_internal` *sub-elements of e,*
3. *And the element e itself.*

At any point of the XML tree it is therefore possible to retrieve the list of nodes which compose the typing environment. The typing environment at a CGa XML element is the sequence of elements visible from an internal point of view at this element. This environment is easily computable. The following Caml code computes it.

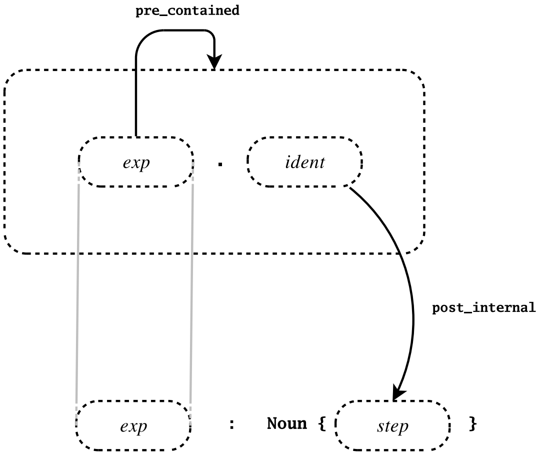
Listing 6.1: Functions written in Caml rebuilding the environment

```
let rec containers element =
  (pre_contained element)
  @ (List.flatten
       (List.map containers (pre_local element)))
  @ (List.flatten
       (List.map containers (pre_external element)))
and visible_internal element =
  (List.flatten
     (List.map visible_internal (containers element)))
  @ (List.flatten
       (List.map visible_external (pre_local element)))
  @ (List.flatten
       (List.map visible_external (pre_external element)))
and visible_external element =
  (List.flatten
     (List.map visible_external (pre_external element)))
  @ (List.flatten
       (List.map visible_external (post_internal element)))
  @ [element]
```

6.2.2.2 Environment referees

In this section we show the environment referees for the CGa language constructions of Section 4.1. Each environment referee is represented by an arrow labeled with the kind of referee. We use CGa's abstract syntax generic construction and represent by dash-border boxes the underlying cga XML elements. We ignore here the handling of labels/references.

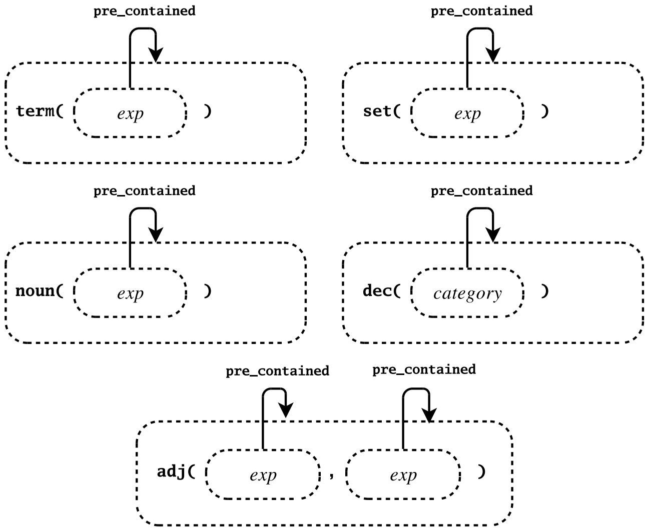
6.2.2.2.1 Vocabulary level We start here with the environment referees for characters. We use two environment referees to reflect the content of the CHARACTER rule. The pre_contained referee indicates that the left hand side premise $s \vdash e \mathbin{:} \mathit{Term}(m)$ of the rule and the conclusion of the rule share the same typing environment. The post_internal referees shows that we retrieve the term's character in the step describing the noun this term inhabits.



The lower part is a representation of the typing of the term (we used an CGa's plain syntax declaration instead of a typing judgment to show the step describing the noun expression).

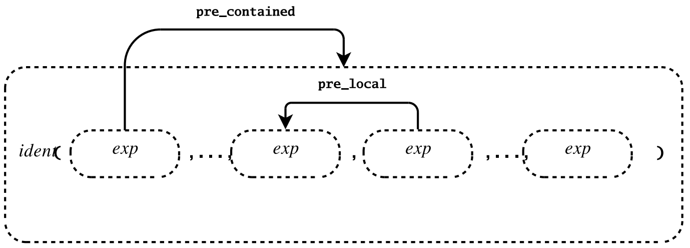
6.2.2.2.2 Category level

The environment referees for the category expressions are simply one `pre_contained` per case for the sub-expression. This `pre_contained` represents the fact that both premises and conclusion share the same typing environment in the CATEG-TERM, CATEG-NOUN, CATEG-SET, CATEG-DEC and CATEG-ADJ rules.

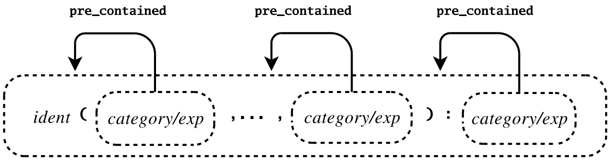


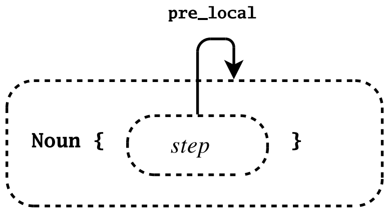
6.2.2.2.3 Expression level

The Environment referees for the instantiation encode the typing environment of the instantiation's arguments. Firstly, the `pre_local` referees show that each argument is typed in the environment composed by the preceding argument (and by extent the preceding argument's environment as well). Note the slight difference with the typing INSTANCE rule which includes in the environment only the arguments declaring a local variable. The reason was then not to break the typing with non-phrase expressions (i.e. expressions that can not be included as such in a step because they would not a be valid phrase). Here, we are only interested in showing the path to construct the environment. Secondly, the `pre_contained` referee indicates that the arguments share the same environment as the whole instance.

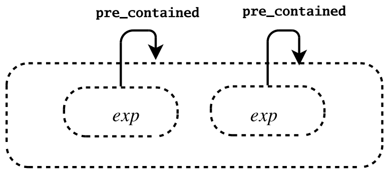


The environment referee for declaration simply indicates with one `pre_contained` per parameter that the typing environment for every parameter is the same as for the declaration itself (see DEC).

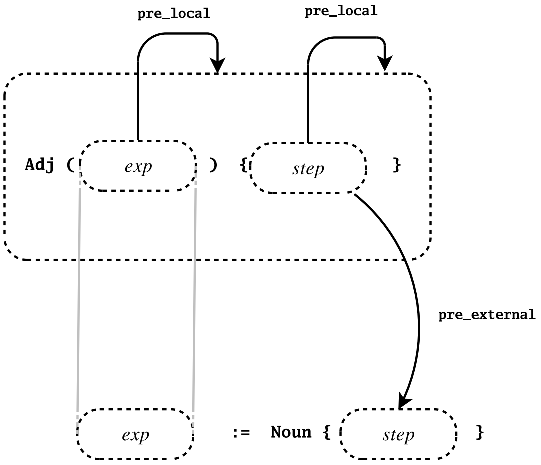




The step which describes a noun in the noun description construction is typed under the same environment as the noun description. Nevertheless, we use a `pre_local` environment referee to forbid access to future character search. This feature appears in the NOUN rule in the form of the premise $i \in dI(s') \cup DI(s')$ and the self marker.

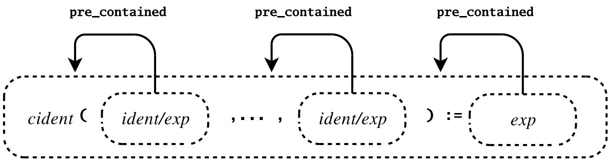


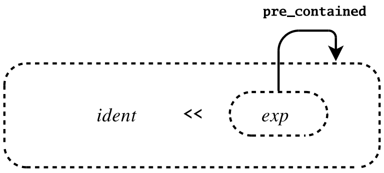
The environment referees for the refinement indicate that both premises and conclusion share the same typing environment in the TERM-REFINEMENT, SET-REFINEMENT and NOUN-REFINEMENT rules.



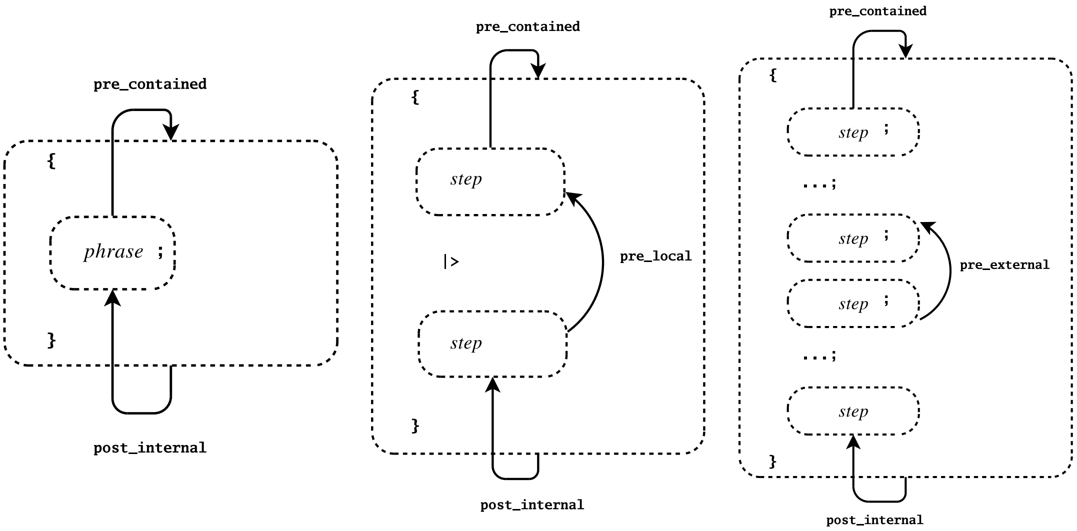
The environment referees for the adjective description construction are similar to the ones for character and for noun description. The similarity with the noun description comes from the `pre_local` referee we use. It forbids future external accesses. The reason for the similarity with the character's environment referees is the need to get the type of an expression to link with a `pre_external` to the step describing the adjective argument.

6.2.2.2.4 Phrase level The environment referees for definition and sub refinement are one `pre_contained` referee per argument and one `pre_contained` referee for the expression respectively.





6.2.2.2.5 Discourse level The environment referees for the phrase, local-scoping and block of step language constructions reflect the behavior of the rules for extracting an identifier's typing from the context (see Section 4.2.3.1.1) which are IDENT-BASIC, IDENT-LOCAL-SCOPING and IDENT-BLOCK respectively.



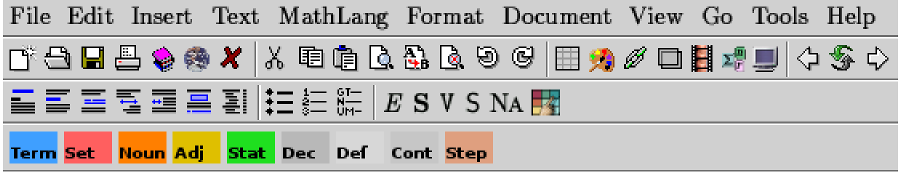
6.3 TSa's Natural Language Interface

This section summarises the methods we developed to handle natural language texts and to provide a user interface prototype. In Section 6.3.1 we present our effort to develop a software capable of integrating CGa encoding inside CML phrasing using TSa's methods. In Section 6.3.2 we present our early method for rendering MWTT in a CML form.

6.3.1 TSa's editing tools

The goal of our editing tools is to provide the mathematician with a user-friendly editor for CGa which gives a complete freedom in the phrasing. We used TEXMACS to develop our first prototype because it is both "What You See Is What You Get" and structured. We created the TSa-CGa plugin for TEXMACS to bring it to the level of "What You Edit Is What You Mean". In Chapter 5, we went into the details of the TSa authoring method of which the TSa-CGa plugin for TEXMACS is an implementation.

The TSa-CGa plugin for TEXMACS supplies several ways to input CGa and TSa elements in a TEXMACS document. One can access TSa functions via a "MathLang" menu, via an icon bar and via keyboard shortcuts. Appendix C reproduces the TSa-CGa plugin documentation (also accessible inside $\TeX_{\text{MACS}}$).


File Edit Insert Text MathLang Format Document View Go Tools Help
E S V S NA
Term Set Noun Adj Stat Dec Def Cont Step

The menu of the TSa-CGa plugin contains five groups of entries.

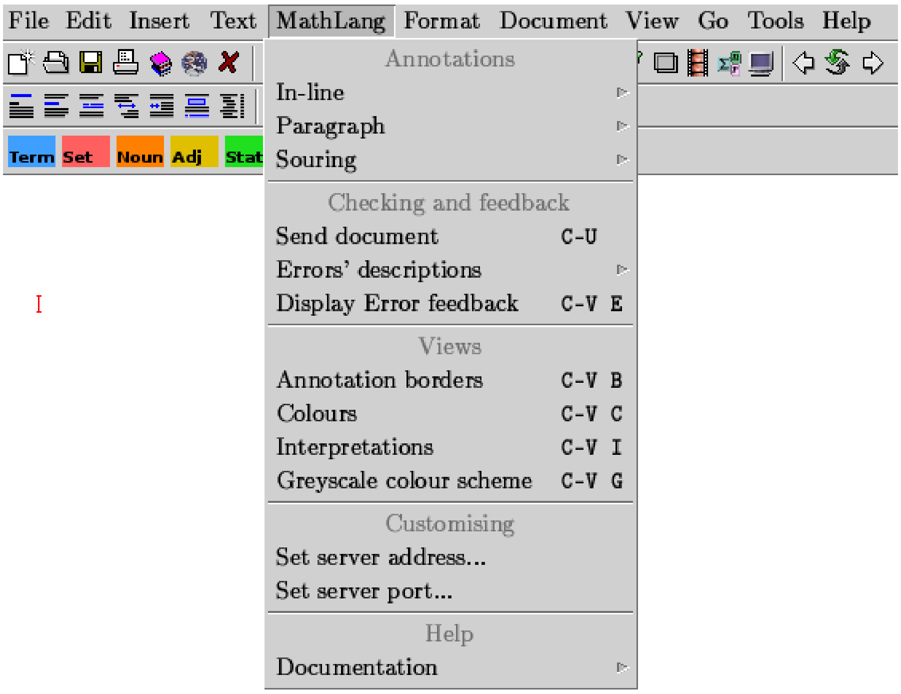
1. The "Annotations" group which provides grammatical annotations (in-line or paragraph width).

2. The "Checking and feedback" group contains the function to communicate the entire document to a CGa checker, and the sub-menu where the errors' descriptions will be listed.

3. The "Views" group which permits the user to customise the document's view (see Section 5.1.5 for examples of customised views).

4. The "Customising" group which could be used to set alternative addresses and ports for the connection to a CGa checker.

5. The "Help" group which gives access to the plugin's documentation.



Our prototype implementation of CGa's client editor is based on TeXmacs but it is possible to develop other clients for the CGa checker server. As we can see with this menu, the TSa-CGa plugin extends the editing facility and do not replace it.

6.3.2 Presentation tools in the previous MWTT implementation

In Section 3.4.1 we mentioned our early extension of WTT and in Section 4.4 we presented the syntax and type system of this language which we named MWTT. We developed an implementation of this type system. At the time, we also designed a system for customising the rendering views of MWTT documents.

This system uses several XSL transformations to generate a CML text with colour annotations showing the weak types in the document. This process takes into account the presentation information given by the author inside his MWTT code to generate an output which is close to the original CML version of the document.

With our development of TSa (direct annotation of CML text with grammatical and souring information), this previous rendering system became obsolete. Figure 3.4 describes the system's transformation procedure. We illustrate here the manner in which the natural language templates were encoded. We are using the example from Section 5.1. We consider the MWTT line corresponding to the entire sentence "There is an element 0 in R such that $a + 0 = a$ for all a in R". We would therefore define a template "There is $\boxed{1}$ such that $\boxed{2}$". The box $\boxed{1}$ being the context of the line and $\boxed{2}$ its body.

This template links one MWTT construction to its rendering view. Templates for MWTT variables, constants and binders are defined either globally or locally. One association could be reused at different locations of the document. They are defined using XSL with some MWTT-specific commands to easily refer to the information contained in the MWTT encoding. The template for the line above is as follows.

```
<template output="cml.tex" kind="xsl">
  <categ kind="par" boxed="no">
    <xsl:text>There is </xsl:text>
    <xsl:apply-templates select="context"/>
    <xsl:text> such that </xsl:text>
    <xsl:apply-templates select="body"/>
    <xsl:text>.</xsl:text>
  </categ>
```

```
    </template>
```

This information is a mix of XSL standard elements and MWTT specific ones. A first process transforms all the presentation data that coats the document into one single XSL transformation file. In a second process, this XSL transformation is applied to the document itself producing the CML rendering view.

6.4 TSa: Syntax Souring Operational System

In this section we define an operational system implementing the syntax souring transformations defined in Section 5.2. The set of syntax souring rewriting rules we define in this section are summarised in Appendix A.4. Most material in this section were published in [KLMW07].

6.4.1 Souring abstract syntax

The following develops a foundation for MathLang-TSa documents.

6.4.1.1 Conventions

We assume the following conventions: the symbol $\mathbb{N}$ denotes the natural numbers, an ordered pair is denoted $(a; b)$ and functions are taken to be sets φ of ordered pairs with a domain $\mathrm{dom}(\varphi) = \{a \mid \exists b \ni (a; b) \in \varphi\}$. A sequence is a function s for which $\mathrm{dom}(s) = \{n \mid 0 \leq n < k\}$ for some $k \in \mathbb{N}$ where $[] = \emptyset$ denotes the empty sequence and $[x_0, x_1, \ldots, x_n]$ denotes the sequence such that $s(i) = x_i$ for each $i \in \mathrm{dom}(s) = \{0, \ldots, n\}$. Upon that sequence is defined the metric $|s| = n+1$. We define s_1, s_2 to be the concatenation of sequences s_1 and s_2, as the new sequence s such that

$$s(i) = \begin{cases} s_1(i) \text{ for } & i \in \mathrm{dom}(s_1) \\ s_2(i) \text{ for } & i - k \in \mathrm{dom}(s_2) \text{ where } k = |s_1| \end{cases}$$

where $\mathrm{dom}(s_1, s_2) = \{0, \ldots, |s_1| + |s_2| - 1\}$. This concatenation operator is associative, so that $s_1, (s_2, s_3) = (s_1, s_2), s_3$. Furthermore, the identity provides the properties that $[], s = s$ and $s, [] = s$. Finally, we define several types referred to above.

6.4.1.2 Labels and documents

TSa documents are constructed out of elements of several different kinds. Each kind of element is denoted by a different label. We define $\mathcal{L}$ to be the set of labels with ℓ ranging over $\mathcal{L}$. The set $\mathcal{L}$ is the disjunction

$$\begin{array}{rcll} \mathcal{L} & = & \mathcal{F} & \text{(Formatting labels)} \\ & \cup & \mathcal{G} & \text{(Grammatical labels)} \\ & \cup & \mathcal{S} & \text{(Souring labels)} \end{array}$$

of sets that we define as follows.

6.4.1.2.1 Formatting instructions The set $\mathcal{F}$ consists of *formatting instructions.* This set varies depending on the typesetting system used (i.e., the formatting instructions used in a single document ought to be associated with one single typesetting system). Definition 13 formally defines the set $\mathcal{F}$ over which f ranges.

6.4.1.2.2 Grammatical labels The set $\mathcal{G}$ consists of *grammatical labels.* Each grammatical label is composed of a grammatical category and an interpretation.

$$\mathcal{G} = \mathcal{C} \times \mathcal{I}.$$

The set of grammatical categories is defined as

$$\mathcal{C} = \{\mathbf{term}, \mathbf{set}, \mathbf{noun}, \mathbf{adj}, \mathbf{stat}, \mathbf{decl}, \mathbf{defn}, \mathbf{step}, \mathbf{cont}\}.$$

It contains identifiers for the primitive grammatical categories used in TSa's annotations as enumerated in Table 5.1. The set $\mathcal{I}$ consists of strings used for identifying abstract interpretations (e.g., `0`, `R`, `eq`, `plus` and `a` are the interpretation strings used in the examples throughout Section 5.1). Let g, c and i range respectively over $\mathcal{G}$, $\mathcal{C}$ and $\mathcal{I}$.

6.4.1.2.3 Souring labels The set $\mathcal{S}$ defines the souring labels. This set, over which s ranges, is split into $\mathcal{S}_u$ and $\mathcal{S}_i$ ($\mathcal{S} = \mathcal{S}_u \cup \mathcal{S}_i$).

$$\begin{aligned}
\mathcal{S}_u &= \{\texttt{fold-left}, \texttt{fold-right}, \texttt{map}, \texttt{base}, \texttt{list}, \texttt{hook}, \texttt{loop}, \texttt{shared}\} \\
&\quad \cup (\{\texttt{position}\} \times \mathbb{N}) \\
\mathcal{S}_i &= \{\texttt{hook-travel}, \texttt{head}, \texttt{tail}, \texttt{daeh}, \texttt{liat}, \texttt{right-travel}, \texttt{left-travel}\} \\
&\quad \cup (\{\texttt{cursor}\} \times \mathbb{N})
\end{aligned}$$

The set $\mathcal{S}_u$ contains *souring identifiers* to be employed directly by the user while $\mathcal{S}_i$ holds several identifiers used internally for rewriting.

6.4.1.2.4 Documents A TSa document is an element of the set $\mathcal{D}$, see Definition 10. In addition, we denote by $\mathcal{D}_{\mathcal{F}}$, $\mathcal{D}_{\mathcal{G}}$, $\mathcal{D}_{\mathcal{F}\cup\mathcal{G}}$, $\mathcal{D}_{\mathcal{G}\cup\mathcal{S}}$ and $\mathcal{D}_{\mathcal{F}\cup\mathcal{G}\cup\mathcal{S}}$ the sets of documents for which labels are in $\mathcal{F}$, $\mathcal{G}$, $\mathcal{F} \cup \mathcal{G}$, $\mathcal{G} \cup \mathcal{S}$ and $\mathcal{F} \cup \mathcal{G} \cup \mathcal{S}$, respectively.

Definition 10 (Document) *Let $\mathcal{D}$ be the smallest set such that*

1. $[] \in \mathcal{D}$,
2. *if* $d \in \mathcal{D}$ *and* $\ell \in \mathcal{L}$ *then* $[(\ell; d)] \in \mathcal{D}$, *and*
3. *if both* d_1 *and* d_2 *are elements of* $\mathcal{D}$ *then* $(d_1, d_2) \in \mathcal{D}$.

Remark 10 (Notational conventions) *For convenience,* $[(\ell; d)]$ *abbreviates to* $\ell\langle d\rangle$*. Furthermore, when not ambiguous* $\ell\langle[]\rangle$ *abbreviates to* ℓ*. In the case of grammatical labels ordered pairs, we denote the interpretation (second element of the pair) by an adjoined superscript. A pair* $(c; i)$ *from* $\mathcal{G}$ *is denoted by* c^i*. Similarly, an ordered pair from* $\{\texttt{position}\} \times \mathbb{N}$ *(respectively* $\{\texttt{cursor}\} \times \mathbb{N}$*) is denoted by a superscript number (second element of the pair) adjoined to* `position` *(respectively* `cursor`*).*

Upon $\mathcal{D}$ is defined the following relations.

Definition 11 (Sub-document) *We define* sub-document*, and we denote by* $\sqsubset_{\mathcal{G}}$ *, the binary relation between documents such that:*

$$d \sqsubset_{\mathcal{G}} d \tag{SUB1}$$

$$d \sqsubset_{\mathcal{G}} g\langle d_1\rangle \textit{ if } d \sqsubset_{\mathcal{G}} d_1 \tag{SUB2}$$

$$d \sqsubset_{\mathcal{G}} (d_1, d_2) \textit{ if } d \sqsubset_{\mathcal{G}} d_1 \textit{ or } d \sqsubset_{\mathcal{G}} d_2 \tag{SUB3}$$

Remark 11 *It is important to notice that our sub-document property* (SUB2) *is restricted to grammatical labels which means that for any label $\ell \notin \mathcal{G}$ and any documents d_1 and d_2 such that $d_1 \sqsubset_{\mathcal{G}} d_2$, we have that $d_1 \not\sqsubset_{\mathcal{G}} \ell\langle d_2\rangle$.*

Example 24 (Abstract syntax) *Let us illustrate this encoding with a sub-part of the example we used in Section 5. We consider the expression "an element* 0 *in R" which we annotated as follow:* $\boxed{\boxed{^{\texttt{<0>}}\text{an element } 0}\ \text{in}\ \boxed{^{\texttt{<R>}}R}}$. *This expression is equivalent to the following document where formatting instructions are printed as* `"m"`*, `m` being a string using a similar syntax to the one of of LaTeX macros' definition body.*

$$\mathbf{decl}\left\langle\texttt{"\#1 in \#2"}\left\langle\mathbf{term}^{\texttt{0}}\left\langle\texttt{"an element \$0\$"}\right\rangle, \mathbf{set}^{\texttt{R}}\left\langle\texttt{"\$R\$"}\right\rangle\right\rangle\right\rangle$$

6.4.1.2.5 Document filters

Definition 12 (Label inclusion) *We define* label inclusion, *and we denote by $\widetilde{\in}_{\mathcal{G}}$, the binary relation between a label and a document such that:*

$$\ell \mathrel{\widetilde{\in}_{\mathcal{G}}} \ell\langle d\rangle \tag{INC1}$$

$$\ell \mathrel{\widetilde{\in}_{\mathcal{G}}} g\langle d\rangle \text{ if } \ell \neq g \text{ and } \ell \mathrel{\widetilde{\in}_{\mathcal{G}}} d \tag{INC2}$$

$$\ell \mathrel{\widetilde{\in}_{\mathcal{G}}} (d_1, d_2) \text{ if } \ell \mathrel{\widetilde{\in}_{\mathcal{G}}} d_1 \text{ or } \ell \mathrel{\widetilde{\in}_{\mathcal{G}}} d_2 \tag{INC3}$$

Remark 12 *It is important to notice that our label inclusion property* (INC2) *is restricted to grammatical labels, which means that for any labels $\ell_1 \notin \mathcal{G}$ and $\ell_2 \in \mathcal{L}$ such that $\ell_1 \neq \ell_2$, and any document d such that $\ell_2 \mathrel{\widetilde{\in}_{\mathcal{G}}} d$, we have that $\ell_2 \mathrel{\widetilde{\notin}_{\mathcal{G}}} \ell_1\langle d\rangle$.*

The two main uses of documents are rendering and grammatical interpretation. There are separate filters apropos to each situation.

Definition 13 (Rendering functions) *Let $\mathrm{f} : \mathcal{D} \to \mathcal{D}_{\mathcal{F}}$ be a function defined such that:*

$$\mathrm{f}([]) = [] \tag{FORM1}$$

$$\mathrm{f}(\ell\langle d\rangle) = \begin{cases} \ell\langle \mathrm{f}(d)\rangle & \text{if } \ell \in \mathcal{F} \\ \mathrm{f}(d) & \text{otherwise} \end{cases} \tag{FORM2}$$

$$\mathrm{f}(d_1, d_2) = \mathrm{f}(d_1), \mathrm{f}(d_2) \tag{FORM3}$$

Thus, f *flattens a given document d at any label from $\mathcal{G}$ or $\mathcal{S}$, removing all such labels. Once this is achieved, it will be possible to use* $\mathrm{r} : \mathcal{D}_{\mathcal{F}} \to \mathcal{F}$*, defined so that:*

$$\mathrm{r}([]) = \varepsilon \tag{REN1}$$

$$\mathrm{r}(f\langle d\rangle) = \mathrm{fill}(f, [\mathrm{r}(d(0)), \ldots, \mathrm{r}(d(|d|-1))]) \tag{REN2}$$

$$\mathrm{r}(d_1, d_2) = \mathrm{r}(d_1) \bullet \mathrm{r}(d_2) \tag{REN3}$$

Where, in a specific typesetting system, ε is the blank formatting instruction, $\bullet$ is the concatenation operator and fill *is a formatting-system-specific function. The function* fill *interprets a formatting instruction (first argument) with a sequence of rendered documents (second argument) passed as argument. One can imagine a formatting instruction to be a template with holes and* fill *to simply fill these holes. The number of vacancies exhibited by the first argument of* fill *should be equal to the length of the sequence (which is the second argument of* fill*). The function* fill *returns an element of the set $\mathcal{F}$ which should be a formatting instruction awaiting for no argument.*

Remark 13 *According to Definition 13, the number of vacancies exhibited by the first argument of* fill *should be equal to the length of the sequence, second argument of* fill*. But it may happen that a formatting instruction f is paired with a longer sequence in a $\mathcal{D}$-document d. Nevertheless this formatting instruction should see its paired sequence decrease in length by the application of the function* f *(case FORM2 can make a sequence to decrease in length).*

Definition 14 (De-formatting function) *To prepare a document for souring, we define* $\mathrm{df} : \mathcal{D} \to \mathcal{D}_{\mathcal{G}\cup\mathcal{S}}$ *as a function which strips a document of all formatting entities.*

$$\mathrm{df}([]) = [] \tag{DF1}$$

$$\mathrm{df}(\ell\langle d\rangle) = \begin{cases} d & \textit{if } \ell \in \mathcal{F} \\ \ell\langle \mathrm{df}(d)\rangle & \textit{otherwise} \end{cases} \tag{DF2}$$

$$\mathrm{df}(d_1, d_2) = \mathrm{df}(d_1), \mathrm{df}(d_2) \tag{DF3}$$

6.4.2 The souring rewriting system

In this section we formally define the souring rewriting rules [BN98] of the souring transformations given in Section 5.2.

Definition 15 (Compatibility) *We define the following compatibility property for a rewriting rule $\rightarrow_n$.*

$$d_1, d, d_2 \rightarrow_n d_1, d', d_2 \text{ if } d \rightarrow_n d' \qquad \text{(COMP1)}$$

$$g\langle d\rangle \rightarrow g\langle d'\rangle \text{ if } d \rightarrow_n d' \qquad \text{(COMP2)}$$

Remark 14 *Note that our compatibility rule* (COMP2) *is restricted to grammatical labels.*

Definition 16 (Reflexive transitive closure) *We denote by $\twoheadrightarrow_n$ the reflexive transitive closure of $\rightarrow_n$.*

Definition 17 (Normal form) *We define the n-normal form relatively to $\rightarrow_n$ and we denote by NF_n the property on a document d such that no $\twoheadrightarrow_n$ rewriting can be applied to d.*

6.4.2.1 The souring rewriting rules

Below are the formal rewriting rules for souring transformations from Section 5.2.1.

6.4.2.1.1 The list machinery The $\rightarrow_{list}$ rewriting rules describe how operations on list are processed. Four internal souring labels from $\mathcal{S}_u$ labels: `head`, `tail`, `daeh` and `liat` (the word `daeh` is the reverse of `head` and `liat` is `tail` reversed). Combined with a sequence starting by a `list` label, they disappear and alter the sequence paired with `list` during rewriting. In L9 (respectively L10, L11 and L12), the `head` (respectively `tail`, `daeh` and `liat`) isolate the head (respectively tail, last element and head elements) of the `list` paired sequence. The rules L1–L8 show

how these internal souring labels can navigate in a document in search for a `list`.

$$\begin{aligned}
\texttt{head}\langle d_1, d_2\rangle &\rightarrow_{\mathit{list}} d_1, \texttt{head}\langle d_2\rangle \quad \text{where } \texttt{list} \mathrel{\widetilde{\notin}_{\mathcal{G}}} d_1 \tag{L1}\\
\texttt{tail}\langle d_1, d_2\rangle &\rightarrow_{\mathit{list}} d_1, \texttt{tail}\langle d_2\rangle \quad \text{where } \texttt{list} \mathrel{\widetilde{\notin}_{\mathcal{G}}} d_1 \tag{L2}\\
\texttt{daeh}\langle d_1, d_2\rangle &\rightarrow_{\mathit{list}} d_1, \texttt{daeh}\langle d_2\rangle \quad \text{where } \texttt{list} \mathrel{\widetilde{\notin}_{\mathcal{G}}} d_1 \tag{L3}\\
\texttt{liat}\langle d_1, d_2\rangle &\rightarrow_{\mathit{list}} d_1, \texttt{liat}\langle d_2\rangle \quad \text{where } \texttt{list} \mathrel{\widetilde{\notin}_{\mathcal{G}}} d_1 \tag{L4}\\
\texttt{head}\langle g\langle d_1\rangle, d_2\rangle &\rightarrow_{\mathit{list}} g\langle \texttt{head}\langle d_1\rangle\rangle, d_2 \quad \text{where } \texttt{list} \mathrel{\widetilde{\in}_{\mathcal{G}}} d_1 \tag{L5}\\
\texttt{tail}\langle g\langle d_1\rangle, d_2\rangle &\rightarrow_{\mathit{list}} g\langle \texttt{tail}\langle d_1\rangle\rangle, d_2 \quad \text{where } \texttt{list} \mathrel{\widetilde{\in}_{\mathcal{G}}} d_1 \tag{L6}\\
\texttt{daeh}\langle g\langle d_1\rangle, d_2\rangle &\rightarrow_{\mathit{list}} g\langle \texttt{daeh}\langle d_1\rangle\rangle, d_2 \quad \text{where } \texttt{list} \mathrel{\widetilde{\in}_{\mathcal{G}}} d_1 \tag{L7}\\
\texttt{liat}\langle g\langle d_1\rangle, d_2\rangle &\rightarrow_{\mathit{list}} g\langle \texttt{liat}\langle d_1\rangle\rangle, d_2 \quad \text{where } \texttt{list} \mathrel{\widetilde{\in}_{\mathcal{G}}} d_1 \tag{L8}\\
\texttt{head}\langle \texttt{list}\langle g\langle d_1\rangle, d_2\rangle, d_3\rangle &\rightarrow_{\mathit{list}} g\langle d_1\rangle, d_3 \tag{L9}\\
\texttt{tail}\langle \texttt{list}\langle g\langle d_1\rangle, d_2\rangle, d_3\rangle &\rightarrow_{\mathit{list}} d_2, d_3 \tag{L10}\\
\texttt{daeh}\langle \texttt{list}\langle d_1, g\langle d_2\rangle\rangle, d_3\rangle &\rightarrow_{\mathit{list}} g\langle d_2\rangle, d_3 \tag{L11}\\
\texttt{liat}\langle \texttt{list}\langle d_1, g\langle d_2\rangle\rangle, d_3\rangle &\rightarrow_{\mathit{list}} d_1, d_3 \tag{L12}
\end{aligned}$$

6.4.2.1.2 Sharing The $\rightarrow_{\mathit{share}}$ rewriting rule is the operational definition of the sharing transformation of Definition 2. It duplicates the document paired with a `share` souring label at both the tail of a preceding grammatical label's paired sequence and the head of a following grammatical label's paired sequence.

$$g_1 \left\langle d_1 \right\rangle, \texttt{shared} \left\langle d \right\rangle, g_2 \left\langle d_2 \right\rangle \rightarrow_{\mathit{share}} g_1 \left\langle d_1, d \right\rangle, g_2 \left\langle d, d_2 \right\rangle \tag{S1}$$

6.4.2.1.3 Chaining The $\twoheadrightarrow_{\mathit{chain}}$ rewriting rule is the operational definition of the chaining transformation of Definition 3. The `hook` souring label is replaced in C1 by a copy of its paired document and by a `hook-travel` paired with a copy of this paired document. The $\mathcal{S}_i$ `hook-travel` internal souring label travels with C3–C5 to reach a `loop`. Rule C2 specifies that when a `hook-travel` meets a `loop`, they both disappear and are replaced by one copy of the document paired with `hook-travel`.

$$\texttt{hook}\left\langle d\right\rangle \rightarrow_{chain} d, \texttt{hook-travel}\left\langle d\right\rangle \tag{C1}$$

$$\texttt{hook-travel}\left\langle d\right\rangle, \texttt{loop} \rightarrow_{chain} d \tag{C2}$$

$$\texttt{hook-travel}\left\langle d_0\right\rangle, d_1, d_2 \rightarrow_{chain} d_1, \texttt{hook-travel}\left\langle d_0\right\rangle, d_2 \quad \text{where } \texttt{loop}\ \widetilde{\not\sqsubset}_{\mathcal{G}}\ d_1 \tag{C3}$$

$$\texttt{hook-travel}\left\langle d_0\right\rangle, g\left\langle d_1\right\rangle \rightarrow_{chain} g\left\langle \texttt{hook-travel}\left\langle d_0\right\rangle, d_1\right\rangle \tag{C4}$$

$$g\left\langle d_1, \texttt{hook-travel}\left\langle d_0\right\rangle\right\rangle \rightarrow_{chain} g\left\langle d_1\right\rangle, \texttt{hook-travel}\left\langle d_0\right\rangle \tag{C5}$$

6.4.2.1.4 Mapping The $\twoheadrightarrow_{map}$ rewriting rule is the operational definition of the mapping transformation of Definition 6. The mapping rewriting reproduces the document paired with a $\mathcal{S}_u$ `map` souring label for each element of the sequence associated with the `list` souring sub-label. Rule M2 replaces the mapping pattern by two version of the document paired with `map`. The first one has its sub-pair `list`-sequence replaced by the head of the sequence (the rule makes use of $\twoheadrightarrow_{list}$ and `head` of Section 6.4.2.1.1). The second one is paired with `map` and has its sub-pair `list`-sequence replaced by the tail of the sequence (the rule makes use of $\twoheadrightarrow_{list}$ and `tail` of Section 6.4.2.1.1). Rule M1 simply states that a pair of a `map` and a document containing an empty list `list` (or precisely `list`$\langle[]\rangle$) gets removed by this rewriting.

$$\texttt{map}\langle d\rangle \rightarrow_{map} [] \quad \text{where } \texttt{list} \sqsubset_{\mathcal{G}} d \tag{M1}$$

$$\texttt{map}\langle d_0\rangle \rightarrow_{map} d_1, \texttt{map}\langle d_2\rangle \quad \text{where} \quad \begin{aligned} & d_0 \twoheadrightarrow_{souring} d_0', \\ & \texttt{head}\langle d_0'\rangle \twoheadrightarrow_{list} d_1 \\ & \text{and } \texttt{tail}\langle d_0'\rangle \twoheadrightarrow_{list} d_2 \end{aligned} \tag{M2}$$

6.4.2.1.5 Folding The $\twoheadrightarrow_{fold}$ rewriting rule is the operational definition of the right and left folding transformations of Definitions 4 and 5. Right folding works in similar fashion to mapping. Rule F1 duplicates the folding pattern for the head and the tail of the list (the rule makes use of $\twoheadrightarrow_{list}$, `head` and `tail` of Section 6.4.2.1.1). `right-travel` travels (rules F2 and F3) to reach the `base`. If the list is empty in the pattern, only the base is left (rule F4) otherwise a new `fold-right` replaces the base (rule F5).

$$\texttt{fold-right}\langle d_0\rangle \rightarrow_{\mathit{fold}} \texttt{right-travel}\langle d_2\rangle, d_1 \tag{F1}$$

$$\text{where } d_0 \twoheadrightarrow_{\mathit{souring}} d_0',\ \texttt{head}\langle d_0'\rangle \twoheadrightarrow_{\mathit{list}} d_1 \text{ and } \texttt{tail}\langle d_0'\rangle \twoheadrightarrow_{\mathit{list}} d_2$$

$$\texttt{right-travel}\langle d_1, d_2\rangle \rightarrow_{\mathit{fold}} d_1, \texttt{right-travel}\langle d_2\rangle \tag{F2}$$

$$\text{where } \texttt{base} \mathrel{\widetilde{\notin}_{\mathcal{G}}} d_1$$

$$\texttt{right-travel}\langle g\langle d_1\rangle, d_2\rangle \rightarrow_{\mathit{fold}} g\langle\texttt{right-travel}\langle d_1\rangle\rangle, d_2 \tag{F3}$$

$$\text{where } \texttt{base} \mathrel{\widetilde{\in}_{\mathcal{G}}} d_1$$

$$\texttt{right-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} d_2 \ \text{ where } \texttt{list} \sqsubset_{\mathcal{G}} d_1 \tag{F4}$$

$$\texttt{right-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} \texttt{fold-right}\langle d_1\rangle \tag{F5}$$

Left folding differs from right folding by using a left recursion on the list with `daeh` and `liat` from Section 6.4.2.1.1.

$$\texttt{fold-left}\langle d_0\rangle \rightarrow_{\mathit{fold}} \texttt{left-travel}\langle d_2\rangle, d_1 \tag{F6}$$

$$\text{where } d_0 \twoheadrightarrow_{\mathit{souring}} d_0',\ \texttt{daeh}\langle d_0'\rangle \twoheadrightarrow_{\mathit{list}} d_1 \text{ and } \texttt{liat}\langle d_0'\rangle \twoheadrightarrow_{\mathit{list}} d_2$$

$$\texttt{left-travel}\langle d_1, d_2\rangle \rightarrow_{\mathit{fold}} d_1, \texttt{left-travel}\langle d_2\rangle \tag{F7}$$

$$\text{where } \texttt{base} \mathrel{\widetilde{\notin}_{\mathcal{G}}} d_1$$

$$\texttt{left-travel}\langle g\langle d_1\rangle, d_2\rangle \rightarrow_{\mathit{fold}} g\langle\texttt{left-travel}\langle d_1\rangle\rangle, d_2 \tag{F8}$$

$$\text{where } \texttt{base} \mathrel{\widetilde{\in}_{\mathcal{G}}} d_1$$

$$\texttt{left-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} d_2 \ \text{ where } \texttt{list} \sqsubset_{\mathcal{G}} d_1 \tag{F9}$$

$$\texttt{left-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} \texttt{fold-left}\langle d_1\rangle \tag{F10}$$

6.4.2.1.6 Reordering The $\twoheadrightarrow_{\mathit{pos}}$ rewriting rule is the operational definition of the re-ordering transformation of Definition 1. As one may expect this rule work as a sorting engine. Rule P1 re-orders position elements depending on their number attribute. An internal $\mathcal{S}_i$ souring label `cursor` takes the role of a marker. Its preceding elements are in order while the following ones still need to be ordered.

$$\texttt{position}^i\langle d_1\rangle, \texttt{position}^j\langle d_2\rangle \rightarrow_{pos} \texttt{position}^j\langle d_2\rangle, \texttt{position}^i\langle d_1\rangle \quad \text{where } j < i \tag{P1}$$

$$\ell\langle \texttt{position}^1\langle d_1\rangle, d_2\rangle \rightarrow_{pos} \ell\langle d_1, \texttt{cursor}^1, d_2\rangle \tag{P2}$$

$$\texttt{cursor}^i, \texttt{position}^{i+1}\langle d_1\rangle, d_2 \rightarrow_{pos} d_1, \texttt{cursor}^{i+1}, d_2 \tag{P3}$$

$$\ell\langle d, \texttt{cursor}^i\rangle \rightarrow_{pos} \ell\langle d\rangle \tag{P4}$$

Remark 15 (Reordering restriction) *Equations* (P2) *and* (P4) *restrict the ordering rewriting to a sequence of documents inside a label ℓ. The ordering rules cannot rewrite something of the form*

$$\texttt{position}^1\langle d_1\rangle, \ldots, \texttt{position}^n\langle d_n\rangle,$$

whereas it could rewrite

$$\ell\langle \texttt{position}^1\langle d_1\rangle, \ldots, \texttt{position}^n\langle d_n\rangle\rangle.$$

6.4.2.1.7 Souring rewriting rule We can finally define the souring rewriting rule as being a combination of the previous rules.

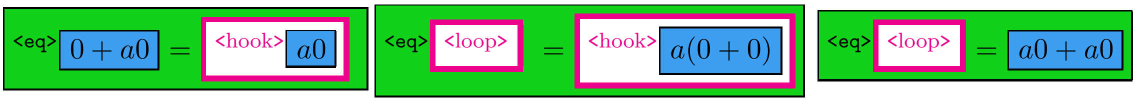
Definition 18 (Souring rewriting rule) *The souring rewriting rule, denoted by $\rightarrow_{souring}$ is defined as follow:*

$$d_0 \rightarrow_{souring} d_4$$

where

$$\begin{array}{lll} d_0 & \twoheadrightarrow_{share} d_1 & (d_1 \text{ being a } NF_{share}), \\ d_1 & \twoheadrightarrow_{chain} d_2 & (d_2 \text{ being a } NF_{chain}), \\ d_2 & \twoheadrightarrow_{pos} d_3 & (d_3 \text{ being a } NF_{pos}), \\ d_3 & \twoheadrightarrow_{lists} d_4 & (d_4 \text{ being a } NF_{lists}). \end{array}$$

Example 25 *Let us illustrate the souring rewriting with a concrete example. We consider the formula* $0+a0 = a0 = a(0+0) = a0+a0$ *where* $a0$ *stands for the multiplication of* a *and* 0*. Here is its annotated version using two chainings.*



Let us see step-by-step the souring rewriting.

1. *Here is its representation using our souring abstract syntax.*

   $\mathbf{stat}^{\mathtt{eq}}\left\langle \underline{0+a0}, \mathtt{hook}\left\langle \underline{a0}\right\rangle\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \mathtt{loop}, \mathtt{hook}\left\langle \underline{a(0+0)}\right\rangle\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \mathtt{loop}, \underline{a0+a0}\right\rangle$

2. *In a first step, case C1 of rule* $\longrightarrow_{chain}$ *is applied (following its compatibility property).*

   $\mathbf{stat}^{\mathtt{eq}}\left\langle \underline{0+a0}, \underline{a0}, \mathtt{hook\text{-}travel}\left\langle \underline{a0}\right\rangle\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \mathtt{loop}, \mathtt{hook}\left\langle \underline{a(0+0)}\right\rangle\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \mathtt{loop}, \underline{a0+a0}\right\rangle$

3. *Then case C5.*

   $\mathbf{stat}^{\mathtt{eq}}\left\langle \underline{0+a0}, \underline{a0}\right\rangle, \mathtt{hook\text{-}travel}\left\langle \underline{a0}\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \mathtt{loop}, \mathtt{hook}\left\langle \underline{a(0+0)}\right\rangle\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \mathtt{loop}, \underline{a0+a0}\right\rangle$

4. *Case C4.*

   $\mathbf{stat}^{\mathtt{eq}}\left\langle \underline{0+a0}, \underline{a0}\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \mathtt{hook\text{-}travel}\left\langle \underline{a0}\right\rangle, \mathtt{loop}, \mathtt{hook}\left\langle \underline{a(0+0)}\right\rangle\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \mathtt{loop}, \underline{a0+a0}\right\rangle$

5. *Case C2.*

   $\mathbf{stat}^{\mathtt{eq}}\left\langle \underline{0+a0}, \underline{a0}\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \underline{a0}, \mathtt{hook}\left\langle \underline{a(0+0)}\right\rangle\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \mathtt{loop}, \underline{a0+a0}\right\rangle$

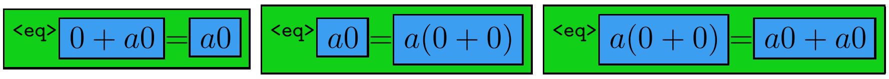
6. *This operation gets repeated for the second chain.*

   $\ldots$

7. *We finally obtain the following document.*

   $\mathbf{stat}^{\mathtt{eq}}\left\langle \underline{0+a0}, \underline{a0}\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \underline{a0}, \underline{a(0+0)}\right\rangle, \mathbf{stat}^{\mathtt{eq}}\left\langle \underline{a(0+0)}, \underline{a0+a0}\right\rangle$

   And here is this document in its annotated version.



Note that we showed the completion of first chaining souring rewriting first. Both chainings could happen in parallel.

6.4.2.2 Input document and document transformations

As a conclusion for the description of the souring operational rewriting system we compare here different authoring paths depending on the input document format.

Traditional natural language input method

Natural language text with grammatical annotations (Section 5.1)

Natural language text with grammatical and souring annotations (Section 5.2)

$$\begin{tikzcd}
\mathcal{D}_{\mathcal{F}} \arrow[d] & \mathcal{D}_{\mathcal{F}\cup\mathcal{G}} \arrow[dl, "f"] \arrow[d, "df"] & \mathcal{D}_{\mathcal{F}\cup\mathcal{G}\cup\mathcal{S}} \arrow[dll, bend left, "f"] \arrow[d, "df"] \\
\mathcal{D}_{\mathcal{F}} \arrow[d, "r"] & \mathcal{D}_{\mathcal{G}} \arrow[dr] & \mathcal{D}_{\mathcal{G}\cup\mathcal{S}} \arrow[d, "souring"] \\
\mathcal{F} & & \mathcal{D}_{\mathcal{G}}
\end{tikzcd}$$

Typesetting system's rendering on paper or screen, e.g. TEXMACS, MathML HTML, LATEX (Section 2.4.1)

Grammatical automatic checking (Section 4.2) and further computations or/and formalisations (Section 7.1.1)

Conclusion

This chapter gives a large overview of the implementation of MathLang's CGa and TSa aspects presented in the preceding chapters. We presented the XML default syntax, the plain syntax and the TSa syntax for CGa. We also presented the manner CGa's typing results are embedded in a CGa document and the rewriting system implementing the TSa syntax souring transformations. Since the souring rewriting rules are defined on top of a generic document format, it should be straightforward to adapt the rules to some specific formatting system and core "sour" language.

Chapter 7

Future Developments and Conclusion

In this chapter we reflect on the current and future developments related to this thesis and we give a conclusion to the thesis. We split these outcomes into three categories, the MathLang specific developments in Section 7.1, the possible reuse of some techniques developed in this thesis, for OpenMath and OMDoc in Section 7.2, and a general perspective on document management in Section 7.3.

7.1 MathLang's Current and Future of Developments

7.1.1 Current and Future aspects

The MathLang project initiated in 2000 by F. Kamareddine and J.B. Wells advocated the computerisation/formalisation of mathematical documents by means of aspects. This thesis reported on two aspects: CGa and TSa. The development of TSa involved the collaboration of another PhD research student in the MathLang project, R. Lamar. The forthcoming PhD thesis of another PhD student in the MathLang project, K. Retel, concerns the development of the Document Rhetorical aspect Document Rhetorical aspect (DRa). We briefly describe DRa in Section 7.1.1.2. Then we review other aspect possibilities, but these could change as progress on the MathLang project is made.

7.1.1.1 Extending CGa and TSa

Current known shortcomings in the TSa-CGa system include good handling of expressions with ellipses, such as $\overbrace{x + \ldots + x}^{n \text{ times}}$, $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$, and $\frac{1}{1+\frac{1}{1+\cdots}}$. Other priorities are developing and integrating methods to automate the annotation process, which at present can be redundant and tedious (see Section 7.1.2). These automated recognition tools would permit to ask mathematicians to test the system (up to now the computer scientists working in the Mathematical Knowledge Management field have been the only users of the system).

7.1.1.2 Document Rhetorical aspect (DRa)

In CGa a document is decomposed into steps either put in a sequence or contextualise by the local-scoping construction. One would encode division labels (such as chapter, section) and narrative labels (such as axiom, theorem, proof) by this unique step construction (see Chapters 4 and 5), our ground level language CGa does not make the difference between them to enhance its flexibility. These labels need to be computerised and could be based on the CGa generic step construction. From a logical/mathematical point of view, a statement is a statement. The DRa is about the subjective value, the social context and the expected use by humans of such steps.

The DRa captures the author's understanding of what is a theorem or an axiom and what this labelling imposes for the relationship of the associated piece of text with the rest of the document. The DRa describes a fix set of relation kinds between labeled elements that could occur in a mathematical document (such as *justifies* or *uses*). These relations are either directly stated by the author or deduced from a pre-defined grammars (in such standard of user-defined grammars a label *theorem* would impose a relation *justifies* to a *proof*). Coherence between these stated relationships (for instance a step labeled *axiom* can not be at the same time related to a *proof* by a relation *justifies*) is then automatically validated.

Additionally, from these labels and relations, a graph of logical precedences is automatically extracted. This graph could be seen as a prognostication of the structure of a formal proof of the informal CML document. This aspect and the use of this aspect to move further into a formalisation in Mizar are presented in K. Retel's report [Ret05] and in the articles [KMRW07b] and [KMRW07a]. Further extensive details can be found in K. Retel's forthcoming PhD thesis.

7.1.1.3 Informal Justification aspect (IJa)

CGa has no words like *hence*, *by*, *since* which give clues about the logical coherence of text. Within CGa's local scoping the difference is not made between considerations, assumptions or hypothesis of declarations. The logical meaning of these context components will be exposed by this aspect and composed with DRa relations. The logical justifications to move from one step to another in the document will be made explicit by the author. These logical indications will be automatically checked and will result in a well formed proof with unchecked pieces. In the spirit of H. Barendregt's [Bar03] and S. Autexier and A. Fiedler's [AF06], IJa will point at the holes in the proof and will identify the proof obligations with no restriction on the logic to be used. IJa completes the initial CGa structure into a more formal and homogeneous document, keeping track of all holes (open places in reasonings) and proof obligations so that the mathematician knows exactly what remains to be done on the theory-in-construction.

7.1.1.4 Meta-logic aspect (MLa)

Texts written in CGa+DRa+IJa will contain all the logical and semantical information that a CML proof could have. Such a text will contain holes that need to be filled to reach logical completeness. The purpose of this aspect in the path to full formalisation is to provide a language to *describe* the logical framework the author wants to work with and to *use* this framework to fulfill the proof requirements. The well-formation and well-use of this logical content will be analysed by this aspect but the proof checking itself is to be done by further aspects or external systems. This aspect should adapt to different logical systems.

7.1.1.5 Universal indexing aspect (UIa)

The way to relate documents with each others is an important authoring activity. This aspect will bring all the methodology and tools to relate documents and their contents in a universal manner. This aspect will facilitate cooperation and sharing including version control. This aspect will re-use established web-techniques (W3C's URLocation, URIdentifier, URName, XPOINTER, XPATH and XLINK) and be inspired by the time-honoured librarians techniques.

7.1.2 Automatic annotation and recognition

The method we presented in Chapter 5 requires a lot of input from the author. In an accomplished framework, such annotation of text should be automated to be fully usable. Mathematical texts are a mixture of formula and natural language texts. CGa offers constructions to capture both in the same language.

7.1.2.1 Parsing formulas

Recognition-wise, formulas are already well structured. From formula already annotated, the author unconsciously indicates how to decompose a formula. This includes an indication on the precedence of symbols. This symbol precedence could be represented in lattices. This approach would be more generic than the one proposed in [MLUM06] of attributing to each symbol a precedence ponderation.

7.1.2.2 Dynamic natural language recognition

Natural language recognition has been a prolific area of research in computer science for decades. The approach we defined in Chapter 5 joint with the approach of an evolutionary corpus successfully used by Y. Baba and M. Suzuki [BS03], would be a good experimental starting point. Implementation-wise, many natural language text parsers would be suitable for this task as suggested by C. Brown.

7.1.2.3 Prospection

The annotation of natural language we presented in Chapter 5 will be useful in a wide variety of settings. One possible area of application is work being done with optical character recognition of mathematics. In the work of the Infty Project [KSSS06, RRSS06], for example, it is desirable to extract information from printed materials. As TSa annotations becomes capable of being automated, it will provide further aid to extracting semantic information from a document with as little hand-translation as possible.

7.2 Towards an OpenMath/OMDoc Checker

Someone willing to use OpenMath or OMDoc for encoding some mathematical knowledge has to put some effort into structuring his formulas with OpenMath and organising the document content accordingly to OMDoc guidances. This process could be frustrating as the level of encoding effort does not match the level of automatic analysis OpenMath and OMDoc tools provide (see Section 3.2.1.5).

7.2.1 Checking

As a matter of fact, a weak typing (such as MathLang-CGa or WTT) would be a realistic alternative to ECC and STS (see Section 2.4.2.1). Which OMDoc (see Section 2.4.2.2) content could be translated into CGa?

- The structure of an OMDoc document could be translated into a CGa *step*-based structure.

- Formal contents (i.e. `<FMP>` elements) could be automatically translated into CGa expressions.

- The formulas embedded in informal contents (i.e. `<CMP>` elements) could be also translated as the natural language part of the informal content could benefit from user direct TSa annotation..

We do not consider that the transformation from OMDoc to CGa could be automated but a semi-automation could make use of OMCD extended with CGa type signatures. For example `transc1`'s `sin` could get CGa's type signature `(term) -> term`.

7.2.1.1 A MathLang-CGa OMCD

A MathLang-CGa OMCD (see its definition in Listing 7.1) could provide symbols to express CGa types with OpenMath symbols. Following the guidelines of the articles [CC99] and [Dav99] we have created an OpenMath symbol for each CGa type listed in Section 4.2.1.1.

```
<CD xmlns="http://www.openmath.org/OpenMathCD">
  <CDName> mathlang
  <CDDate> 2007-04-29
  <CDVersion> 0.1
  <CDRevision> 1
  <CDStatus> experimental
  <Description>
    This CD contains symbols expressing CGa types.
  <!-- CGa types -->
  <CDDefinition> <Name> CGa-term
    <Description> This symbol represents CGa type "term".
  <CDDefinition> <Name> CGa-set
    <Description> This symbol represents CGa type "set".
  <CDDefinition> <Name> CGa-noun
    <Description> This symbol represents CGa type "noun".
  <CDDefinition> <Name> CGa-adjective
    <Description> This symbol represents CGa type "adjective".
  <CDDefinition> <Name> CGa-statement
    <Description> This symbol represents CGa type "statement".
  <CDDefinition> <Name> CGa-declaration
    <Description> This symbol represents CGa type "declaration".
  <!-- CGa signature -->
  <CDDefinition> <Name> CGa-type
    <Role>semantic-attribution
    <Description> A symbol to be used within an OpenMath attribute
                  to specify the CGa type of the object.
  <CDDefinition> <Name> CGa-in
    <Description> Sub element of "type" listing a symbol's
                  input types.
  <CDDefinition> <Name> CGa-out
    <Description> Sub element of "type" containing a symbol's
                  output type.
```

Listing 7.1: MathLang-CGa OMCD

For readability and brevity, we show only the opening tag of each XML element; instead we use indentation to express nesting.

Using the symbol `type` one can indicate the type signature of an OpenMath object with the attribution `<OMATTR>`. As an example we give here the OpenMath signature of the symbol `plus` of `arith1` using respectively ECC, STS and CGa. In the current state of the OpenMath public *Content Directories* no signatures (either in ECC or STS) are provided. Also there is no generic way to write signature. A signature is simply an OpenMath object.

Example 26 (OpenMath `arith1` `plus` with ECC attributes) *The type of* `plus` *given as an example in [CC99] is* $\Pi x : \mathtt{integer}.(\Pi y : \mathtt{integer}.\mathtt{integer})$.

```
<Signature name="plus">
  <OMOBJ>
    <OMBIND>
      <OMS cd="ecc" name="PiType">
      <OMBVAR>
        <OMATTR>
          <OMATP>
```

```
            <OMS cd="ecc" name="type">
            <OMS cd="ecc" name="integer">
          <OMV name="x">
        <OMATTR>
          <OMATP>
            <OMS cd="ecc" name="type">
            <OMS cd="ecc" name="integer">
          <OMV name="y">
      <OMS cd="ecc" name="integer">
```

Example 27 (OpenMath `arith1` `plus` with STS attributes) *The type of* `plus` *given as an example in [Dav99] is an associative application. The arguments are elements of an abelian semi group, so is the result.*

```
<Signature name="plus">
  <OMOBJ>
    <OMA>
      <OMS cd="sts" name="mapsto">
      <OMA>
        <OMS cd="sts" name="nassoc">
        <OMV name="AbelianSemiGroup">
      <OMV name="AbelianSemiGroup">
```

In the draft description of STS, three type constructors are described. `mapsto` *for applications,* `nassoc` *and* `nary` *for n-arguments of associative and non-associative functions. In STS a function could have a fixed number of arguments.*

```
<Signature name="minus">
  <OMOBJ>
    <OMA>
      <OMS cd="sts" name="mapsto">
      <OMV name="AbelianSemiGroup">
      <OMV name="AbelianSemiGroup">
      <OMV name="AbelianSemiGroup">
```

Simple types in STS are encoded as variables.

Example 28 (OpenMath `arith1` `plus` with CGa attributes) *The CGa type signature of the* ***`plus`*** *symbol of the Content Dictionary* ***`arith1`*** *will be written as follow in OpenMath.*

```
<Signature name="plus">
  <OMOBJ>
    <OMA>
```

```
<OMS name="cga-type" cd="mathlang">
<OMA>
  <OMS name="cga-in" cd="mathlang">
  <OMS name="cga-term" cd="mathlang">
  <OMS name="cga-term" cd="mathlang">
<OMA>
  <OMS name="cga-out" cd="mathlang">
  <OMS name="cga-term" cd="mathlang">
```

7.2.1.2 Translation from OpenMath to CGa

We do not intend here to define a proper translation between OpenMath and CGa but simply to give some hints for such translation. For a CGa of an OpenMath formula we need to specify some translations. The target domain of this translation is CGa's abstract syntax.

- The OpenMath application construction corresponds to CGa's instance in most cases. Due to CGa's weak typing (see Section 4.3.2.2), some OpenMath application would need to be represented in CGa with a specific n-ary application operator.

- The OpenMath attribution with `<OMS cd="mathlang" name="type">` attribute symbol will be translated, depending on the situation, into a declaration.

- OpenMath bindings would represented by CGa's declaration arguments passed to binder-identifiers.

7.2.2 Editing

OMDoc, is aimed to become a universal medium for communicating mathematical documents. As OMDoc is mainly intended for machine consumption, it remains human-user unfriendly due to the omnipresence of its XML syntax. There exists several OMDoc editors [GP03, Act03, Pro] but they always constrain the author to a rigid structure driven by OMDoc's grammar. We believe that CGa/TSa can help by offering a mathematician-oriented editor for OMDoc.

7.3 Towards a Comprehensible Literate Proving

The $\TeX_{\text{MACS}}$ plugin environment and method of editing causes TSa to be a *visual language*. Using visual languages for knowledge representation is becoming more popular, and its benefits are obvious. By displaying and editing the logical structure of a mathematical document, the categorisation of various portions of text is made more clear and the structure more lucid. This could certainly lead to a new generation of literate programming [Knu84b]. The perspective of a collaborative framework for the edition of formal documents in a manner merging literate proving and Mathematical Semantic Web is giving hope for the emergence of a new motto "*What You Edit Is What You Mean*".

> It is quite conceivable that MV, or variations of it, can have an impact on computing science. A thing that comes at once into mind, is the use of MV as an intermediate language in "expert systems". Another possible use might be formal or informal specification language for computer programs. *[dB87, §1.3]*

7.4 Conclusion

This PhD thesis is in keeping with the general goal of the MathLang project: faithful computerisation of mathematics such that one can combine the efficiency of formal systems with the expressiveness of traditional mathematical authoring.

We demonstrate in this thesis the relevance of a low-level encoding concerned only with the grammatical structure of mathematical argumentation. Such encoding does not attenuate the value of a computerisation if it comes with a validation system. The benefit is a clear identification of one aspect of the computerisation/formalisation process on which further computerisations can be based. We apply well-established techniques in type theory to define a formal language and its type system. Additionally, we provide a full implementation of this system.

We also demonstrate in this thesis the feasibility of restoring natural language as the primary input for mathematical authoring on computers. This method, which combines text annotations and syntax souring, will be beneficial for mathematicians as it permits the use of computer-assisted authoring without requiring skills in computer-based formalisation. Thus, this method will benefit the mathematical knowledge community as it makes the bridge between traditional and computerised mathematics. We formally define the syntax souring rewriting system and implement a prototype editor for this system.

This thesis provides a practical solution for representing mathematical knowledge and for connecting this knowledge with corresponding common mathematical text pieces. Understanding the essence of mathematical writings via its computerisation ensures progress in leaning, teaching, exploring and inventing mathematics.

Appendix A

MathLang's CGa and TSa, Summary

A.1 CGa's Abstract Syntax

Summary of MathLang-CGa's abstract syntax as defined in Section 4.1.

		Vocabulary level	
cident, ci	$::=$	ident	Identifier
	$\mid$	$\mathit{exp}.\mathit{ident}$	Character
		Category level	
$\mathit{category}$, c	$::=$	$\mathtt{term}(\mathit{exp})$	Term category
	$\mid$	$\mathtt{set}(\mathit{exp})$	Set category
	$\mid$	$\mathtt{noun}(\mathit{exp})$	Noun category
	$\mid$	$\mathtt{adj}(\mathit{exp}, \mathit{exp})$	Adjective category
	$\mid$	$\mathtt{stat}$	Statement category
	$\mid$	$\mathtt{dec}(\mathit{category})$	Declaration category
	$\mid$	cvar	Category variable
		Expression level	
exp, e	$::=$	$\mathit{cident}(\overrightarrow{\mathit{exp}})$	Instance
	$\mid$	$\mathit{ident}(\overrightarrow{\mathit{category} \mid \mathit{expr}}) : \mathit{exp}$	Elementhood declaration
	$\mid$	$\mathit{ident}(\overrightarrow{\mathit{category} \mid \mathit{expr}}) : \mathit{category}$	Declaration
	$\mid$	$\mathtt{Noun}\ \{\mathit{step}\}$	Noun description
	$\mid$	$\mathtt{Adj}(\mathit{exp})\ \{\mathit{step}\}$	Adjective description
	$\mid$	$\mathit{exp}\ \mathit{exp}$	Refinement
	$\mid$	$\mathtt{self}$	Self

	$\mid$	`ref` *label*	Step reference
		Phrase level	
phrase, *p*	$::=$	*exp*	Statement phrase
	$\mid$	$\mathit{cident}(\overrightarrow{\mathit{ident}}) := \mathit{exp}$	Definition
	$\mid$	$\mathit{cident}(\overrightarrow{\mathit{exp}}) := \mathit{exp}$	Case definition
	$\mid$	$\mathit{ident} \ll \mathit{exp}$	Sub refinement
		Discourse level	
step, *s*	$::=$	*phrase*	Basic step
	$\mid$	$\mathit{step} \;\rhd\; \mathit{step}$	Local scoping
	$\mid$	$\{\overrightarrow{\mathit{step}}\}$	Block
	$\mid$	`label` *label step*	Step label

A.2 CGa's Type System

Summary of MathLang-CGa's typing rules as defined in Section 4.2.

A.2.1 Rules for the vocabulary level

$$\frac{\{\vec{s}\} \vdash p \mathrel{:} \mathit{Dec}(t) \qquad dI(p) = \{i\}}{\{\vec{s}; p\} \vdash i \mathrel{:} t}\ \textsc{ident-dec}$$

$$\frac{\{\vec{s}\} \vdash p \mathrel{:} \mathit{Def}(t) \qquad DI(p) = \{i\}}{\{\vec{s}; p\} \vdash i \mathrel{:} t}\ \textsc{ident-def}$$

$$\frac{\{\vec{s}\} \vdash p \mathrel{:} \mathit{Sub}(t) \qquad I(p) = \{i\}}{\{\vec{s}; p\} \vdash i \mathrel{:} t}\ \textsc{ident-sub}$$

$$\frac{s \vdash e \mathrel{:} \mathit{Term}(m) \qquad i \in \mathrm{dom}(m)}{s \vdash e.i \mathrel{:} m(i)}\ \textsc{character}$$

$$\frac{\{\overrightarrow{s}\} \vdash i : t \qquad i \notin I(p)}{\{\overrightarrow{s};\ p\} \vdash i : t}\ \textsc{ident-basic}$$

$$\frac{\{\overrightarrow{s}; s'; s''\} \vdash i : t \qquad i \in I(s'')}{\{\overrightarrow{s};\ s' \rhd s''\} \vdash i : t}\ \textsc{ident-local-scoping}$$

$$\frac{\{\overrightarrow{s_1};\ \overrightarrow{s_2}\} \vdash i : t}{\{\overrightarrow{s_1};\ \{\overrightarrow{s_2}\}\} \vdash i : t}\ \textsc{ident-block} \qquad \frac{\{\overrightarrow{s}; s'\} \vdash i : t}{\{\overrightarrow{s}; \texttt{label}\ l\ s'\} \vdash i : t}\ \textsc{ident-label}$$

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash e : \mathit{Noun}(m)}{s \vdash \texttt{term}(e)/\texttt{set}(e)/\texttt{noun}(e) : \mathit{Categ}(\mathit{Term}(m)/\mathit{Set}(m)/\mathit{Noun}(m))}\ \textsc{categ-}\begin{array}{l}\textsc{term}/\\ \textsc{set}/\\ \textsc{noun}\end{array}$$

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash e : \mathit{Noun}(m) \qquad s \vdash e' : \mathit{Noun}(m') \qquad m \preccurlyeq m'}{s \vdash \texttt{adj}(e, e') : \mathit{Categ}(\mathit{Adj}(m, m'))}\ \textsc{categ-adj}$$

$$\frac{\vdash s : \mathit{Step}}{s \vdash \texttt{stat} : \mathit{Categ}(\mathit{Stat})}\ \textsc{categ-stat}$$

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash c : \mathit{Categ}(a)}{s \vdash \texttt{dec}(c) : \mathit{Categ}(\mathit{Dec}(() \to a))}\ \textsc{categ-dec} \qquad \frac{\vdash s : \mathit{Step}}{s \vdash v : \mathit{Categ}(v)}\ \textsc{categ-var}$$

A.2.2 Rules for the expression level

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad s \vdash ci : (a_1, \ldots, a_n) \to a \\ \forall j \in \{1 \ldots n\},\ f = \mathrm{enum}(\{q \mid 1 < q < j \text{ and } dI(e_q) \neq \emptyset\}) \\ \text{and } \{s; e_{f(1)}; \ldots; e_{f(j-1)}\} \vdash e_j : a'_j \\ a' \notin \mathcal{V} \qquad (a_1, \ldots, a_n) \to a \bar{\preccurlyeq} (a'_1, \ldots, a'_n) \to a'\end{array}}{s \vdash ci(e_1, \ldots, e_n) : a'}\ \textsc{instance}$$

$$\frac{\begin{array}{c}\vdash s \boldsymbol{:} \mathit{Step} \qquad i \notin I(s) \qquad \forall j \in \{1 \ldots n\},\ \text{if } \mathit{ce}_j = c_j \text{ then } s \vdash c_j \boldsymbol{:} \mathit{Categ}(a_j) \\ \forall j \in \{1 \ldots n\},\ \text{if } \mathit{ce}_j = e_j \text{ then } s \vdash e_j \boldsymbol{:} \mathit{Noun}(m_j)/\mathit{Set}(m_j) \text{ and } a_j = \mathit{Term}(m_j) \\ \text{if } \mathit{ce} = c \text{ then } s \vdash c \boldsymbol{:} \mathit{Categ}(a) \\ \text{if } \mathit{ce} = e \text{ then } s \vdash e \boldsymbol{:} \mathit{Noun}(m)/\mathit{Set}(m) \text{ and } a = \mathit{Term}(m)\end{array}}{s \vdash i(c_1, \ldots, c_n) : e \boldsymbol{:} \mathit{Dec}((a_1, \ldots, a_n) \to a)}\ \textsc{dec}$$

$$\frac{\begin{array}{c}\vdash s \boldsymbol{:} \mathit{Step} \qquad \{s;\ \mathtt{self} : \mathtt{term}\} \vdash s' \boldsymbol{:} \mathit{Step} \\ \forall i \in \mathit{dI}(s') \cup \mathit{DI}(s'),\ \{s;\ \mathtt{self} : \mathtt{term};\ s'\} \vdash i \boldsymbol{:} m(i)\end{array}}{s \vdash \mathtt{Noun}\ \{s'\}\ \boldsymbol{:} \mathit{Noun}(m)}\ \textsc{noun}$$

$$\frac{\begin{array}{c}\vdash s \boldsymbol{:} \mathit{Step} \qquad s \vdash e \boldsymbol{:} \mathit{Noun}(m) \\ \{s;\ \mathtt{self} : e\} \vdash s' \boldsymbol{:} \mathit{Step} \qquad \forall i \in I(s'),\ \{s;\ \mathtt{self} : e;\ s'\} \vdash i \boldsymbol{:} m'(i)\end{array}}{s \vdash \mathtt{Adj}\ (e)\ \{s'\}\ \boldsymbol{:} \mathit{Adj}(m, m')}\ \textsc{adj}$$

$$\frac{\begin{array}{c}\vdash s \boldsymbol{:} \mathit{Step} \qquad s \vdash e_1 \boldsymbol{:} \mathit{Adj}(m_1, m_1') \\ s \vdash e_2 \boldsymbol{:} \mathit{Noun}(m_2)/\mathit{Set}(m_2)/\mathit{Term}(m_2) \qquad m_1 \preccurlyeq m_2 \\ \forall i \in (\mathrm{dom}(m_1') \setminus \mathrm{dom}(m_1)) \cap \mathrm{dom}(m_2),\ m_2(i) \mathrel{\bar{\preccurlyeq}} m_1'(i)\end{array}}{s \vdash e_1\ e_2 \boldsymbol{:} \mathit{Noun}(m_1' \uplus m_2)/\mathit{Set}(m_1' \uplus m_2)/\mathit{Term}(m_1' \uplus m_2)}\ \begin{array}{l}\textsc{term}/\\ \textsc{set}/\\ \textsc{noun}\end{array}\ \textsc{-refinement}$$

$$\frac{\begin{array}{c}\vdash s \boldsymbol{:} \mathit{Step} \\ s \vdash e_1 \boldsymbol{:} \mathit{Adj}(m_1, m_1') \qquad s \vdash e_2 \boldsymbol{:} \mathit{Adj}(m_2, m_2') \qquad m_1 \preccurlyeq m_2' \\ \forall i \in (\mathrm{dom}(m_1') \setminus \mathrm{dom}(m_1)) \cap (\mathrm{dom}(m_2') \setminus \mathrm{dom}(m_2)),\ m_2'(i) \mathrel{\bar{\preccurlyeq}} m_1'(i)\end{array}}{s \vdash e_1\ e_2 \boldsymbol{:} \mathit{Adj}(m_2, m_2' \uplus m_1')}\ \textsc{adj-refinement}$$

$$\frac{\vdash \{\overrightarrow{s}\} \boldsymbol{:} \mathit{Step}}{\{\overrightarrow{s};\ \mathtt{self} : \mathtt{term}\} \vdash \mathtt{self} \boldsymbol{:} \mathit{Term}(\emptyset)}\ \textsc{self-noun}$$

$$\frac{\vdash \{\overrightarrow{s}\} \boldsymbol{:} \mathit{Step} \qquad \{\overrightarrow{s}\} \vdash e \boldsymbol{:} \mathit{Noun}(m)}{\{\overrightarrow{s};\ \mathtt{self} : e\} \vdash \mathtt{self} \boldsymbol{:} \mathit{Term}(m)}\ \textsc{self-adj}$$

$$\frac{\begin{array}{c}\vdash \{\overrightarrow{s}\} \boldsymbol{:} \mathit{Step} \\ \{\overrightarrow{s}\} \vdash \mathtt{self} \boldsymbol{:} \mathit{Term}(m) \qquad i \in I(p) \qquad \{\overrightarrow{s};\ p\} \vdash i \boldsymbol{:} t\end{array}}{\{\overrightarrow{s};\ p\} \vdash \mathtt{self} \boldsymbol{:} \mathit{Term}((i, t) \uplus m)}\ \textsc{self-character}$$

$$\frac{\{\overrightarrow{s}\} \vdash \texttt{self} : a \qquad I(s') = \emptyset}{\{\overrightarrow{s}; p\} \vdash \texttt{self} : a}\ \textsc{self-basic}$$

$$\frac{\{\overrightarrow{s}; s_2\} \vdash \texttt{self} : a}{\{\overrightarrow{s}; s_1 \rhd s_2\} \vdash \texttt{self} : a}\ \textsc{self-local-scoping}$$

$$\frac{\{\overrightarrow{s_1}; \overrightarrow{s_2}\} \vdash \texttt{self} : a}{\{\overrightarrow{s_1}; \{\overrightarrow{s_2}\}\} \vdash \texttt{self} : a}\ \textsc{self-block} \qquad \frac{\{\overrightarrow{s}; s'\} \vdash \texttt{self} : a}{\{\overrightarrow{s}; \texttt{label}\ l\ s'\} \vdash \texttt{self} : a}\ \textsc{self-label}$$

$$\frac{\vdash s : \mathit{Step} \qquad l \in L(s)}{s \vdash \texttt{ref}\ l : \mathit{Stat}}\ \textsc{ref}$$

A.2.3 Rules for the phrase level

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad i \notin \mathit{DI}(s) \\ \forall j,k \in \{1 \ldots n\},\ j \neq k \Rightarrow i_j \neq i_k \qquad \forall j \in \{1 \ldots n\},\ s \vdash i_j : () \to a_j \\ s \vdash e : a \qquad \text{if}\ \ i \in dI(s)\ \ \text{then}\ \ s \vdash i : (a_1, \ldots, a_n) \to a\end{array}}{s \vdash i(i_1, \ldots, i_n) := e : \mathit{Def}((a_1, \ldots, a_n) \to a)}\ \textsc{def}$$

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad \text{if}\ \ i \in I(s)\ \ \text{then}\ \ s \vdash i : (a_1, \ldots, a_n) \to a \\ \forall j \in \{1 \ldots n\},\ s \vdash e_j : a_j \qquad s \vdash e : a\end{array}}{s \vdash i(e_1, \ldots, e_n) := e : \mathit{Def}((a_1, \ldots, a_n) \to a)}\ \textsc{def-case}$$

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad s \vdash i : \mathit{Term}(m_1)/\mathit{Set}(m_1)/\mathit{Noun}(m_1) \\ s \vdash e : \mathit{Noun}(m_2) \qquad \forall i' \in \mathrm{dom}(m_1) \cap \mathrm{dom}(m_2),\ m_1(i') \bar{\preccurlyeq} m_2(i')\end{array}}{s \vdash i \ll e : \mathit{Sub}(\mathit{Term}(m_1 \uplus m_2)/\mathit{Noun}(m_1 \uplus m_2)/\mathit{Set}(m_1 \uplus m_2))}\ \textsc{sub-noun}$$

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad s \vdash i : \mathit{Term}(m_1) \\ s \vdash e : \mathit{Adj}(m_2, m_2') \qquad \forall i' \in \mathrm{dom}(m_1) \cap \mathrm{dom}(m_2'),\ m_1(i') \bar{\preccurlyeq} m_2'(i')\end{array}}{s \vdash i \ll e : \mathit{Sub}(\mathit{Term}(m_1 \uplus m_2')/\mathit{Noun}(m_1 \uplus m_2')/\mathit{Set}(m_1 \uplus m_2'))}\ \textsc{sub-adj}$$

A.2.4 Rules for the discourse level

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash p : \mathit{Stat}/\mathit{Dec}(t)/\mathit{Def}(t)/\mathit{Sub}(t)}{s \vdash p : \mathit{Step}}\ \textsc{basic-step}$$

$$\frac{\vdash s_1 : \mathit{Step} \qquad s_1 \vdash s_2 : \mathit{Step} \qquad \{s_1; s_2\} \vdash s_3 : \mathit{Step}}{s_1 \vdash s_2 \rhd s_3 : \mathit{Step}}\ \text{LOCAL-SCOPING}$$

$$\frac{\vdash s_1 : \mathit{Step} \qquad s_1 \vdash \{\vec{s}\} : \mathit{Step} \qquad \{s_1; \{\vec{s}\}\} \vdash s_2 : \mathit{Step}}{s_1 \vdash \{\vec{s}; s_2\} : \mathit{Step}}\ \text{BLOCK}$$

$$\frac{}{\vdash \{\} : \mathit{Step}}\ \text{EMPTY-STEP} \qquad\qquad \frac{\vdash s : \mathit{Step}}{s \vdash \texttt{self} : \texttt{term}/e : \mathit{Step}}\ \text{SELF-MARKER}$$

$$\frac{\vdash s_1 : \mathit{Step} \qquad l \notin L(s_1) \qquad s_1 \vdash s_2 : \mathit{Step}}{s_1 \vdash \texttt{label}\ l\ s_2 : \mathit{Step}}\ \text{LABEL}$$

A.3 TSa's Souring Transformations

Summary of MathLang-TSa's souring transformations as defined in Section 5.2.3.

A.3.1 Reordering

$$T\left[\begin{array}{l}\boxed{^{\text{<position 1>}}T_1}\\ \vdots \\ \boxed{^{\text{<position n>}}T_n}\end{array}\right] \xrightarrow{\mathit{souring}} T\,(T_1, \ldots, T_n)$$

A.3.2 Sharing

$$\boxed{^{<G_1>}T_1}\ \boxed{^{\text{<shared>}}T}\ \boxed{^{<G_2>}T_2} \xrightarrow{\mathit{souring}} \boxed{^{<G_1>}T_1\ T}\ \boxed{^{<G_2>}T\ T_2}$$

A.3.3 Chaining

$$T\begin{pmatrix}\boxed{^{\texttt{<hook>}}T'}\\ \boxed{^{\texttt{<loop>}}}\end{pmatrix} \xrightarrow{\mathit{souring}} T\begin{pmatrix}T'\\ T'\end{pmatrix}$$

A.3.4 Right folding

$$\boxed{\texttt{<fold-right>}\ T_f \begin{bmatrix} b: \boxed{\texttt{<base>}\ T_b} \\ l: \boxed{\texttt{<list>}\ T_1 \dots T_k} \end{bmatrix}} \xrightarrow{\text{souring}}$$

$$T_f \begin{bmatrix} b: T_f \begin{bmatrix} b: T_f \left[\dots T_f \begin{bmatrix} b: T_b \\ l: T_k \end{bmatrix} \dots \right] \\ l: T_2 \end{bmatrix} \\ l: T_1 \end{bmatrix}$$

A.3.5 Left folding

$$\boxed{\texttt{<fold-left>}\ T_f \begin{bmatrix} b: \boxed{\texttt{<base>}\ T_b} \\ l: \boxed{\texttt{<list>}\ T_1 \dots T_k} \end{bmatrix}} \xrightarrow{\text{souring}}$$

$$T_f \begin{bmatrix} b: T_f \begin{bmatrix} b: T_f \left[\dots T_f \begin{bmatrix} b: T_b \\ l: T_1 \end{bmatrix} \dots \right] \\ l: T_{k-1} \end{bmatrix} \\ l: T_k \end{bmatrix}$$

A.3.6 Mapping

$$\boxed{\texttt{<map>}\ T_f \left(\boxed{\texttt{<list>}\ T_1 \dots T_n} \right)} \xrightarrow{\text{souring}} T_f(T_1) \dots T_f(T_n)$$

A.4 TSa's Souring Rewriting Rules

Summary of MathLang-TSa's souring rewriting rules as defined in Section 6.4.

A.4.1 Reordering

$$\texttt{position}^{i}\langle d_1\rangle, \texttt{position}^{j}\langle d_2\rangle \rightarrow_{\mathit{pos}} \texttt{position}^{j}\langle d_2\rangle, \texttt{position}^{i}\langle d_1\rangle \tag{P1}$$

$$\text{where } j < i$$

$$\ell\langle \texttt{position}^{1}\langle d_1\rangle, d_2\rangle \rightarrow_{\mathit{pos}} \ell\langle d_1, \texttt{cursor}^{1}, d_2\rangle \tag{P2}$$

$$\texttt{cursor}^{i}, \texttt{position}^{i+1}\langle d_1\rangle, d_2 \rightarrow_{\mathit{pos}} d_1, \texttt{cursor}^{i+1}, d_2 \tag{P3}$$

$$\ell\langle d, \texttt{cursor}^{i}\rangle \rightarrow_{\mathit{pos}} \ell\langle d\rangle \tag{P4}$$

A.4.2 Sharing

$$g_1\left\langle d_1\right\rangle, \texttt{shared}\left\langle d\right\rangle, g_2\left\langle d_2\right\rangle \rightarrow_{\mathit{share}} g_1\left\langle d_1, d\right\rangle, g_2\left\langle d, d_2\right\rangle \tag{S1}$$

A.4.3 Chaining

$$\texttt{hook}\left\langle d\right\rangle \rightarrow_{\mathit{chain}} d, \texttt{hook-travel}\left\langle d\right\rangle \tag{C1}$$

$$\texttt{hook-travel}\left\langle d\right\rangle, \texttt{loop} \rightarrow_{\mathit{chain}} d \tag{C2}$$

$$\texttt{hook-travel}\left\langle d_0\right\rangle, d_1, d_2 \rightarrow_{\mathit{chain}} d_1, \texttt{hook-travel}\left\langle d_0\right\rangle, d_2 \quad \text{where } \texttt{loop} \mathrel{\widetilde{\notin}_{\mathcal{G}}} d_1 \tag{C3}$$

$$\texttt{hook-travel}\left\langle d_0\right\rangle, g\left\langle d_1\right\rangle \rightarrow_{\mathit{chain}} g\left\langle \texttt{hook-travel}\left\langle d_0\right\rangle, d_1\right\rangle \tag{C4}$$

$$g\left\langle d_1, \texttt{hook-travel}\left\langle d_0\right\rangle\right\rangle \rightarrow_{\mathit{chain}} g\left\langle d_1\right\rangle, \texttt{hook-travel}\left\langle d_0\right\rangle \tag{C5}$$

A.4.4 Right folding

$$\texttt{fold-right}\langle d_0\rangle \rightarrow_{\mathit{fold}} \texttt{right-travel}\langle d_2\rangle, d_1 \tag{F1}$$

$$\text{where } d_0 \twoheadrightarrow_{\mathit{souring}} d_0',\ \texttt{head}\langle d_0'\rangle \twoheadrightarrow_{\mathit{list}} d_1 \text{ and } \texttt{tail}\langle d_0'\rangle \twoheadrightarrow_{\mathit{list}} d_2$$

$$\texttt{right-travel}\langle d_1, d_2\rangle \rightarrow_{\mathit{fold}} d_1, \texttt{right-travel}\langle d_2\rangle \tag{F2}$$

$$\text{where } \texttt{base} \mathrel{\widetilde{\notin}_{\mathcal{G}}} d_1$$

$$\texttt{right-travel}\langle g\langle d_1\rangle, d_2\rangle \rightarrow_{\mathit{fold}} g\langle \texttt{right-travel}\langle d_1\rangle\rangle, d_2 \tag{F3}$$

$$\text{where } \texttt{base} \mathrel{\widetilde{\in}_{\mathcal{G}}} d_1$$

$$\texttt{right-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} d_2 \quad \text{where } \texttt{list} \sqsubset_{\mathcal{G}} d_1 \tag{F4}$$

$$\texttt{right-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} \texttt{fold-right}\langle d_1\rangle \tag{F5}$$

A.4.5 Left folding

$$\texttt{fold-left}\langle d_0\rangle \rightarrow_{\mathit{fold}} \texttt{left-travel}\langle d_2\rangle, d_1 \tag{F6}$$

where $d_0 \twoheadrightarrow_{\mathit{souring}} d_0'$, $\texttt{daeh}\langle d_0'\rangle \twoheadrightarrow_{\mathit{list}} d_1$ and $\texttt{liat}\langle d_0'\rangle \twoheadrightarrow_{\mathit{list}} d_2$

$$\texttt{left-travel}\langle d_1, d_2\rangle \rightarrow_{\mathit{fold}} d_1, \texttt{left-travel}\langle d_2\rangle \tag{F7}$$

where $\texttt{base} \mathrel{\widetilde{\notin}_{\mathcal{G}}} d_1$

$$\texttt{left-travel}\langle g\langle d_1\rangle, d_2\rangle \rightarrow_{\mathit{fold}} g\langle\texttt{left-travel}\langle d_1\rangle\rangle, d_2 \tag{F8}$$

where $\texttt{base} \mathrel{\widetilde{\in}_{\mathcal{G}}} d_1$

$$\texttt{left-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} d_2 \quad \text{where } \texttt{list} \sqsubset_{\mathcal{G}} d_1 \tag{F9}$$

$$\texttt{left-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} \texttt{fold-left}\langle d_1\rangle \tag{F10}$$

A.4.6 Mapping

$$\texttt{map}\langle d\rangle \rightarrow_{\mathit{map}} [] \quad \text{where } \texttt{list} \sqsubset_{\mathcal{G}} d \tag{M1}$$

$$\texttt{map}\langle d_0\rangle \rightarrow_{\mathit{map}} d_1, \texttt{map}\langle d_2\rangle \quad \text{where} \quad \begin{array}{l} d_0 \twoheadrightarrow_{\mathit{souring}} d_0', \\ \texttt{head}\langle d_0'\rangle \twoheadrightarrow_{\mathit{list}} d_1 \\ \text{and } \texttt{tail}\langle d_0'\rangle \twoheadrightarrow_{\mathit{list}} d_2 \end{array} \tag{M2}$$

Appendix B

CGa XML syntax

B.1 CGa XML syntax Error Elements

Abstract syntax for errors and XML representation.

Namespace URI (usual prefix: `cga-meta`)
`http://www.macs.hw.ac.uk/ultra/mathlang/grammatical-core-meta`

For readability, we show only the opening tag of each XML element; instead we use indentation to express nesting.

- Multiple declarations of an identifier
  ```
  <cga-meta:Multiple_declarations>
    [xml:id of the identifier]
  ```
- Multiple definitions of an identifier
  ```
  <cga-meta:Multiple_definitions>
    [xml:ids of the identifier's components]
  ```
- Post-instance declaration of an identifier
  ```
  <cga-meta:Post_instance_declaration>
    [xml:id of the identifier]
  ```
- Post-definition declaration of an identifier
  ```
  <cga-meta:Post_definition_declaration>
    [xml:id of the identifier]
  ```
- Non matching status for an identifier
  ```
  <cga-meta:Status_mismatch>
    [xml:id of the identifier]
    [expected status]
    [effective status]
  ```

- Non matching types for an identifier
  `<cga-meta:Types_mismatch>`
  [`xml:id` *of the identifier]*
  [expected type]
  [effective type]
- Non matching nouns
  `<cga-meta:Nouns_mismatch>`
  [first noun description]
  [second noun description]
- Nouns overlapping
  `<cga-meta:Nouns_overlap>`
  [first noun description]
  [second noun description]
- Non matching categories for an expression
  `<cga-meta:Categories_mismatch>`
  [`xml:id` *of the expression]*
  [expected category]
  [effective category]
- Non matching type for declaration ([xml:id] of declaration, type of expression)
  `<cga-meta:Declaration_type_mismatch>`
  [`xml:id` *of the declaration]*
  [effective type]
- Instance before declaration or definition
  `<cga-meta:Anticipated_instance>`
  [`xml:id` *of the identifier]*
- Unbound variable
  `<cga-meta:Unbound_identifier>`
  [`xml:id` *of the identifier]*
- Unknown field's name (of complex identifier)
  `<cga-meta:Unknown_field_name>`
  [name's `xml:id`*]*
- Invalid parameter status (variable and status)
  `<cga-meta:Invalid_parameter_status>`
  [identifier's `xml:id`*]*
  [effective status]

- Parameter itself with parameters

  <cga-meta:Parameter_not_parameterless>

  [identifier’s xml:id*]*

- Invalid instance

  <cga-meta:Invalid_instance>

  [instance’s xml:id*]*

- Invalid declaration

  <cga-meta:Invalid_declaration>

  [declaration’s xml:id*]*

- Invalid refinement

  <cga-meta:Invalid_refinement>

  [refinement’s xml:id*]*

- Invalid phrase

  <cga-meta:Invalid_phrase>

  [phrase’s xml:id*]*

- Invalid local scoping

  <cga-meta:Invalid_local>

  [Local scoping’s xml:id*]*

- Invalid step

  <cga-meta:Invalid_step>

  [step’s xml:id*]*

Appendix C

TSa-CGa TeXmacs Plugin Documentation

Using MathLang menu and icon bar
Text & Symbol aspect

After the installation of the MathLang plugin, a MathLang menu should appear as part of the header menu, a MathLang help menu should be append to the main Help menu, and a set of MathLang icons should appear on the User provided bar (View → User provided icons to display this icon bar). Be aware that the **MathLang TeXmacs style** is not automatically loaded by the plugin. Use Document → Add package → mathlang to load the MathLang style for your document. The MathLang menu is divided in five parts.

Annotations. Editing a MathLang's Text & Symbol aspect (TSa) – TeXmacs document differs from a normal TeXmacs editing by the annotation of text elements with their grammatical categories or with syntax souring information.

The grammatical categories of the MathLang Core Grammatical aspect (CGa) are: term, set, noun, adjective, declaration, definition, statement, context and step. Due to rendering issues, each annotation action is provided twice. Once for in-line annotation: MathLang → In-line, and one for paragraph annotation: MathLang → Paragraph. One can either create an empty annotation or annotate a selected piece of text. If no text is selected then, performing an annotation action will create an empty annotation. If a piece of text is selected, performing an annotation action will wrap this text in the annotation. In both cases, one is required to provide an argument for the annotation (see Grammatical and Syntax Souring Annotations for more details about annotation arguments). These annotation actions (for example the term annotation) can be accessed via the menu for in-line annotations (MathLang → In-line → Term) and for paragraph annotations (MathLang → Paragraph → Term). They can also be accessed via key shortcuts for in-line annotations: `C-G T` for a term, `C-G S` for a set, `C-G N` for a noun, `C-G A` for an adjective, `C-G Z` for a declaration, `C-G D` for a definition, `C-G P` for a statement, `C-G C` for a context and `C-G B` for a step, and for paragraph annotations: `C-G P T` for a term, `C-G P S` for a set, `C-G P N` for a noun, `C-G P A` for an adjective, `C-G P Z` for a declaration, `C-G P D` for a definition, `C-G P P` for a statement, `C-G P C` for a context and `C-G P B` for a step. An annotation is rendered as a box with a particular background colour. For example here the word 'hypothesis' is annotated as a statement.

> *[...]* the hypothesis that *[...]*

Here is the colour scheme: term, set, noun, adjective, declaration, definition, statement, context and step (the MathLang icons are coloured following this scheme). Here is the greyscale scheme (for black & white printing purposes): term, set, noun, adjective, declaration, definition, statement, context and step. See Grammatical and Syntax Souring Annotations for more details on the meaning of these annotations.

The syntax souring annotation actions can be accessed via the menu (MathLang → Souring) or via key shortcuts: `C-S P` for a position, `C-S F L` for a left folding, `C-S F R` for a right folding, `C-S M` for a map, `C-S B` for a folding base, `C-S L` for a list, `C-S S` for a share, `C-S C H` for a hook and `C-S C L` for a loop. See Grammatical and Syntax Souring Annotations for more details on the meaning of these souring annotations.

Checking and feedback. The grammatical and souring annotation of pieces of text gives essential clues about the role of each text element in the reasoning expressed in the document. The MathLang plugin for TeXmacs can act as a client for a MathLang-CGa server (see Launching MathLang server to learn more about MathLang server). The uploading action can be accessed via the menu MathLang → Send document or via the key shortcut `C-U`. This action sends the entire document to a MathLang server at `127.0.0.1:9933` (to customize this address and port see below the **Customizing** section).

Warning. It is highly recommended to save the TeXmacs file before using this command. MathLang server is still in development and therefore might hang up unexpectedly which would make TeXmacs to freeze.

After receiving the document, the MathLang-CGa server analyses it and returns a list of errors encountered in the document. In a perfect situation, no errors would be found which should result in an empty menu MathLang → Errors' descriptions. In an unlikely case, the descriptions of the errors found in the document will appear in this menu MathLang → Errors' descriptions. An error description is a tuple of an error identifier and a short phrase describing the error. For example, the following text in the menu is the description of the error `e-3`.

> e-3: Anticipated instance of 'h'

This should correspond to one or more occurrences of the `e-3` error label in the document. The border's width of an erroneous annotation will automatically be increased to 1pt. Here is an example of such occurrence (here, the green box representing a statement named 'h' in the MathLang annotation argument).

> *[...]* the `*e-3*` hypothesis that *[...]*

Note that there exist two kinds of errors. The first one consists of grammatical errors that the MathLang-CGa checker has cought. The labels, corresponding to this kind of error, look like: `e-`n with n being a number. The errors in the second category are identified during the Text & Symbol aspect's transformations which consist roughly of a transformation from an annotated document to a CGa'*abstract syntax* document. Their labels look like `sour-`n where n is a number.

For cleanliness during editing, it is possible to hide the appearance of these error labels. When MathLang → Display Error feedback is unchecked, the errors will not be displayed. In this way, the appearance of the document is less cluttered for further editing and annotation. When the document is sent to the server again, this setting is automatically turned off so that the new set of errors (if any) are displayed.

Views. A set of actions are provided to change the way MathLang plugin should render the annotations.

MathLang → Annotation Borders

When checked, TEXMACS displays the borders of annotations. This does not affect the behaviour of MathLang → Display Error feedback.

MathLang → Colours

When checked, TEXMACS displays the background colors of annotations.

MathLang → Interpretations

When checked, TEXMACS displays the MathLang interpretation of each annotation.

MathLang → Greyscale colour scheme

Changes the displayed background color of annotations. When not checked (the default setting), TEXMACS employs the default MathLang colour scheme. When this menu item is checked, the colour scheme is altered to use only shades of grey for annotation backgrounds. If MathLang → Colours is unchecked, this will change the setting but have no immediate visual impact. Useful for checking the appearance of a document when printed on a black-and-white printer.

Customizing. A set of actions are provided to customize MathLang plugin.

MathLang → Set server address

This action provides a prompt to set the address to which to connect to a MathLang server. The default value is `127.0.0.1`. New address are stored in TEXMACS file.

MathLang → Set server port

This action provides a prompt to set the port to which to connect to a MathLang server. The default value is `9933`. New ports are stored in TEXMACS file.

Help. One can access this MathLang plugin help and documentation via MathLang → Documentation and Help → MathLang help.

Heriot-Watt University, ULTRA group.

`$Id: mathlang-man-server.en.tm 1813 2006-09-20 19:07:29Z mm20 $`

Grammatical and syntax souring annotations with MathLang's Text & Symbol aspect

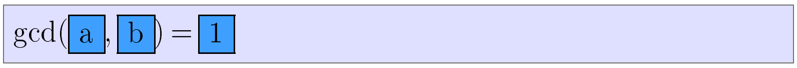
The editing of MathLang using TeXmacs is a straightforward process. It consist in a normal TeXmacs editing with an extra task of annotating pieces of text according to their grammatical categories. The choice of annotation does not require strong knowledge in Mathematics, linguistics, computation, set theory or type theory but mainly require some familiarities with mathematical discourse and to be aware of what the MathLang's Core Grammatical aspect (CGa) and Text & Symbol aspect (TSa) are capable and not capable of analysing. It is important to mention here that MathLang is an ongoing project still under development. In an attempt to clarify the annotation process, we explain here the way one can annotate some common phrasing.

Be aware that **MathLang TeXmacs style** is not automatically loaded by the plugin. Use Document → Add package → mathlang to use MathLang style for the current file.

Expressing facts and constructing formulas. Mathematical facts and formulas are built by composing assertions and symbols. In TSa we offer simple annotations to highlight the role of each piece of text. One might be confused by the reunion of *facts* and *symbols* under a unique consideration, this example attempts to show the reason why. Our example is the following.

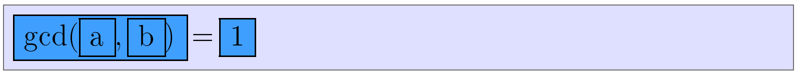
$\gcd(a, b) = 1$

Here we can identify 1, a, b. They are annotated as terms (MathLang → Inline → Term or `C-G T`) in the text with respectively 1, a and b as annotation's argument.



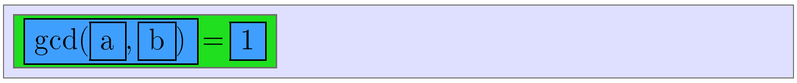
$\gcd(\boxed{a}, \boxed{b}) = \boxed{1}$

We can also identify that $\gcd(a, b)$ is a term itself, build from the application of the gcd function on the terms a and b. $\gcd(a, b)$ is therefore annotated as term with gcd as annotation's argument.



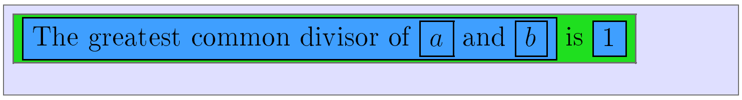
$\boxed{\gcd(\boxed{a}, \boxed{b})} = \boxed{1}$

Similarly we can identify the adjunction of $\gcd(a, b)$ and 1 with the operator $=$. This is a fact and is therefore annotated as statement (MathLang → Inline → Statement or `C-G P`) with = as annotation's argument.



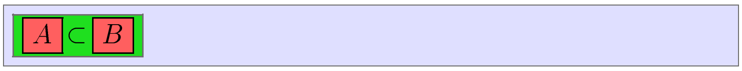
$\boxed{\boxed{\gcd(\boxed{a}, \boxed{b})} = \boxed{1}}$

Here we dealt with a formula but the annotation would have been identical if the formula was expressed in natural language.



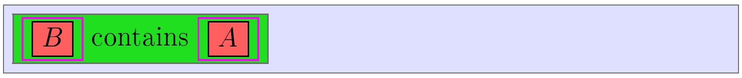
The greatest common divisor of a and b is 1

The annotations presented here highlight the instantiation of an identifier (operator, function, constant or variable). This identifier is the annotation's argument. The result of this instantiation belongs to a specific grammatical category (here it was either terms or statements), the annotation indicates it. If the instantiation requires arguments (this was the case for gcd and =), these should be located under the constructor's annotation. From these annotations, MathLang's TSa interprets this phrase as: (= (gcd a b) 1), using a lisp-like syntax. Here the arguments' order in the texts follows the computerised one. It might not always be the case. Extra annotations can be added to the grammatical annotations to explain the manner to reshuffle the text for interpretation. This is done using syntax souring annotations. For example, one can use the position reordering annotation (MathLang → Souring → Position) to indicate how a sequence of grammatical annotations should be reordered. The argument for this annotation is a number n which is the relative position of the annotation in the computerised interpretation. For example, the phrases "$A \subset B$" and "B contains A" which one would like to interpret as (subset A B), would be annotated as follows.



$A \subset B$

(where A, B and $\subset$ are annotated with arguments A, B and subset respectively)



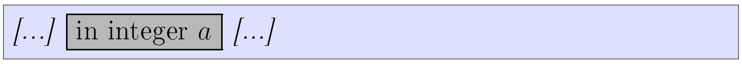
B contains A

(where B, A and "contains" are annotated with arguments 2 then B, 1 then A, and subset respectively)

Declaring variables and notions, defining functions, operators and notions. To annotate that a piece of text is declaring an identifier or defining its meaning, one can use the declaration and definition annotations. Our example is the first mention in a text of a certain integer a.

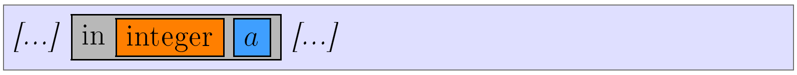
[...] in integer a *[...]*

We first annotate the overall as a declaration (MathLang → In-line → Declaration or `C-G Z`) with the variable's name as annotation's argument.



[...] in integer a *[...]*

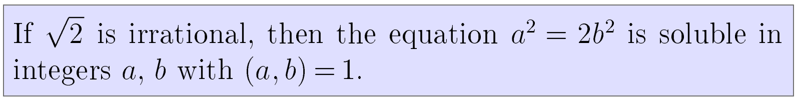
Then we need to annotate the variable itself to indicate its grammatical category. In this example we also annotate "integer" which is the family to which a belongs. Therefore a is annotated as a term (MathLang → In-line → Term or `C-G T`) with annotation's argument a and "integer" is annotated as a noun (MathLang → In-line → Noun or `C-G N`) with annotation's argument integer.



[...] in integer a *[...]*

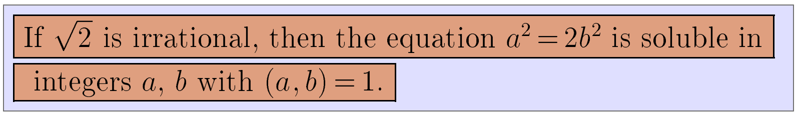
The MathLang interpretation is `a:integer`. Often, declarations come in group or even are hidden. It is required to make them explicit with annotations.

Contexts and sequences of steps. In our previous examples we mainly saw how to compose annotations to express the meaning of a formula or a phrase. We need now to annotate their combinations.



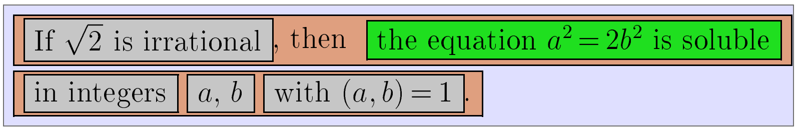
If $\sqrt{2}$ is irrational, then the equation $a^2 = 2b^2$ is soluble in integers a, b with $(a,b) = 1$.

The overall sentence is a step. We annotate it (MathLang → In-line → Step or `C-G B`).



If $\sqrt{2}$ is irrational, then the equation $a^2 = 2b^2$ is soluble in integers a, b with $(a,b) = 1$.

We can identify that several pieces of this sentence compose its context. The assumption "If $\sqrt{2}$ is irrational", the declarations "in integers a, b" and the condition "$(a,b) = 1$" form the context of this sentence, we therefore annotate them as such (MathLang → In-line → Context or `C-G C`). The phrase on the resolvability of the equation is the statement (MathLang → In-line → Statement or `C-G P`) brought by this sentence.



If $\sqrt{2}$ is irrational, then the equation $a^2 = 2b^2$ is soluble in integers a, b with $(a,b) = 1$.

MathLang plugin interprets it by grouping the contextual and non-contextual bits together.

References

[ABFL05] S. Autexier, C. Benzmüller, A. Fiedler, and H. Lesourd. Integrating proof assistants as reasoning and verification tools into a scientific WYSIWIG editor. In *User Interfaces for Theorem Provers (UITP '05) [Workshop]*, Edinburgh, 2005.

[ABFL06] Serge Autexier, Christoph Benzmüller, Armin Fiedler, and Henri Lesourd. Integrating proof assistants as plugins in a scientific editor. [Koh06], pages 309–312.

[AC96] Martìn Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[Act03] The ActiveMath Group. OMDoc JDOM visual editor, 2003. DFKI and Universität des Saarlandes. http://www.activemath.org/projects/OmdocJdom/.

[AF06] Serge Autexier and Armin Fiedler. Textbook proofs meet formal logic — the problem of underspecification and granularity. In MKM '05 [MKM06a].

[AFNW07] Serge Autexier, Armin Fiedler, Thomas Neumann, and Marc Wagner. Supporting user-defined notations when integrating scientific text-editors with proof assistance. In MKM '07 [MKM07]. To appear.

[ALW06] David Aspinall, Christoph Lüth, and Burkhart Wolff. Assisted proof document authoring. In MKM '05 [MKM06a], pages 65–80.

[APSS01] A. Asperti, L. Padovani, C. Sacerdoti Coen, and I. Schena. HELM and the semantic math-web. In TPHOLs '01 [TPH01], pages 59–74.

[AR03] P. Audebaud and L. Rideau. TeXmacs as authoring tool for publication and dissemination of formal developments. In *Proceedings of the*

User Interfaces for Theorem Provers Workshop, UITP 2003, volume 103 of *ENTCS*, pages 27–48, Rome, 2003.

[AS06] Serge Autexier and Claudio Sacerdoti Coen. A formal correspondence between OMDoc with alternative proofs and the $\bar{\lambda}\mu\tilde{\mu}$-calculus. In MKM '06 [MKM06b], pages 67–81.

[Asp00] David Aspinal. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.

[Aut] Automath archive. `http://automath.webhop.net/` Brouwer Institute in Nijmegen and the Formal Methods section of Eindhoven University of Technology.

[Aut03] Serge Autexier. *Hierarchical Contextual Reasoning.* PhD thesis, Computer Science Department, Saarland University, Saarbrücken, Germany, December 2003.

[Aut05] Serge Autexier. The CORE calculus. In Robert Nieuwenhuis, editor, *Proceedings of the 20th International Conference on Automated Deduction (CADE-20)*, volume 3632 of *Lecture Notes in Artificial Intelligence*, Tallinn, Estonia, July 2005. Springer.

[AvLS96] J. Abbott, A. van Leeuwen, and A. Strotmann. Objectives of openmath. Technical Report 12, RIACA, 1996.

[Bar03] Henk Barendregt. *Towards an Interactive Mathematical Proof Mode*, pages 25–36. Volume 28 of Kamareddine [Kam03], November 2003.

[Bar04] Henk Barendregt. Foundations of mathematics from the perspective of computer mathematics. In *Buchberger Festschrift.* 2004. To appear. Accessible at `ftp://ftp.cs.ru.nl/pub/CompMath.Found/buchberger.ps` (last visited 2006–04–24).

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, May 2004.

[BCC$^+$04] S. Buswell, O. Caprotti, D. P. Carlisle, M. C. Dewar, M. Gaëtano, and M. Kohlhase. *The OpenMath Standard.* The OpenMath Society, `http://www.openmath.org`, June 2004. Version 2.0.

[BCF$^+$97] Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Wolf Schaarschmidt, Jörg H. Siekmann, and Volker Sorge. Omega: Towards a mathematical assistant. In *The Fourteenth International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Computer Science*, pages 252–255, Townsville, North Queensland, Australia, July 1997. Springer.

[BCJ$^+$06] B. Buchberger, A. Crăciun, T. Jebelean, L. Kovács, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, pages 470–504, 2006.

[BL92] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. Int'l Conf. Computer Languages*, pages 282–290, 1992.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Bou39] Nicolas Bourbaki. Eléments de mathématique, Since 1939.

[Bou42] Nicolas Bourbaki. *Algèbre*, volume IV (Livre II) of *Eléments de mathématique* [Bou39]. Actualités Scientifiques et Industrielles, no. 934. Hermann & Cie, Paris, 1942.

[Bou54] Nicolas Bourbaki. *Théorie des ensembles*, volume XVII (Livre I) of *Eléments de mathématique* [Bou39]. Actualités Scientifiques et Industrielles, no. 1212. Hermann & Cie, Paris, 1954. Première partie: Les structures fondamentales de l'analyse. Chapitre I: Description de la mathématique formelle. Chapitre II: Théorie des ensembles.

[Bou68] Nicolas Bourbaki. *Theory of Sets*, volume I of *Elements of Mathematics.* Addison-Wesley Publishing Company, 1968. Chapters I and II are translations of [Bou54].

[Bou74] Nicolas Bourbaki. *Algebra*, volume II of *Elements of Mathematics*. Addison-Wesley Publishing Company, 1974. Chapter I is a translation of [Bou42].

[BR02] Grzegorz Bancerek and Piotr Rudnicki. A compendium of continuous lattices in Mizar. *J. Automated Reasoning*, 29(3–4):189–224, 2002.

[Bro06] Chad E. Brown. Verifying and invalidating textbook proofs using Scunak. In MKM '06 [MKM06b], pages 110–123.

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics.* MIT Press, 2002.

[BS03] Y. Baba and M. Suzuki. An annotated corpus and a grammar model of theorem description. In MKM '03 [MKM03], pages 93–104.

[BW05] H. Barendregt and F. Wiedijk. The challenge of computer mathematics. *Royal Society of London Transactions Series A*, 363(1835):2351–2375, October 2005.

[CAD94] The QED manifesto. In Alan Bundy, editor, *The 12th International Conference on Automated Deduction*, volume 814, pages 238–251, Nancy, France, June 1994. Springer.

[Can32] Georg Cantor. *Gesammelte Abhandlungen mathematischen und philosophischen Inhalts.* Springer-Verlag, Berlin, 1932.

[Cas97] Giuseppe Castagna. *Object Oriented Programming: A Unified Foundation.* Birkhäuser, 1997.

[CC99] O. Caprotti and A. M. Cohen. A type system for OpenMath. Technical report, The OpenMath Consortium, February 1999. RIACA, The Netherlands D1.3.2b.

[CD03] David Carlisle and Mike Dewar. Nag library documentation. In MKM '03 [MKM03], pages 56–65.

[CG04] Paul Cairns and Jeremy Gow. Using and parsing the Mizar language. In *Proc. [MKMNET] Mathematical Knowledge Management Symposium* [MKM04a], pages 60–69.

[CG06] Paul A. Cairns and Jeremy Gow. Literate proving: Presenting and documenting formal proofs. In MKM '05 [MKM06a], pages 159–173.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Inform. & Comput.*, 76:95–120, 1988.

[Cho57] Noam Chomsky. *Syntactic Structures.* Mouton, The Hague, 1957.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[Con03] Robert L. Constable. *Recent Results in Type Theory and their Relationship to Automath*, pages 37–48. Volume 28 of Kamareddine [Kam03], November 2003.

[Cor07] Pierre Corbineau. A declarative proof language for the coq proof assistant. Poster presented at BRICKS Midterm Symposium's poster session, March 2007. Accessible at `http://www.cs.ru.nl/~corbineau/ftp/publis/bricks-poster.ps` (last visited 2007–04–24).

[Dav99] James Davenport. A small OpenMath type system. Technical report, The OpenMath Consortium, April 1999. Bath D1.3.2c.

[Dav00] James H. Davenport. A small OpenMath type system. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 34(2):16–21, 2000.

[dB70] N.G. de Bruijn. The mathematical language Automath – its usage and some of its extensions. In L. Laudet, D. Lacombe, and M. Schuetzenberger, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61, Heidelberg, 1970. Springer-Verlag. Reprinted in [NGdV94, A.2].

[dB78] N.G. de Bruijn. Generalizing Automath by means of a lambda-typed lambda calculus. In D.W. Kueker, E.G.K. Lopez-Escobar, and C.H. Smith, editors, *Mathematical Logic and Theoretical Computer Science*, volume 106 of *Lecture Notes in Pure and Applied Mathematics*, pages 71–92. Marcel Dekker, 1978. Reprinted in [NGdV94, B.7].

[dB80] N.G. de Bruijn. A survey of the project Automath. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory*

Logic, Lambda Calculus, and Formalisms, pages 579–606, Orlando, 1980. Academic Press. Reprinted in [NGdV94, A.5].

[dB87] N.G. de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In *Workshop on Programming Logic*, 1987. Reprinted in [NGdV94, F.3].

[dB91a] N. G. de Bruijn. Checking mathematics with computer assistance. *Notices of the American Mathematical Society*, 38(1):8–15, 1991. Available at [Aut].

[dB91b] N.G. de Bruijn. Telescopic mappings in a typed lambda calculus. *Inform. & Comput.*, 91:189–204, 1991.

[dB94a] N.G. de Bruijn. *AUT-QE without type inclusion*, chapter B.4. Volume 133 of *Studies in Logic and the Foundations of Mathematics* [NGdV94], 1994. Presented in 1978.

[dB94b] N.G. de Bruijn. *AUT-SL, a single line version of Automath*, chapter B.2. Volume 133 of *Studies in Logic and the Foundations of Mathematics* [NGdV94], 1994. Presented in 1971.

[dB94c] N.G. de Bruijn. *Some extensions of Automath: the AUT-4 family*, chapter B.3. Volume 133 of *Studies in Logic and the Foundations of Mathematics* [NGdV94], 1994. Presented in 1974.

[dR03] Daniel de Rauglaudre. *Camlp4 – Reference Manual version 3.07.* Institut National de Recherche en Informatique et Automatique, Rocquencourt, France, September 2003. Available at `http://caml.inria.fr/pub/docs/manual-camlp4/` (last visited 2007–04–20).

[Far05] Patrick Farrell. *Grammatical Relations.* Oxford Surveys in Syntax and Morphology. Oxford Linguistics, 2005.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.

[Fre14] Gottlob Frege. Logic in mathematics. Manusrcipt published in [HKK79, pp.203–250], 1914.

[Gal02] Joseph A. Gallian. *Contemporary Abstract Algebra.* Houghton Mifflin Company, 5th edition, 2002.

[Gel04] G. Geleijnse. Comparing two user-friendly formal languages for mathematics: Weak Type Theory an mizar. Master's thesis, Technische Universiteit Eindhoven, May 2004.

[GHK+80] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M .W. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, Berlin Heidelberg New York, 1980.

[Gon] Georges Gonthier. A computer-checked proof of the four colour theorem. Available at `http://research.microsoft.com/~gonthier/4colproof.pdf`.

[GP03] G. Goguadze and A. Gonzalez Palomo. Adapting mainstream editors for semantic authoring of mathematics, November 2003. Presented at the Mathematical Knowledge Management Symposium, Heriot-Watt University, Edinbourgh, Scotland.

[GW03] M. Giero and F. Wiedijk. MMode, a Mizar mode for the proof assistant Coq. Technical Report NIII-R0333, University of Nijmegen, 2003. Available at `http://www.cs.ru.nl/~freek/mmode/mmode.pdf` (last visited 2007–04–24).

[HA28] David Hilbert and Wilhelm Ackermann. *Grundzüge der theoretischen Logik.* Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen (Band XXVII). Springer-Verlag, Berlin, 1928.

[Har96a] John Harrison. A mizar mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, August 1996. Springer.

[Har96b] John Harrison. Proof style. In Eduardo Giménex and Christine Pausin-Mohring, editors, *Types for Proofs and Programs: International Workshop (TYPES'96)*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172, Aussois, France, 1996. Springer.

[HB34] David Hilbert and Paul Bernays. *Grundlagen der Mathematik*, volume 1. Springer-Verlag, Berlin, 1934.

[HB39] David Hilbert and Paul Bernays. *Grundlagen der Mathematik*, volume 2. Springer-Verlag, Berlin, 1939.

[Hea56] Heath. *The 13 Books of Euclid's Elements.* Dover, 1956.

[Hin97] J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1997.

[HK68] Seymour Hayden and John F. Kenninson. *Zermelo Fraenkel Set Theory.* Charles E. Merrill Publishing Co., Columbus, Ohio, 1968.

[HKK79] Hans Hermes, Friedrich Kambartel, and Friedrich Kaulbach, editors. *Gottlob Frege, Posthumous Writings.* Basil Blackwell, Oxford, 1979.

[HW80] Godfrey Harold Hardy and Edward Maitland Wright. *An introduction to the Theory of Numbers.* Oxford University Press, 5th edition, April 1980.

[HW04] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Programming*, 50:189–224, 2004. Special issue of best papers from ESOP '03.

[Ifr94] Georges Ifrah. *Histoire universelle des chiffres.* Robert Laffont, 1994.

[Ifr99] Georges Ifrah. *The Universal History of Numbers. From Prehistory to the Invention of the Computer.* John Wiley & Sons, New York, November 1999. Translation of [Ifr94] by D. Bellos, E.F. Harding, S. Wood and I. Monk.

[JN04] G. Jojgov and Rob Nederpelt. A path to faithful formalizations of mathematics. In MKM '04 [MKM04b], pages 145–159.

[JNS04] G. Jojgov, Rob Nederpelt, and M. Scheffer. Faithfully reflecting the structure of informal mathematical proofs into formal type theories. In *Proc. [MKMNET] Mathematical Knowledge Management Symposium* [MKM04a], pages 102–117.

[Joj06] Gueorgui I. Jojgov. Translating a fragment of Weak Type Theory into Type Theory with open terms. In MKM '05 [MKM06a].

[Kam03] Fairouz Kamareddine, editor. *Thirty Five Years of Automating Mathematics*, volume 28 of *Kluwer Applied Logic series*. Kluwer Academic Publishers, November 2003.

[KF01] Michael Kohlhase and Andreas Franke. MBase: Representing knowledge and context for the integration of mathematical software systems. *J. Symbolic Comput.*, 32(4):365–402, 2001.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer.

[KLMW07] Fairouz Kamareddine, Robert Lamar, Manuel Maarek, and J. B. Wells. Restoring natural language as a computerised mathematics input method. In MKM '07 [MKM07]. To appear.

[KLN03] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *De Bruijn Automath and Pure Type Systems*, pages 71–123. Volume 28 of Kamareddine [Kam03], November 2003.

[KLN04] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory From its Origins Until Today*, volume 29 of *Kluwer Applied Logic Series*. Kluwer Academic Publishers, May 2004.

[KMRW07a] Fairouz Kamareddine, Manuel Maarek, Krzysztof Retel, and J. B. Wells. Gradual computerisation/formalisation of mathematical texts into Mizar. In preparation, available at `http://www.macs.hw.ac.uk/~retel/`, 2007.

[KMRW07b] Fairouz Kamareddine, Manuel Maarek, Krzysztof Retel, and J. B. Wells. Narrative structure of mathematical texts. In MKM '07 [MKM07]. To appear.

[KMW04a] Fairouz Kamareddine, Manuel Maarek, and J. B. Wells. Flexible encoding of mathematics on the computer. In MKM '04 [MKM04b], pages 160–174.

[KMW04b] Fairouz Kamareddine, Manuel Maarek, and J. B. Wells. Mathlang: Experience-driven development of a new mathematical language. In *Proc. [MKMNET] Mathematical Knowledge Management Symposium* [MKM04a], pages 138–160.

[KMW06] Fairouz Kamareddine, Manuel Maarek, and J. B. Wells. Toward an object-oriented structure for mathematical text. In MKM '05 [MKM06a], pages 217–233.

[KN02] Fairouz Kamareddine and Rob Nederpelt. *Logical Reasoning.* Cornerstones Series. Macmillan Press, 2002.

[KN04] Fairouz Kamareddine and Rob Nederpelt. A refinement of de Bruijn's formal language of mathematics. *J. Logic Lang. Inform.*, 13(3):287–340, 2004.

[Knu84a] Donald E. Knuth. *The TEXbook.* Addison-Wesley, Reading, Massachusetts, 1984.

[Knu84b] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Koh04] Michael Kohlhase. Semantic markup for TEX/LATEX. Mathematical User-Interfaces Workshop, 2004.

[Koh06] M. Kohlhase. *An open markup format for mathematical documents, OMDoc (Version 1.2)*, volume 4180 of *Lecture Notes in Artificial Intelligence.* Springer, 2006.

[Kor05] Artur Korniłowicz. Jordan curve theorem. *Formalized Mathematics*, 13(4):481–491, 2005. Identifier JORDAN in [MML].

[KSSS06] Toshihiro Kanahori, Alan Sexton, Volker Sorge, and Masakazu Suzuki. Capturing abstract matrices from paper. In MKM '06 [MKM06b], pages 124–138.

[KW00] Fairouz Kamareddine and J. B. Wells. Formath. A research proposal to UK funding body, 2000.

[KW01] Fairouz Kamareddine and J. B. Wells. MathLang: A new language for mathematics, logic, and proof checking. A research proposal to UK funding body, 2001.

[Lam94] Leslie Lamport. *LaTeX: A document preparation system. User's guide and reference manual.* Addison Wesley Professional, 2nd edition, June 1994.

[Lam95] Leslie Lamport. How to write a proof. *The American Mathematical Monthly*, 102(7):600–608, August–September 1995.

[Lan30] Edmund Landau. *Grundlagen der Analysis.* Chelsea, 1930.

[Lan51] Edmund Landau. *Foundations of Analysis.* Chelsea, 1951. Translation of [Lan30] by F. Steinhardt.

[Lan71] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer, Berlin, 1971.

[Lau85] Roger Laufer, editor. *La Notion de paragraphe.* Éditions du Centre National de la Recherche Scientifique, Paris, 1985.

[Ler83] Uri Leron. Structuring mathematical proofs. *The American Mathematical Monthly*, 90(3):174–185, March 1983.

[Log06] LogiCal Project, INRIA, Rocquencourt, France. *The Coq Proof Assistant Reference Manual – Version 8.1*, 2006. Available at `ftp://ftp.inria.fr/INRIA/coq/V8.1/doc/`.

[Maa85] Marcel Maarek. *L'impossible coupure en mathématique*, pages 105–107. In Laufer [Lau85], 1985.

[Maa02] Manuel Maarek. Conception d'une librairie OMDoc pour FoC. Technical report, Universit'e Pierre et Marie Curie Paris VI, September 2002. Rapport de DEA.

[Maa03] Manuel Maarek. First year PhD report. Technical report, Heriot-Watt University, August 2003.

[MBG$^+$03] E. Melis, J. Büdenbender, G. Goguadze, P. Libbrecht, M. Pollet, and C. Ullrich. Knowledge representation and management in activemath. *Annals of Mathematics and Artificial Intelligence Special Issue on Management of Mathematical Knowledge*, 38(1–3):47–64, 2003.

[MG06] Lionel Elie Mamane and Herman Geuvers. A document-oriented Coq plugin for TeXmacs. In *Mathematical User-Interfaces Workshop 2006 [Workshop]*, Workingham, 2006.

[MKM01] *Mathematical Knowledge Management, 1st Int'l Workshop, Proceedings*. RISC-Linz, Research Institute of Symbolic Computation, Johannes Kepler University, Linz, Austria, 2001. Electronic version available at `http://www.risc.uni-linz.ac.at/about/conferences/MKM2001/Proceedings/` (last visited 2007–04–21). Appeared as special issue of Annals of Mathematics and Artificial Intelligence, vol. 38, no 1–3, May 2003.

[MKM03] *Mathematical Knowledge Management, 2nd Int'l Conf., Proceedings*, volume 2594 of *Lecture Notes in Computer Science*. Springer, 2003.

[MKM04a] *Proc. [MKMNET] Mathematical Knowledge Management Symposium*, volume 93 of *ENTCS*, Edinburgh, UK (2003-11-25/---29), February 2004. Elsevier Science.

[MKM04b] *Mathematical Knowledge Management, 3rd Int'l Conf., Proceedings*, volume 3119 of *Lecture Notes in Computer Science*. Springer, 2004.

[MKM06a] *Mathematical Knowledge Management, 4th Int'l Conf., Proceedings*, volume 3863 of *Lecture Notes in Artificial Intelligence*. Springer, 2006.

[MKM06b] *Mathematical Knowledge Management, 5th Int'l Conf., Proceedings*, volume 4108 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006.

[MKM07] *Mathematical Knowledge Management, 6th Int'l Conf., Proceedings*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007.

[MLUM06] Shahid Manzoor, Paul Libbrecht, Carsten Ullrich, and Erica Melis. Authoring presentation for OpenMath. In MKM '05 [MKM06a], pages 33–48.

[MML] Mizar Mathematical Library. Published and updated via the Journal of Formalized Mathematics (accessible at `http://mizar.org/JFM`). Accessible at `http://mizar.org/library/`.

[MP03] Manuel Maarek and Virgile Prevosto. FoCDoc: The documentation system of FoC. In *11th Calculemus Symposium*, September 2003.

[Ned02] R. Nederpelt. Weak type theory: A formal language for mathematics. Technical report, Eindhoven University of Technology, May 2002.

[NGdV94] Rob Nederpelt, J. H. Geuvers, and Roel C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1994.

[NK01] R. P. Nederpelt and F. Kamareddine. Formalising the natural language of mathematics: A mathematical vernacular. In *4th Int'l Tbilisi Symp. Language, Logic & Computation*, 2001.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[NS06] Koji Nakagawa and Masakazu Suzuki. Mathematical knowledge browser with automatic hyperlink detection. In MKM '05 [MKM06a].

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *The 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga Springs, NY, USA, June 1992. Springer.

[Pad03] L. Padovani. On the roles of LaTeX and MathML in encoding and processing mathematical expressions. In MKM '03 [MKM03], pages 66–79.

[PD02] Virgile Prevosto and Damien Doligez. Algorithms and proof inheritance in the FoC language. *J. Automated Reasoning*, 29(3–4):337–363, December 2002.

[PDH02] Virgile Prevosto, Damien Doligez, and Thérèse Hardin. Algebraic structures and dependent records. In *Theorem Proving in Higher Order Logics: 15th Int'l Conf., Proceedings*, volume 2410 of *Lecture Notes in Computer Science*, pages 298–313. Springer, 2002.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pou06] Nicolas Pouillard. Rénovation de Camlp4, un prṕrocesseur pretty-printer pour Caml. Technical report, Équipe Gallium, INRIA, Rocquencourt, France, 2006. Available at `http://gallium.inria.fr/~pouillar/pub/camlp4/Rapport_Nicolas_Pouillard_CSI.pdf` (last visited 2007–04–29).

[Pre03] Virgile Prevosto. *Conception and implementation of the FoC language for the development of certified softwares.* PhD thesis, Université Paris 6, September 2003.

[Pro] The Course Capsules Project. OMDoc mode, version 0.8, an Emacs mode for OMDoc documents. Version 0.8, released on 26–09–2002. `http://aiki.ccaps.cs.cmu.edu/DownloadIndex.html#omdocmode`.

[PZ06] Luca Padovani and Stefano Zacchiroli. From notation to semantics: There and back again. In MKM '06 [MKM06b], pages 194–207.

[Ram26] F.P. Ramsey. The foundations of mathematics. In *Proceedings of the London Mathematical Society*, volume 25 of *2nd series*, pages 338–384, 1926.

[Ret05] K. Retel. First year PhD report: Towards semi-automatic bridging of existing tools for formalising mathematics. September 2005.

[RRSS06] Amar Raja, Matthew Rayner, Alan Sexton, and Volker Sorge. Towards a parser for mathematical formula recognition. In MKM '06 [MKM06b], pages 139–151.

[RT99] Piotr Rudnicki and Andrzej Trybulec. On equivalents of well-foundedness. an experiment in MIZAR. *Journal of Automated Reasoning*, 23(3–4):197–234, 1999.

[Rud92] P. Rudnicki. An overview of the Mizar project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[Sch03] M. Scheffer. Formalizing mathematics using Weak Type Theory. Master's thesis, Technische Universiteit Eindhoven, September 2003.

[Ste96] Mark Steedman. *Surface Structure and Interpretation.* Number 30 in Linguistic Inquiry Monograph. MIT Press, Cambridge, MA, April 1996.

[Str03] Andreas Strotmann. *Content Markup Language Design Principles.* PhD thesis, Computer Science Department, The Florida State University, 2003. Published as Technical Report TR-030702, Computer Science Department, The Florida State University, Tallahasee, FL, July 2003.

[Str04] Andreas Strotmann. The categorial type of OpenMath objects. In MKM '04 [MKM04b], pages 378–392.

[Tar39] Alfred Tarski. On well-ordered subsets of any set. *Fundamenta Mathematicae*, 32:176–183, 1939.

[TPH01] *Theorem Proving in Higher Order Logics: 14th Int'l Conf., Proceedings*, volume 2152 of *Lecture Notes in Computer Science.* Springer, 2001.

[Try80] A. Trybulec. The Mizar logic information language. *Studies in Logic*, 1, 1980. Bialystok.

[Try89] Andrzej Trybulec. Tarski Grothendieck set theory. *Journal of Formalized Mathematics*, Axiomatics, 1989. Identifier TARSKI in [MML] `http://mizar.org/JFM/Axiomatics/tarski.html`.

[vBJ77a] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system.* PhD thesis, Eindhoven, 1977. Partially reprinted in [NGdV94, B.5,D.2,D.3,D.5,E.2].

[vBJ77b] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system.* PhD thesis, Eindhoven, 1977.

[vBJ94] L.S. van Benthem Jutting. *Description of AUT-68*, chapter B.1. Volume 133 of *Studies in Logic and the Foundations of Mathematics* [NGdV94], 1994.

[vD80] D.T. van Daalen. *The Language Theory of Automath.* PhD thesis, Eindhoven University of Technology, 1980. Partially reprinted in [NGdV94, A.6,B.6,C.5].

[vdH01] Joris van der Hoeven. GNU TeXmacs: a free, structured, wysiwyg and technical text editor. *Cahiers GUTenberg*, (39–40):39–50, 2001. Actes de GUT 2001 : ń Le document au XXIe siècle ż.

[vdH04] Joris van der Hoeven. GNU TeXmacs. *SIGSAM Bulletin*, 38(1):24–25, 2004.

[vT06] Paul van Tilburg. Exploring the core of mathlang. Technical report, Heriot-Watt University, 2006.

[W3C03] W3C. Mathematical markup language (MathML) version 2.0. W3C Recommendation, October 2003. `http://www.w3.org/TR/MathML/`.

[W3C04] W3C. OWL web ontology language. W3C Recommendation, February 2004. `http://www.w3.org/TR/owl-ref/`.

[W3C06] W3C. XHTML-Print. W3C Recommendation, September 2006. `http://www.w3.org/TR/xhtml-print/`.

[WAB06] Marc Wagner, Serge Autexier, and Christoph Benzmüller. PLATO: A mediator between text-editors and proof assistance systems. In Christoph Benzmüller Serge Autexier, editor, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, ENTCS. Elsevier, August 2006.

[WAL05] Daniel Winterstein, David Aspinall, and Christoph Lüth. Proof General / Eclipse: A generic interface for interactive proof. In *User Interfaces for Theorem Provers (UITP '05) [Workshop]*, Edinburgh, 2005.

[Wen99] M. Wenzel. Isar – a generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics: 12th Int'l Conf., Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 1999.

[Wen02] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.

[Wie01] F. Wiedijk. Mizar light for HOL light. In TPHOLs '01 [TPH01], pages 378–394.

[Wie03] F. Wiedijk. Comparing mathematical provers. In MKM '03 [MKM03], pages 188–202.

[Wie04] F. Wiedijk. Formal proof sketches. In *Proceedings of TYPES'03*, volume 3085 of *LNCS*, pages 378–393. Springer-Verlag, December 2004.

[Wie06] F. Wiedijk, editor. *The Seventeen Provers of the World, foreword by Dana S. Scott*, volume 3600 of *LNCS*. Springer Berlin, Heidelberg, 2006.

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

[Worb] World Wide Web Consortium (W3C), http://www.w3.org/XML/. *Extensible Markup Language (XML).*

[WR13] Alfred North Whitehead and Bertrand Russel. *Principia Mathematica.* Cambridge University Press, Cambridge, 1910–1913.

[WW02] Markus Wenzel and Freef Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. *Journal of Automated Reasoning*, 29:389–411, 2002.

[Zin04] Claus Zinn. *Understanding Informal Mathematical Discourse*. PhD thesis, Arbeitsberichte des Instituts fuer Informatik, Friedrich-Alexander Universitaet Erlangen-Nuernberg, 2004. Band 37, Nr. 4.

[ZK02] Jürgen Zimmer and Michael Kohlhase. System description: The MathWeb Software Bus for distributed mathematical reasoning. In Andrei Voronkov, editor, *CADE*, volume 2392 of *Lecture Notes in Computer Science*, pages 139–143. Springer, 2002.

Index