A Partial Translation Path from MathLang to Isabelle

BY
Robert Lamar

SUBMITTED FOR THE DEGREE OF
Doctor of Philosophy

Heriot-Watt University
SCHOOL OF MATHEMATICAL AND COMPUTER SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

May 2011

Abstract

This dissertation describes certain developments in computer techniques for managing mathematical knowledge. Computers currently assist mathematicians in presenting and archiving mathematics, as well as performing calculation and verification tasks. MathLang is a framework for computerising mathematical documents which features new approaches to these issues. In this dissertation, several extensions to MathLang are described: a system and notation for annotating text; improved methods for annotating complex mathematical expressions; and a method for creating rules to translate document annotations. A typical MathLang work flow for document annotation and computerisation is demonstrated, showing how writing style can complicate the annotation process and how these may be resolved. This workflow is compared with the standard process for producing formal computer theories in a computer proof assistant (Isabelle is the system we choose). The rules for translation are further discussed as a way of producing text in the syntax of Isabelle (without a deep knowledge of the system), with possible use cases of providing a text which can be used either as an aid to learning Isabelle, or as a skeleton framework to be used as a starting point for a formal document.

For my family
both present and future

Soli Deo Gloria

Contents

Chapter 1

Introduction

In the day-to-day work of the mathematician, computers provide some assistance. Number crunching has long been the domain of computer systems, and more recently great strides have been made in the areas of document preparation and assistance with algebra. However, there is a host of other applications in which computers could conceivably assist mathematicians in their work. This dissertation describes certain efforts towards making these applications more accessible to the average working mathematician. In particular, the work described herein is focused on several problems: computerising the semantics of a document while accommodating various writing styles (described in Section 4.3); techniques to accommodate computer-unfriendly expressions (Chapter 5); mapping these computerised semantics to the language of a target computer system which provides facilities which may be of interest to mathematicians (Chapter 8); and tools to help make sure that such mappings are complete (Chapter 7). Before getting into the details of the current work, we first describe its place in the larger project (Chapter 4) and the field at large (Section 1.2). We also provide an overview (in Chapter 2) of the particular technologies and tools which are used to support the work at hand and a thorough working example which shows all methods described (Chapter 9). Chapter contents are discussed in more detail in Section 1.4.

The work described herein falls in the field of mathematical knowl-

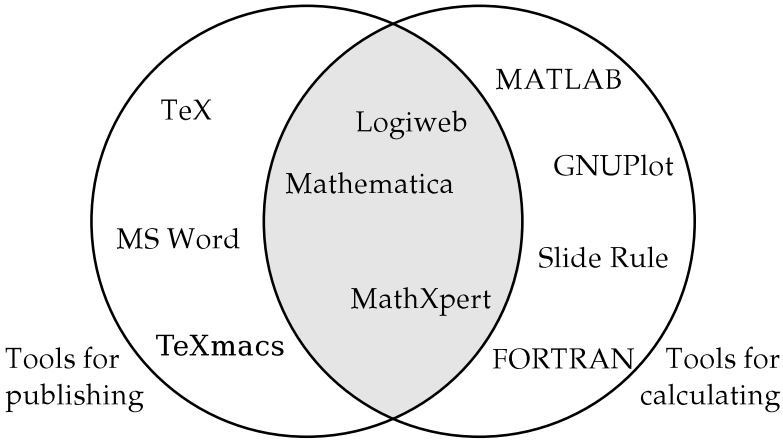


Figure 1.1 Popular tools placed in categories of utility.

edge management (MKM). Generally speaking, MKM is concerned with using computers to better process, control, analyse, and index the fruit of the mathematician's labour. There are a number of people working in this field, tackling this vast, general problem from a number of angles. Approaches include schemes and document formats for storing data, systems for testing the veracity of documents to varying levels of rigour, languages to express mathematics in a highly formal expression, and processors to take formalised mathematics and restore it to human-friendly verbiage. There are grammars for parsing formal out of natural language, syntax for making formal language feel natural, and file formats to permit formal and natural language to coexist side-by-side.

Prior to any detailed discussions of our work, we will spend a few pages to provide some motivation for the problems we work on, an overview of related projects, and a summary of our solutions to the problem.

1.1 Automation

In their work, mathematicians generate a great deal of knowledge each year. During 2010, for instance, the mathematical archive at arXiv.org

grew by roughly 1,564 voluntarily-submitted papers each month.[1] For centuries, this kind of knowledge has been organised into documents, reviewed by peers, published in journals, indexed in countless ways, and pored over by other mathematicians. Mathematicians want their work to be read and used. They need access to the work of others in order to make progress. This knowledge is managed and archived by people like publishers and librarians.

Mathematicians have benefited from technology in many ways, both in developing their ideas and also in the dissemination of the knowledge they produce. Over time, technologies which have helped mathematicians develop their ideas include the abacus, slide rule, pocket calculator and computer algebra system. Technologies which have helped them disseminate ideas include the printing press, word processor, and World Wide Web. It is only recently that these two categories began to overlap, as shown in Figure 1.1. The intersection is currently a very small collection when compared to the tools that lie in one camp or another. It includes mostly tools which lie primarily in one camp but incorporate facilities expected in the other. Examples include computer algebra systems which provide typesetting and publishing features, and teaching systems which allow students to perform calculations in the middle of an interactive textbook.

In the traditional model for mathematicians' work, one of the primary problems is in the cycle of developing, reviewing, sharing, and gleaning feedback. The environment in which the mathematician develops ideas is not the same as the one in which the ideas are presented. Thus, there is some risk that the ideas will not transfer faithfully from one form to another. While this is acceptable if the development and checking are being performed entirely by human beings, the risk of disparity increases steadily as more and more assistance is being provided by computers in the development and verification of mathematical results.

[1] Statistics for 2010 are available at `http://arxiv.org/year/math/10`.

1.2 Related Work

Since the advent of computer-aided proof in the 1960s, mathematicians and computer scientists have been seeking effective ways to encode mathematical concepts in languages of varying structure. Some theorem provers are highly rigid and distant from natural language, while others such as Mizar and Isar have a syntax similar to the mathematician's writing style. Each prover has its proponents and favoured applications, but they are all stark and restrictive when compared with the fluidity of natural language. None currently has an infrastructure to provide a direct mapping from a typical natural language mathematical text to its own language but they all have methods to offer natural language integration. We group these methods into four categories.

1.2.1 Proof code with embedded natural language

Systems such as Isabelle [42] and Coq [34] take an approach that allows formalised content to exist in parallel with natural language text. In formal proof languages such as these, there are facilities to incorporate natural language alongside formal definitions and proofs. Natural language text parts are treated as commentary in a literate proof document and are excluded from verification. This method uses *structured comments*, akin to programming languages, for generating documentation out of programming code. In a similar fashion, recent developments of scientific word processors have permitted plugin-interfacing with theorem provers such as Coq and Ωmega [2, 3, 36]. These plugins allow WYSIWYG editing of documents, where the primary document is edited according to the author's taste, but islands of formal content are inserted in special environments. A menu command tells the external program to verify the special content, effectively ignoring the non-formal text which is provided for the benefit of human readers.

1.2.2 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 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 [48] and Isar [53], but more recently some calculi [4, 7] were developed pursuing the idea of a formal representation for pseudo-natural language.

1.2.3 Semantic Web data model

While some projects develop domain-specific languages for encoding mathematics in a machine-friendly way, others are augmenting the existing systems for human-friendly documents so as to enrich the semantics of the document. The goal is to create a simngle document which can be used by human and computer alike. STEX [27, 29] is one project which provides new macros for LATEX, providing semantic information in addition to the existing formatting information. This provides a semantically-rich system for paper-publishing software that is already in wide use. For publishing on the web, OMDoc [28] provides a scheme for XML markup to describe the structure of a document, interacting with MathML [] and OpenMath [] to completely describe the semantics and appropriate presentation of mathematical documents. These systems provide assistance both for humans reading documents, and computer systems passing documents back and forth.

1.2.4 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 [1] has this capability. Furthermore, there are systems under development [44] which provide the ability to personalise the natural language generated. This is significant because automatically generated text can be

rigid and repetitive, making it difficult for a human reader to remain engaged with the text.

1.2.5 Relationships between the systems

From this categorisation it appears that the primary input for a theorem prover is generally a formal language and that the natural language of a theorem prover's document is a formalisation side effect. In case 1.2.2 the document is written in an altered and restricted natural language while in case 1.2.4 the generated (natural) text is only available *after* providing the input through a significantly restricted language. These pseudo-natural languages are by no means the only legitimate representation of mathematics. Recent work – not pertaining to any particular system – has explored the more general issue of comparing various formal representations [25], demonstrating the importance of flexibility in establishing formal models and providing concrete examples such as the formalisation of matrices [45].

In the context of the MathLang project, this work provides tools which produce output which is closer to existing formal systems than ever before. Specifically, it uses the facilities provided by [18, 22] to translate the bulk of a text to the syntax of the proof assistant Isabelle. This is in parallel with [20], which was translating the same kind of document to Mizar [48]. However, in that work the focus was on identifying relevant theories from the Mizar Mathematical Library to include in the environment of a new Mizar document, more so than translating the main text of a MathLang document to Mizar.

In the larger field, there are a number of projects which are attempting to bridge the gap between human-friendly mathematical texts and easily processed and verified computer documents. Most focus primarily on one side or another. On the formal end of the spectrum are projects like Mizar [48], Theorema [8] and Isar [42, 53]. These three computer proof systems are designed with syntax that is constructed to be similar to the way that mathematicians write in natural English. A similar approach is

being taken in the work of Muhammad Humayoun and Christophe Raffalli on MathNat [16, 17], in which they try to express both mathematical proofs and natural language. In the cases of Mizar and Isar, the language is *like* natural English, but does not provide the author with much flexibility. MathNat and Theorema allow much greater flexibility – in MathNat's case due to its incorporation of GF [46] – but still force the user to employ a controlled language, which may often be a subset of what an author would normally employ. MathLang endeavours to accommodate any writing style through the use of flexible annotation, accommodating documents that were never intended to be computerised, such as Euclid's *Elements* [15] and Landau's *Grundlagen der Analysis* [32].

On the other side of the natural–formal gap, Aarne Ranta's system GF [46] has a flexible system for defining grammars, and provides an API for interfacing with other programs, but is not, itself, designed to process documents to further formal states. We wish to process such documents in other interesting ways.

Finally, there are systems such as Isabelle [42] and Coq [34], which are systems for computer proof, along with Logiweb [13], which is a system for document processing that interfaces with arbitrary systems. Each of these allow natural language text to be interleaved with formal expressions in a kind of literate proof document in the manner of CWEB [26]. However, care during revision is necessary to ensure that natural language and formalism remain consistent.

1.2.6 Integrated Systems

As development of these various projects improves, there is a growing interest in integrating various document services. In order to benefit from the services of multiple systems, authors sometimes need to create documents in various forms, each form providing some benefit. Other systems (such as Logiweb) permit the user to create single documents which have different information formats embedded in parallel. Thus, the different parts are sent to different systems as appropriate. However, it is desire-

able that authors derive benefit from more than one system while creating only one input document. The system Plato [52] is designed for that purpose. The authors of that system have created an interface using the TEXMACS scientific text editor which allows the user to create a uniform, sematically-rich, human-readable document which may be processed by multiple systems.

One focus of Plato is the development of change-oriented elements in the architecture of the system. This allows the user to make changes in the middle of a document which may be sent to a service, and the service then processes only the parts of the document which have changed, preserving and re-using the context and previous results without needing to process the entire document again. For large documents, this provides a tangible benefit in reducing processing time.

1.2.7 XML Applications for Structuring Data

One class of projects which are related to MathLang consists of systems which describe mathematics in XML. Three such systems are MathML, OpenMath, and OMDoc. As described below, these systems are complimentary to one another.

MathML

MathML is a system designed for communicating mathematics on the World Wide Web [51]. It is meant to permit encoding of K–14 mathematics, meaning any mathematical content taught in standard courses in primary and secondary schools, and universities. This is done by specifying, in the W3C standard for MathML, all the operators and other expression constituents which may be used.

MathML has two components: content and presentation markup. The latter is primarily focused on how different parts of a formula are composed in boxes and glued together in rendering. The former is concerned with semantics, providing a level of abstraction that allows authors and

users to choose how the mathematics will eventually be rendered. The primary advantage to presentation markup is that it is easier for browsers to render efficiently. Thus, in practice, documents are often written in content MathML and then converted to presentation MathML using style sheets.

In MathML, the mathematical expression $\frac{3}{x+2}$ would be stored as follows[2] in MathML presentation markup:

```
<math>
   <mfrac>
      <mn>3</mn>
      <mfenced>
         <mi>x</mi><mo>+</mo><mn>2</mn>
```

The same fraction, similarly, would be stored in this way when using MathML content markup:

```
<math>
   <apply>
      <divide/>
      <cn>3</cn>
      <apply>
         <plus/><ci>x</ci><cn>2</cn>
```

According to the W3C's recommendation for MathML 2.0 [51], the semantics of such a content MathML snippet are not explicit without additional annotation. The following is an example provided by the recommendation to illustrate possible annotations to provide semantics that would be meaningful to various systems:

```
<semantics>
  <apply>
    <divide/>
    <cn>123</cn>
    <cn>456</cn>
  <annotation encoding="Mathematica">
    N[123/456, 39]
  <annotation encoding="TeX">
    $0.269736842105263157894736842105263157894\ldots$
  <annotation encoding="Maple">
    evalf(123/456, 39);
```

[2]For elements spanning lines, children are indented and closing tags are hidden.

```
<annotation-xml encoding="MathML-Presentation">
  <mrow>
    <mn> 0.269736842105263157894 </mn>
    <mover accent='true'>
      <mn> 736842105263157894 </mn>
      <mo> &OverBar; </mo>
<annotation-xml encoding="OpenMath">
  <OMA xmlns="http://www.openmath.org/OpenMath">
    <OMS cd="arith1" name="divide"/>
    <OMI>123</OMI>
    <OMI>456</OMI>
```

Thus, we see a variety of representations for 'the quotient of 123 and 456,' represented first in content MathML, but also in forms suitable to Mathematica, TEX, Maple, presentation MathML, and OpenMath. These are stored in parallel in the document: one potential drawback of this notation is that only human inspection can ensure that the various annotations that are children of the <semantics> tag each correspond to the original <apply> construct. As described momentarily, the OpenMath encoding maps to an OpenMath content dictionary which standardises concepts such as <divide>. This is one of the strengths of OpenMath which has engendered much cooperation between the MathML working group and the OpenMath community.

OpenMath

The XML application OpenMath [9] was developed initially as a way to standardise communication between computer algebra systems. As a result, the semantics for OpenMath are not built in to the system, but are meant to be defined in a highly-extensible system of files called content dictionaries (CDs). These dictionaries are then used to understand the meaning of the mathematics which might be encoded in an OpenMath document. Consider, for example, this excerpt from the above MathML example:

```
<annotation-xml encoding="OpenMath">
  <OMA xmlns="http://www.openmath.org/OpenMath">
    <OMS cd="arith1" name="divide"/>
    <OMI>123</OMI>
```

```
<OMI>456</OMI>
```

The <OMA> tag indicates that we are applying a symbol to one or more arguments. Symbols are represented with <OMS> elements; the particular symbol in this example is divide, which is defined in the arith1 content dictionary. This divide symbol is being applied to two integers (<OMI>), 123 and 456. This provides an unambiguous representation of the fraction $\frac{123}{456}$. Integers have the usual definition as basic OpenMath objects [9]. On the other hand, divide has a much more detailed specification in the content dictionary arith1:

```
<CDDefinition>
<Name> divide </Name>
<Role>application</Role>
<Description>
  This symbol represents a (binary) division function denoting
  the first argument right-divided by the second, i.e.
  divide(a,b)=a*inverse(b). It is the inverse of the
  multiplication function defined by the symbol times in this CD.
<CMP> whenever not(a=0) then a/a = 1 </CMP>
<FMP>
  <OMOBJ xmlns="http://www.openmath.org/OpenMath" version="2.0"
         cdbase="http://www.openmath.org/cd">
    <OMBIND>
      <OMS cd="quant1" name="forall"/>
      <OMBVAR>
        <OMV name="a"/>
      <OMA>
        <OMS cd="logic1" name="implies"/>
        <OMA>
          <OMS cd="relation1" name="neq"/>
          <OMV name="a"/>
          <OMS cd="alg1" name="zero"/>
        <OMA>
          <OMS cd="relation1" name="eq"/>
          <OMA>
            <OMS cd="arith1" name="divide"/>
            <OMV name="a"/>
            <OMV name="a"/>
          <OMS cd="alg1" name="one"/>
```

This definition in the content dictionary first supplies the name of the symbol, the role in which it will be used, and a human-friendly description. Then, an OpenMath object is given which provides a universally-

quantified definition for addition, as multiplication by the inverse of the divisor.

The content dictionary system of OpenMath provides an extensibility that MathML does not offer, since the latter system is restricted to K-14 mathematical concepts[3]. This is one of the reasons that the two systems have historically been complimentary to one another, one providing a robust system for expressing semantics, and the other providing a standardised application for presentation of content.

OMDoc

OpenMath provides a way for mathematical content to be expressed in a portable and reliable way, while MathML provides the means to present the same content on the World Wide Web. Formulæ may be expressed precisely and definitions built up from first principles. However, there are structures which are intrinsic to mathematical documents which are not captured well, if at all, by these systems. The relationship between document elements such as sections and chapters and mathematical units such as proofs and definitions; the structure of proof arguments; the organisation of theories and families of theories. None of these are captured very well in the current system. Enter OMDoc.

OMDoc is designed to provide an XML system for encoding whole documents with all their nuances. It favours the use of OpenMath for encoding sentence-level content, and MathML for presentation, when appropriate. But OMDoc wraps these with information about the larger structure of the document. The following is a quite-minimal OMDoc document:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE omdoc PUBLIC "-//OMDOC//DTD OMDoc Basic V1.2//EN"
  "http://www.mathweb.org/omdoc/dtd/omdoc-basic.dtd" []>

<omdoc xml:id="rings.omdoc" version="1.2" modules="@basic"
       xmlns="http://www.mathweb.org/omdoc">
```

[3]This restriction is meant to provide a specific, sufficiently-broad coverage of mathematical ideas for the majority of common applications, so as to ensure compatibility with a variety of clients with possibly-limited rendering resources.

```
  <omtext xml:id="all">
    <CMP xml:lang="en">
      \backslash textbf{Definition.} A ring $R$ is a nonempty
      set with two binary operations...
      ...
    </CMP>
  </omtext>
</omdoc>
```

The first thing you may notice is that the content (in the <CMP> tags) is listed using LaTeX syntax. This is to emphasise that, because it is an XML technology, OMDoc is format-agnostic, and is able to provide a wrapper for many kinds of documents. However, in typical use the content is expressed using the MathML and OpenMath systems, to ensure smooth communication between computer systems as well as the World Wide Web.

The system is designed to accomodate varying document structures; a document may generally be classified as either *knowledge-structured* or *narrative-structured*. The former is a way of organizing the document content in an astract way, in the manner that an expert or a computer system would. It is treated as a reference, and order is not so important in defining the text. On the other hand, a narrative stucture is organised in the way a student would need the material when encountering it for the first time: it presents the knowledge in a sequence, building up from one step to the next.

Most OMDoc documents have several components in common. There is the document root, which includes information establishing the document data type, the namespaces and OMDoc modules to be used in the document, and the root element of the XML tree. There may be metadata (not shown in our sample document) which can include information about the provenance of the document. OMDoc often employs the Dublin Core format for metadata, although this is not required. The document may include comments, which are not part of the mathematics but for the benefit of those who edit the OMDoc tree. Documents usually include structural information, and they may include references between parts of the document. Finally, there is a system of extensible modules which can

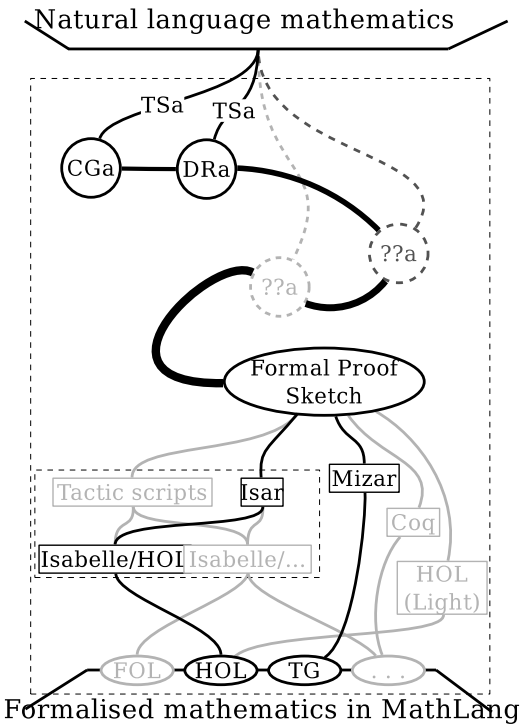


Figure 1.2 Paths one might follow in computerising a document.

add functionality to the OMDoc system.

The current stable version of OMDoc is 1.2. The system is under active development, and at the time of writing is being extended in anticipation of a version 2.0. Together, OMDoc, OpenMath, and MathML provide a system which allows mathematics to be put on the web and passed between systems in a reliable, open, and standard way.

1.3 Contributions

The work of this PhD is to develop a path to computerise written mathematics. This path is constructed as an extension to the existing MathLang framework. Presently the framework allows an author to convert their writings to an intermediate computerised form. The goal of this PhD is to develop a path for translation of texts from this intermediate computerised form to fully-formalised Isabelle texts. Through this we *gain ad-*

ditional confidence that the intermediate form has utility in computerising mathematics.

From the above it may be seen that for a semantically helpful computerisation of mathematical knowledge, today's systems require the use of a formal language which differs in some way from the common, natural, mathematical language. This dissertation proposes a method to **restore natural language as the primary input for computerised mathematics**. The motivation is to provide mathematicians with straightforward tools they can employ to use computers in their everyday work. Efforts towards this goal fall into several categories.

1. *An integrated system for natural-language text input and grammatical categorisation.* A new approach to authoring natural language texts is presented in Section 4.3. As the natural language text is composed, each word or phrase is placed into a certain grammatical category as enumerated in Table 4.3. This is achieved by annotating the original natural language text either during or after its composition. A typical work pattern is presented in Section 9.2.

2. *Tools for reconciling complex expressions to simple grammatical categories.* In Section 5.2 we give several transformations a user may apply to plain text in order to cause the expression to cleanly fit a grammatical classification. These tools are built on top of the aforementioned authoring approach and work to reconcile varying natural writing styles to the stricter grammatical rules. The effect is to duplicate, shuffle, and unfold natural language text so that it is expressed in an explicit manner and strict order. These rewriting rules constitute a "dual" of syntax sugaring which we call *syntax souring*.

3. *An abstract framework to assert the foundational reliability of the proposed system.* The narrative in Section 6.1 presents an operational system which provides a rigorous framework upon which the denotational meaning can rest. It provides a data structure for mathematical documents, incorporating the grammatical categorisation, syntax sour-

ing notions, and a set of rewriting rules which achieve the souring functionality presented in the earlier sections.

4. *A method of eliding redundant portions of a tree.* We introduce summary trees, which allow a human viewer to prune a visually-rendered tree so as to better apprehend the structure which might be found there.

5. *A process for arriving at rules for translation.* Our goal is to produce, from the already-computerised MathLang document, a text in the language of a formal system. Our chosen target language is Isabelle. To do this, rules are created which operate recursively on a document. The nature of these rules is described in Section 8.1, and is illustrated by a detailed example.

The primary illustration for these accomplishments is an excerpt from a textbook [12, Ch. 12] which may be found in Appendix A. Chapter 9 is a narrative which ties all these methods into a cohesive workflow, demonstrating how these tools might be used by a mathematical author.

This dissertation describes developments in MathLang (a system for computerising mathematics, described in Chapter 4) which may help close the gap between natural language and formalised documents. MathLang is a system which tries to give as much flexibility to the user as possible, trying to process any style that a person may use to express their mathematics. The existing parts of MathLang, which are assumed to be the starting point for the developments in this dissertation, are reviewed in Section 4.1.

1.4 Chapter Overview

This dissertation is organised into the following chapters. They can be thought of in several groups. Chapters 1–3 describe the research ecosystem in which the project lives. Chapter 4 describes preexisting portions of the MathLang project. Chapters 5–9 describe in detail contributions made by this PhD. Chapter 10 provides concluding thoughts on the material, the

appendices provide the figures from the examples. Here are the chapter contents:

Chapter 1 This introductory chapter; a discussion motivating the work and another summarising the contributions of the PhD.

Chapter 2 A survey of external projects and tools which have been directly employed in the MathLang project.

Chapter 3 Example of a natural-language text which has been directly formalized in Isabelle, without any intermediate computer asisstance in the creation of the formal document.

Chapter 4 Detailed overview of the preexisting MathLang system; descriptions of Core Grammatical aspect and Document Rhetorical aspect.

Chapter 5 Introduction and description of the method of syntax souring, which is used for making mathematical expressions more explicit, for easier computer processing.

Chapter 6 A description of the operational semantics for certain things described in Chatpers 4 and 5, and a brief discussion of the TEXMACS plugin which implements certain parts of MathLang.

Chapter 7 Explanation of summary trees with examples.

Chapter 8 Technical details of translation rules, showing rule set for main example; method for creating rules.

Chapter 9 Narrative showing process of examples undergoing processing by methods described in chapters 4–7.

Chapter 10 Conclusions and ideas for future work.

Appendix A Ring theory example in figures.

Appendix B Number theory example in figures.

Bibliography Works cited in this document.

Index of Subjects Concepts discussed in this document.

Authors of Referenced Work Index of authors, referencing pages where their work was mentioned.

1.5 Review

In this chapter we saw ways that mathematicians might benefit from automation of computer-aided tools and systems. Although computers now only provide limited, specific assistance to mathematicians in their work, there are a variety of ways in which computers could be of great use. Many of these ways only exist conceptually, while others have been implemented, but not in a manner which has endeared widespread adoption. Many people are actively working to improve the situation.

When it comes to using computers to help mathematicians work better, solutions are plentiful. This chapter has only scratched the surface of the field, and particularly has focused on projects that have similar goals and motivations to the MathLang framework. It included an overview of methods in current use for applications, such as storing information and formalising mathematics. Chapter 2 will describe some specific projects which are directly employed by the MathLang system.

MathLang provides methods of encoding the documents that mathematicians currently write, in the style that they prefer to write, and methods of augmenting the documents with additional information and using this information to process the documents in interesting ways. Some of these methods, including syntax souring for reconciling complex expressions, translation rules for mapping documents to theorem proving languages, and summary trees for discovering such translation rules, were developed by this PhD. student in collaboration with colleagues in the ULTRA research group. The following chapters describe the framework, from its most recent developments to its origins.

Chapter 2

Supporting Technological Systems

In the current project, we strive to develop a theory which is independent of specific implementations and bases. But in implementing ideas, it is necessary to provide proofs of concept and starting points for broader work. In addition, there are concepts which can be nicely abstracted from specific tool sets to be employed in a variety of situations. While Chapter 4 shows how the following systems and concepts are employed in our work, this chapter provides an overview of the tools themselves.

First, Mizar and Isabelle, two systems for formalising mathematics, are discussed. These systems have been used by members of the MathLang team as targets for translation. Mizar was the target for the work of Krzysztof Retel, and Isabelle is the target system for the current dissertation. We then introduce the concept of a 'formal proof sketch', which describes a notion of correctness for formal documents which are not completely formalised. Finally, the editor TEXMACS is introduced, because parts of MathLang have been implemented as a plugin for that editor.

2.1 Mizar

Mizar is a language and system conceived and overseen by Andrzej Trybulec, for the purpose of writing formalised mathematics [49, 56]. The language was crafted with a syntax taken from the grammar that is used by mathematicians in typical communication. This makes Mizar accessible to mathematicians; viewing a sample document is often sufficient to gain an intuition of the general structure and flavor of the language. While the symbols used in Mizar code are restricted to the ASCII character set, they are designed to resemble standard mathematical symbology, using symbols like '`c=`' for $\subseteq$, '`in`' for set membership, and '`ex...being...st...`' for existential qualification. The Mizar group has also produced a checker for Mizar documents, which checks the document for syntactical and rhetorical correctness. Since a Mizar article may refer to others, there are two parts to the checking software. The first looks at the document and gathers information about references it makes to other articles. The second takes this gathered information and processes the current article to check for correctness. Once the document has been checked, any errors found are inserted into the text, to be viewed and revised by the author.

The language and system have been under development for nearly 50 years, and are considered stable and mature. Accordingly, the primary effort of the community has been to use the language to build a library of formalised mathematics. In this they have been productive and successful; the Mizar Mathematical Library at this time contains 805 articles with 35524 theorems between them[1].

In Figure 2.1 we see the first few lines of a Mizar article. The document consists of two parts. The article begins with **environ** (lines 7–25), which references the other articles that will be needed for certain notations and prior facts. This portion is divided from the remainder of the article by **begin** (on line 27), which denotes the start of fresh mathematical results. This article describes the beginning of a theory of partially ordered sets.

[1]The library is accessible at `http://megrez.mizar.org/home.html` – statistics were gathered on 2010-11-05.

```
environ

 vocabularies FUNCT_1, XBOOLE_0, TARSKI, SUBSET_1, MCART_1,
      ZFMISC_1, RELAT_1, PARTFUN1, RELAT_2, ORDINAL1,
      WELLORD1, WELLORD2, SETFAM_1, FINSET_1, ORDERS_1;
 notations TARSKI, XBOOLE_0, ZFMISC_1, SUBSET_1, RELAT_1,
      RELAT_2, FUNCT_1, RELSET_1, PARTFUN1, FUNCT_2, MCART_1,
      ORDINAL1, WELLORD1, SETFAM_1, WELLORD2, FINSET_1;
 constructors SETFAM_1, RELAT_2, WELLORD1, PARTFUN1, WELLORD2,
      FUNCT_2, FINSET_1, RELSET_1;
 registrations XBOOLE_0, SUBSET_1, RELAT_1, ORDINAL1,
      PARTFUN1, FUNCT_2, FINSET_1, RELSET_1, WELLORD2, FUNCT_1;
 requirements BOOLE, SUBSET;
 definitions RELAT_2, TARSKI, WELLORD1, RELAT_1, ORDINAL1,
      XBOOLE_0;
 theorems FUNCT_1, FUNCT_2, MCART_1, RELAT_1, RELAT_2,
      RELSET_1, TARSKI, WELLORD1, WELLORD2, ZFMISC_1,
      XBOOLE_0, XBOOLE_1, PARTFUN1, ORDINAL1, SETFAM_1;
 schemes FUNCT_1, XBOOLE_0, ORDINAL1, TARSKI, RELSET_1, RELAT_1;

begin

reserve X,Y for set,
  x,x1,x2,y,y1,y2,z for set,
  f,g,h for Function;

Lm1: (ex X st X <> {} & X in Y) iff union Y <> {}
proof
  thus (ex X st X <> {} & X in Y) implies union Y <> {}
  proof
    given X such that
A1: X <> {} and
A2: X in Y;
    consider x being Element of X;
    x in X by A1;
    hence thesis by A2,TARSKI:def 4;
  end;
  consider x being Element of union Y;
  assume union Y <> {};
  then consider X such that
A3: x in X and
A4: X in Y by TARSKI:def 4;
  take X;
  thus thesis by A3,A4;
end;
```

Figure 2.1 Excerpt of `ORDERS_1.MIZ`, by Wojciech A. Trybulec, from the Mizar mathematical library.

The first thing done is to indicate that certain variables ($X, Y, x, y, x1, f$, and so on) are reserved for certain uses as sets or functions. Then, on line 33, we see the first lemma. "Lm1:" is a label for reference later in the article. This label prefaces the statement $(\exists X : x \neq \{\} \wedge X \in Y) \Leftrightarrow \bigcup Y \neq \{\}$, expressed in the ASCII symbols of Mizar. This is followed by a proof of the logical equivalence in two parts, the right-to-left part being lines 35–43 and the other direction, lines 44–49. Line 50 brings the two directions together to justify the original statement.

2.2 Isabelle

In addition to the previous work on MathLang, this PhD. relies heavily on the project Isabelle. The basic software system Isabelle [42] allows a user to express and record formulae and reasoning steps, providing tools to work above the most extremely basic steps. It is designed to work with a variety of logical foundations, but the most popular is Higher Order Logic (HOL). Isar [53] is a set of enhancements to the language of Isabelle which allow a user to work in a more declarative fashion. It was originally developed separately from Isabelle, but is now accepted as an official part of the system. In this document, unless otherwise stated, the name Isabelle will refer to the proof assistant with the aid of the Isar language improvements, using a logical foundation of HOL.

The remainder of this section is a summary of certain Isabelle features which may be useful in understanding the remainder of this document. It is replicated here, with minor alterations, from the student's senior research project, performed while an undergraduate [30].

```
theory Example
imports Main
begin
```

Before proceeding with the code by which ring theory was developed, it is essential to understand the constructs that Isabelle supplies for these purposes. In the theory that follows, `Example`, an overview of several dif-

ferent constructs of the language is given along with several proof methods that may be used. The focal point of the examples is the monoid, a very simple algebraic object.

Definition 2.1. A *monoid* is a set M paired with a binary operation "$\cdot$", typically called multiplication, also denoted ab. The multiplication is associative, that is, for any three elements $a, b, c \in M$ we have $a(bc) = (ab)c$. Furthermore, there is an identity element e for which $ea = ae = a$ for each $a \in M$.

2.2.1 Axiomatic Classes

One way to define a monoid is with an axiomatic class. Axiomatic classes are used to define types in the language. The first step is to define the operation of multiplication and declare our identity, one.

```
consts
  axtimes :: "'a => 'a => 'a"  (infixl "*" 70)
  one     :: "'a"              ("1")
```

The `consts` keyword announces that symbolic constants are about to be declared. The first constant is called `axtimes`. Two colons separate the constant name from its type, `'a ⇒ 'a ⇒ 'a`, which indicates that this constant is a function which takes two arguments of the same type (`'a`) and returns another value of that type. The `'` prefix indicates that `'a` is a *type variable*, meaning that any type could be inserted here. Finally, infix notation is declared. Normally, if two elements `a` and `b` are to be multiplied, the function would be applied by writing `axtimes a b`, but now it may also be written `a · b`. The `l` in `infixl` means that if several multiplications are performed in a row without governing parentheses, the multiplication is associated to the left, that is, `a · b · c = (a · b) · c`.

The second constant is `one`. It is of arbitrary type and may be represented by the symbol `1`. It is presently *very* abstract, but will be given deeper meaning when we stipulate its behavior, below.

In actuality, these are very similar to axiomatic classes defined in the Isabelle/HOL source code, so when it comes time to declare the class for the monoid, these constants may be inherited from the existing classes, as below.

```
axclass axmonoid < times, one
  assoc:    "(a * b) * c = a * (b * c)"
  ident_l:  "1 * a = a"
  ident_r:  "a * 1 = a"
```

Here the axiomatic class *axmonoid* is declared which is derived from *one* and *times*, the basic class in Isabelle. It has three axioms which are self–explanatory: the associative axiom and the two axioms dealing with the behavior of the identity.

Recall that this is a class, not a type. This construction describes a whole host of types which are associative and have multiplicative identities. Since it is defined this way, it is now possible to show that some specific types satisfy these properties. This is achieved with the *instance* keyword.

```
instance int :: axmonoid
proof
  show "!! (a::int) (b::int) (c::int). a * b * c = a * (b * c)"
    by arith
  show "!!a. a * (1::int) = a" by arith
  show "!!a. (1::int) * a = a" by arith
qed
```

Here it is shown that the type *int* is an instance of the class *axmonoid*. This is a fact that requires proof, which is achieved by proving that arbitrary elements of type *int* satisfy each of the axioms of the *axmonoid* class. An excellent introduction to proof methods with Isabelle may be found in *Isabelle/HOL — A Proof Assistant for Higher-Order Logic* [43], and an overview of the more lucid methods of proofs with Isar may be found in *Structured Proofs in Isar/HOL* [41]. For the purposes of this example, note that the *proof/qed* pair encloses a proof environment. Within this environment, *show* declares a statement which should satisfy one of the

subgoals of the proof. The keyword *by* announces the tactic by which the statement should be proved. The tactic *arith* is one of many available. This tactic uses properties of arithmetic to prove statements, which is helpful in this setting of the integers.

2.2.2 Locales

In addition to axiomatic classes, monoids and other objects may be implemented through a construct called a *locale*. This approach is explained to great extent in *Locales and Locale Expressions in Isabelle/Isar* [5] and used extensively in *The Hahn-Banach Theorem for Real Vector Spaces* [6]. The principle behind the locale is that of specificity. Axioms, variables, functions, and the like are restricted to certain contexts, and are ignored when outside of the context unless explicitly exported. A monoid could be defined in the following way.

```
locale locmonoid =
  fixes M
  and locMult :: "['a, 'a] => 'a"   (infixl "*" 80)
  and one :: 'a    ("1")
  assumes assoc: "[|a ∈ M; b ∈ M; c ∈ M|]
    ==> a * (b * c) = (a * b) * c"
  and ident_l:  "a ∈ M ==> (1::'a) * a = a"
  and ident_r:  "a ∈ M ==> a * (1::'a) = a"

end
```

2.3 Formal Proof Sketches

A motivating use case for MathLang is the formalisation of mathematical texts. Accordingly, once a document has been annotated with MathLang, it should be easier to compose the initial lines of code in a theorem prover. While the nature of natural mathematical communication leaves gaps that would cause a theorem prover to stumble, an annotated document should

have enough information so that (hopefully) trivial transformations to the language of a theorem prover will provide an outline or skeleton for a formalisation.

Several developments have been made in recent years to improve the ability of users to write documents for various proof systems. One development has been the development of the Isar proof language [41, 53] for the Isabelle proof assistant.

Mizar and Isar have been chosen as two important targets for such transformations. Successful completion of formal proof sketches (as developed by Freek Wiedijk [57] and applied to Isar by Markus Wenzel in collaboration with Wiedijk [54]) from the annotated version of our ring theory excerpt will provide proof-of-concept for this approach. A formal proof sketch is a document written in the syntax of a certain theorem prover, with the allowance that it may be checked by the system with errors, but they must only be justification errors. In Mizar, these errors are the "`*1: It is not true`" and "`*4: This inference is not accepted`". The reader may find an example of a Mizar formal proof sketch in Listing 4.1.

In Isabelle, however, it is not possible to continue checking a document after the first error, so instead we allow the use of the "**sorry**" command to artificially complete proofs. Consider the following Isabelle proofs.

```
5  theorem "p & q ==> q & p"
   proof −
7    assume PQ: "p & q"
     show       "q & p"
9    proof
       from PQ show P: "p" ..
11     from PQ show Q: "q" ..
     qed
13 qed
```

The code starts out with a basic implication, then the proof begins on line 2. The premise is assumed, then the conclusion is asserted with **show**. Then there is a proof which establishes the facts "p" and "q", and from these draws the appropriate conclusion. This is a proof that satisfies Isabelle that the original implication, "p & q ==> q & p", is valid. If we

wish to put off these details for another time,[2] it is possible to replace the inner proof (or any other justification command, like ".") with the command **sorry**, as we see here:

```
theorem "p & q ==> q & p"
proof −
  assume PQ: "p & q"
  show       "q & p"
  sorry
qed
```

In Isabelle, **sorry** is used as a promise from the user to the prover, indicating that the current fact-to-be-proved will be proved at a later time, once more of the theory has been developed. Isabelle marks the result as 'proved, but tainted', and permits the checking of further points in the theory under this qualified assumption that the proof is true. In Isabelle, this proved-but-tainted criterion defines a formal proof sketch.

2.4 TEXMACS

TEXMACS is a scientific word processor whose design was inspired by Emacs and LaTeX. The ethos of the editor is to provide an efficient, extensible way to edit structured documents. A TEXMACS window with an open document is shown in Figure 2.2. The document preparation side of the editor uses professional TEX fonts and page layout algorithms to achieve the highest-quality typesetting possible, and does its best to render documents while editing in a WYSIWYG mode, rather than employing an edit-compile-view workflow. Everything from bold and section commands to lists and macros are rendered in the same presentational manner, and the user is protected from such mistakes as forgetting to match braces or begin/end environment tokens.

Of particular interest to the MathLang project is the editor's capacity to be extended with plugins. There have been plugins created by a variety of people, some of which have the primary purpose of providing an inter-

[2] Often because one is confident that the fact is true, but the appropriate supporting details have not been dug out of Isabelle's library.

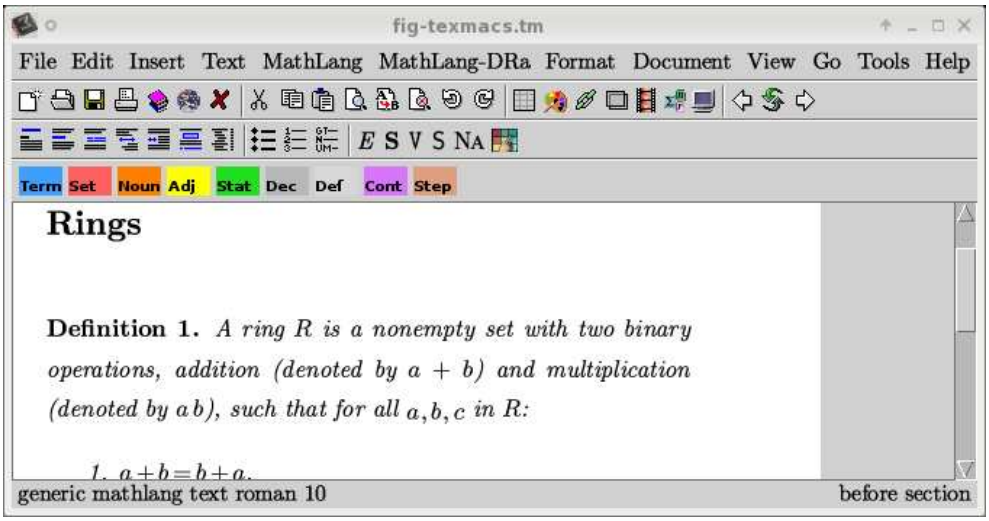


Figure 2.2 The TEXMACS window.

face to external systems like Coq and Ωmega [2, 3, 36]. This same kind of interface, along with style files and macros for special MathLang formatting, have allowed us to create a MathLang plugin for TEXMACS for more natural annotation of text. The annotations are described in Section 4.3, and the particulars of the plugin are shown in Section 6.3.

2.5 Review

In this chapter, we have reviewed several systems which are directly relevant to current developments in MathLang. Some of the systems, such as Isabelle and Mizar, are formal verification systems that are useful targets for translating mathematics. Others, such as TEXMACS, are used in the current implementation of MathLang as supporting systems which are modified through plug-ins to provide a user interface for MathLang. Now that we have described these related systems, the next chapter will lay out the design and details of the MathLang framework.

Chapter 3

Manual Isabelle Translations

Prior to working on this PhD, the student did a dissertation which involved using Isabelle to formalize mathematical content. Work on the dissertation was started with no prior knowledge of Isabelle or other theorem proving systems. It was performed as the two-semester-long senior research project for a Bachelor of Science degree. The major course of study was mathematics, with a minor in computer science. For the dissertation, basic use of the Isabelle system was studied, leading to the formal documents which you see here. In addition, the foundational theories of Isabelle were studied, particularly in relation to the traditional foundations of mathematics. The following sections are excerpted from that dissertation [30].

We include the following excerpts as an example of the kind of Isabelle code which may be produced by a direct encoding from natural language mathematics (Sections 3.1 and 3.2) to a completely formal Isabelle document (Sections 3.3, 3.4, and 3.5). They may be compared to the code shown in Sections 9.6, 9.7, and 9.8.

3.1 Mathematical Definition

The goal of this project is to faithfully model ring theory in Isabelle. The first step is to rigorously define it.[1]

Definition 3.1 (Ring)**.** Let R be a set and a, b, and c be elements of R. Then a *ring* is the set R paired with two binary operations, addition ($+$) and multiplication ($\cdot$) which act on the elements of R such that the following conditions hold:

1. Addition is associative. That is, $a + (b + c) = (a + b) + c$.
2. Addition is commutative. That is, $a + b = b + a$.
3. There is an element $0 \in R$ such that for any $a \in R$, $a + 0 = a$. This element is called the *additive identity*.
4. For every $a \in R$ there is an element $-a$ for which $a + (-a) = 0$. This element is called the *additive inverse* of a.
5. Multiplication is associative. That is, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
6. Multiplication distributes over addition to the left and to the right. That is, $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(b + c) \cdot a = b \cdot a + c \cdot a$.

Note 3.2. By convention, we write $a - b$ to mean $a + (-b)$ and ab to mean $a \cdot b$.

Given a ring or two, it is possible to extract new kinds of objects. The most obvious is the subring.

Definition 3.3 (Subring)**.** Given a ring R and a set $S \subseteq R$, S is a *subring* of R if S is also a ring under the binary operations of R.

Placing certain restrictions on the elements of a subring gives us other constructions.

[1] Some of the definitions below are derived from [12].

Definition 3.4 (Ideal)**.** Consider a ring R and a subring S with $a \in S$ and $r \in R$. If $ra, ar \in S$ for all a, r, then S is called an *ideal* of R.

From here we may construct another type of ring, one which affords an interesting and useful perspective on the study of many rings.

Definition 3.5 (Quotient Ring)**.** Suppose $S \subseteq R$ are rings. Then the set of cosets $Q = \{r + A \mid r \in R\}$ is a ring under the operations $(r + A) + (s + A) = (r + s) + A$ and $(r + A)(s + A) = rs + A$ exactly when S is an ideal of R.

3.2 Definitional Changes

Due to the nature of the above definitions and the manner in which it seemed good to define these objects in Isabelle, it was necessary to arrive at alternate definitions for certain operations. Most pertinent is the definition which was eventually used for coset multiplication in quotient rings.

In the various attempts at defining the quotient of a ring and its ideal, several coset definitions were considered. Based on these various definitions, several definitions were considered for the multiplication of cosets.

Some attempted coset definitions were pointed and some were not. Given $I \lhd R$ and $r \in R$, the coset $I + r = \{i + r \mid i \in I\}$ may be defined either as

- the ideal paired with the characteristic point, (I, r);
- the resulting set paired with the characteristic point, $(\{i + r \mid i \in I\}, r)$; or
- simply the resulting set $\{i + r \mid i \in I\}$.

The first approach was abandoned before much consideration was given. The second definition was given more serious consideration, since it was initially believed that such a definition would permit a more direct use of coset addition and multiplication as given in the quotient definition. However, it was soon obvious that such a definition would require that

the elements of the quotient would then be equivalence classes of cosets, and not the cosets themselves. Thus, the third approach was adopted.

As a result, information about representative members of cosets was lost. This has direct ramifications in defining the sum and product of elements of the quotient. The definitions $(I + a) + (I + b) = I + (a + b)$ and $(I + a)(I + b) = I + (ab)$ are no longer useful. Fortunately, there is a straightforward alternative definition which does not depend on distinguished elements.

Definition 3.6 (Coset Addition)**.** Given sets $S, T \subseteq R$, define

$$S + T = \{a + b \mid a \in S, b \in T\}.$$

If S and T are cosets of some $I \lhd R$, then $S + T$ will be equal to the coset generated by the sum of the representative elements of the cosets. That is, if $S = I + s$ and $T = I + t$ then $S + T = I + (s + t)$.

Proof. Consider S and T cosets of $I \lhd R$. Then $\exists s, t \in I \ni S = I+s, T = I+t$. Note that $s \in S$ and $t \in T$.

Let $x \in I + (s + t)$. Then $\exists i \in I \ni x = i + s + t$. Because I is an ideal, $\exists j, k \in I \ni x = j+k+s+t = j+s+k+t$. But $j+s \in I+s$ and $k+t \in I+t$. Thus $x \in \{a + b \mid a \in S, b \in T\}$.

Let $x \in \{a + b \mid a \in S, b \in T\}$. Then $\exists m \in I + s, n \in I + t \ni x = m + n$. But $\exists i, j \in I \ni x = m + n = i + s + j + t = i + j + s + t$. Since $i + j \in I$, we have $x = i + j + s + t \in I + (s + t)$.

Thus, $I+(s+t) = \{a+b \mid a \in S, b \in T\}$, so this is a legitimate definition for addition of members of the quotient. □

Once addition of quotient elements is defined, attention is directed towards defining the analogous operation for multiplication. While the element-wise plus will work for coset addition, a similar definition for multiplication does not hold in general. Given $S, T \subseteq R$ a ring, we start by

defining the following operations.

$$S \times T = \{st \mid s \in S, t \in T\} \tag{3.1}$$
$$\text{gen}(S) = \{I + s \mid s \in S\} \tag{3.2}$$
$$S \cdot T = \bigcup \text{gen}(S \times T) \tag{3.3}$$

Initially, one would hope that the operation $(I + a) \times (I + b)$ would be a satisfactory substitution for the quotient multiplication operation $(I + a)(I + b) = I + (ab)$. However, this definition is not reliable.

Proposition 3.7. It is false that $(I + a)(I + b) = I + (ab) = (I + a) \times (I + b)$.

Proof. Consider the ring $\mathbb{Z}$ with the ideal

$$\langle 6 \rangle = \{\cdots, -6, 0, 6, 12, 18, \cdots\},$$

and the cosets

$$\begin{aligned} \langle 6 \rangle + 2 &= \{\cdots, -4, 2, 8, 14, 20, \cdots\} \text{ and} \\ \langle 6 \rangle + 3 &= \{\cdots, -3, 3, 9, 15, 21, \cdots\}. \end{aligned}$$

Then we have

$$\begin{aligned} (\langle 6 \rangle + 2) \times (\langle 6 \rangle + 3) &= \{\cdots, -12, -6, 6, 12, 18, \cdots\}, \text{ but} \\ \langle 6 \rangle + (2 \cdot 3) &= \{\cdots, -12, -6, 0, 6, 12, 18, \cdots\}. \end{aligned}$$

Since $(\langle 6 \rangle + 2) \times (\langle 6 \rangle + 3)$ omits 0 from the product, it cannot be a valid definition for multiplication of quotient elements. □

Since this fails, the second definition is used.

Definition 3.8 (Coset Multiplication)**.** Given $I \lhd R$ a ring with cosets S, T, coset multiplication in the quotient R/I defined by $S \cdot T$ in (3.3) is equivalent to the standard definition $(I + a)(I + b) = I + (ab)$.

Proof. Since S and T are cosets of I, there exist $s, t \in R$ for which $S = I + s$ and $T = I + t$. Note that $s \in S$ and $t \in T$. From this we see $st \in S \times T$, so that $I + st \in \text{gen}(S \times T)$. Thus, $I + st \subseteq \bigcup \text{gen}(S \times T) = S \cdot T$.

Let $x \in \bigcup \text{gen}(S \times T) = S \cdot T$. Then $\exists r \in S \times T, i \in I$ for which $x = i + r$. So $\exists j, k \in I \ni x = i + r = i + (j + s)(k + t) = i + jk + jt + sk + st$. But since $i, j, k \in I$, we know $i + jk + jt + sk \in I$, so $x \in I + st$. From this we see $S \cdot T \subseteq I + st$.

Therefore, $S \cdot T = \bigcup \text{gen}(S \times T)$ is a valid definition for multiplication of elements of R/I. □

3.3 Rings with Locales

theory *Ideals = Main:*

In this first development of ring theory, basic objects are supported by locales. This is a paradigm which has a different feel than development of axiomatic classes, which is the route eventually chosen. Locales permit a removal from the base constructs of Isabelle as well as a method of encapsulating families of ideas and objects into a particular exclusive context.

3.3.1 Definition

We declare the initial ring locale. It has a variable R which represents the set of the ring. We state the ring axioms along with several other assertions.

```
locale ring = var R +
  assumes non_empty [iff]:  "R ~= {}"
  and plus_closed [iff]:  "a ∈ R ==> b ∈ R
    ==> a + b ∈ R"
  and times_closed [iff]:  "a ∈ R ==> b ∈ R
    ==> a * b ∈ R"
  and plus_commute:  "a ∈ R ==> b ∈ R
    ==> a + b = b + a"
  and plus_assoc_right:
```

```
    "a ∈ R ==> b ∈ R ==> c ∈ R
    ==> (a + b) + c = a + (b + c)"
  and plus_0_right [iff]:  "a ∈ R ==> 0 ∈ R
    ==> a + 0 = a"
  and inverse_is_in [iff]:  "a ∈ R ==> - a ∈ R"
  and plus_inverse [iff]:  "a ∈ R ==> a + - a = 0"
  and minus_notation [iff]:  "a ∈ R ==> b ∈ R
    ==> a - b = a + (- b)"
  and times_assoc [iff]:  "a ∈ R ==> b ∈ R ==> c ∈ R
    ==> (a * b) * c = a * (b * c)"
  and dist_left [iff]:  "a ∈ R ==> b ∈ R ==> c ∈ R
    ==> a * (b + c) = (a * b) + (a * c)"
  and dist_right [iff]:  "a ∈ R ==> b ∈ R ==> c ∈ R
    ==> (b + c) * a = (b * a) + (c * a)"
  and exist_zero:  "0 ∈ R"
```

Then several statements are proved to finish out the ring definition.

```
lemma (in ring) plus_assoc_left:
"a ∈ R ==> b ∈ R ==> c ∈ R
  ==> a + (b + c) = (a + b) + c"
proof -
  assume abc: "a ∈ R" "b ∈ R" "c ∈ R"
  from abc plus_commute
    have "a + (b + c) = (c + b) + a" by simp
  also from abc plus_assoc_right
    have "... = c + (b + a)" by simp
  also from abc plus_commute
    have "... = (a + b) + c" by simp
  finally show ?thesis .
qed

lemma (in ring) plus_0_left [iff]:
  "a ∈ R ==> 0 + a = a"
proof -
  assume a0: "a ∈ R"
```

```
  from this exist_zero plus_commute
    have "0 + a = a + 0" by simp
  also from a0 exist_zero plus_0_right
    have "... = a" by simp
  finally show ?thesis .
qed
```

In addition to the obvious theorems, two arithmetic properties of equations are shown to make further proofs simpler.

```
lemma (in ring) eq_add:
  "a ∈ R ==> b ∈ R ==> c ∈ R
    ==> a = b --> a + c = b + c"
by auto


lemma (in ring) eq_subtract:
  "a ∈ R ==> b ∈ R ==> c ∈ R
    ==> a + c = b + c --> a = b"
proof
  assume abc: "a ∈ R" "b ∈ R" "c ∈ R"
  and "a + c = b + c"
  from this have "(a + c) + (- c) = (b + c) + (- c)"
    by auto
  from this abc plus_closed plus_assoc_right exist_zero
    have "a + (c + - c) = b + (c + - c)"
    by simp
  from this abc
    have "a + 0 = b + 0"
    by simp
  from this abc plus_0_right exist_zero
    show "a = b"
    by simp
qed
```

3.3.2 Homomorphic Maps

Two tests are defined to determine if a function is a homomorphism or an isomorphism. The hom predicate is left untouched, thus far, in the switch from the axiomatic class approach to defining ring to here in the locale.

constdefs

```
 hom :: "('a::ring => 'b::ring) => bool"
        "hom phi == ALL a b. phi (a + b) = (phi a) + (phi b)
                    & phi (a * b) = (phi a) * (phi b)"
 iso :: "('a::ring => 'b::ring) => bool"
        "iso phi == (hom phi) & (ALL b. EX a. phi a = b)
                     & (ALL a b. phi a = phi b --> a = b)"
```

Initial results about homomorphic maps are forthcoming.

lemma *hom_all_additive: "hom phi --> (ALL a b. phi (a + b) = (phi a) + (phi b))"*
proof
  assume *"hom phi"*
  from *this* **show** *"(ALL a b. phi (a + b) = (phi a) + (phi b))"*
    by *(simp add: hom_def)*
qed

lemma *hom_spec_additive: "hom phi --> phi (a + b) = (phi a) + (phi b)"*
proof
  assume *"hom phi"*
  from *this* **show** *"phi (a + b) = (phi a) + (phi b)"*
    by *(simp add: hom_all_additive)*
qed

In addition, here is a constructor for the kernel of a homomorphism.

constdefs

```
 ker :: "('a::ring => 'b::ring) => 'a set"
        "ker phi == { r. phi r = 0}"
```

3.3.3 More Definitions

From this point, ideals are considered. Given a ring R, we may discuss subsets and subrings $S \subseteq R$ and quotients $Q = R/S$. First several of these objects are defined as

```
locale ring_subset = ring + var S +
  assumes containment:  "s ∈ S ==> s ∈ R"

locale subring = ring_subset +
  assumes sub_plus_neg_closed:
    "a ∈ S ==> b ∈ S ==> a + - b ∈ S"
  and sub_times_closed:
    "a ∈ S ==> b ∈ S ==> a * b ∈ S"

locale ideal = ring_subset +
  assumes sub_plus_neg_closed:
    "a ∈ S ==> b ∈ S ==> a + - b ∈ S"
  and sub_times_sup_in_sub_right:
    "s ∈ S ==> r ∈ R ==> s * r ∈ S"
  and sub_times_sup_in_sub_left:
    "s ∈ S ==> r ∈ R ==> r * s ∈ S"
```

and then the quotient ring is defined, first giving functions for the calculation of cosets and an operation on cosets.

```
constdefs
  lcoset :: "['a::{plus}, 'a set] => 'a set"   (infixl "+" 85)
            "a + A == { q . EX s : A . q = a + s}"
  cosetplus :: "['a::{plus} set, 'a set] => 'a set" (infixl "+"
75)
               "A + B == { c. EX a : A. EX b : B. c = a + b}"

locale quotient_ring = ideal + var Q +
  assumes q_is_coset:  "Q = { C . EX r : R . C = r + S }"
  and coset_addition:  "a ∈ R ==> b ∈ R
    ==> (a + S) + (b + S) = (a + b) + S"
```

Unfortunately, this definition does not afford much flexibility. The operations of + and · are not flexible and results involving multiple rings will be awkward or impossible. At this point it was deemed worthwhile to restructure the basic rings and redevelop the theory.

end

3.4 Rings with Records

theory *Pointless = Main:*

3.4.1 Definition

```
record 'a ringform =
  carrier :: "'a set"
  plus    :: "['a, 'a] => 'a"
  zero    :: 'a
  neg     :: "'a => 'a"
  times   :: "['a, 'a] => 'a"

locale ring =
  fixes
       theRing ::   "'a ringform"
   and R       ::   "'a set"
   and addIden ::   "'a"              ("0")
   and add     ::   "['a, 'a] => 'a"  (infixl "⊕" 75)
   and mult    ::   "['a, 'a] => 'a"  (infixl "⊗" 70)
   and addInv  ::   "'a => 'a"        ( "⊖ "  )

  defines
       R_def:       "R       == carrier theRing"
   and addIden_def: "addIden == zero theRing"
   and add_def:     "add     == plus theRing"
   and mult_def:    "mult    == times theRing"
   and addInv_def:  "addInv  == neg theRing"
```

```
  assumes
       plus_commute:
         "a ∈ R ==> b ∈ R ==> a ⊕ b = b ⊕ a"
   and plus_assoc_right:
         "a ∈ R ==> b ∈R ==> c ∈ R
         ==> (a ⊕ b) ⊕ c = a ⊕ (b ⊕ c)"
   and plus_0_right [iff]:
         "a ∈ R --> 0 ∈ R ==> a ⊕ 0 = a"
   and plus_inverse [iff]:  "a ∈ R ==> a ⊕ ⊖ a = 0"
   and times_assoc [iff]:
         "a ∈ R ==> b ∈ R ==> c ∈ R
         ==> (a ⊗ b) ⊗ c = a ⊗ (b ⊗ c)"
   and dist_left [iff]:
         "a ∈ R ==> b ∈ R ==> c ∈ R
         ==> a ⊗ (b ⊕ c) = (a ⊗ b) ⊕ (a ⊗ c)"
   and dist_right [iff]:  "a ∈ R ==> b ∈ R
         ==> c ∈ R
         ==> (b ⊕ c) ⊗ a = (b ⊗ a) ⊕ (c ⊗ a)"
   and exist_zero: "0 ∈ R"
   and plus_closed: "a ∈ R ==> b ∈ R ==> (a ⊕ b) ∈ R"
   and times_closed: "a ∈ R ==> b ∈ R ==> (a ⊗ b) ∈ R"
   and neg_closed: "a ∈ R ==> ⊖ a ∈ R"

locale ideal = ring +
  fixes I :: "'a set"

  assumes
       is_subset: "a ∈ I ==> a ∈ R"
   and contains_diff: "a ∈ I ==> b ∈ I ==> a ⊕ ⊖ b ∈ I"
   and contains_lprod: "a ∈ I ==> r ∈ R ==> a ⊗ r ∈ R"
   and contains_rprod: "a ∈ I ==> r ∈ R ==> r ⊗ a ∈ R"

lemma self_ideal:
includes struct J
```

```
assumes "ring J"
shows "ideal J (carrier J)"
proof (rule ideal.intro)
  show "ring J" by assumption
  show "ideal_axioms J (carrier J)"
  proof (rule ideal_axioms.intro)
    fix a b r
    assume memb: "a ∈ carrier J" "b ∈ carrier J"
                 "r ∈ carrier J"

    show "a : carrier J" by assumption

    from prems ring.neg_closed [of J b]
         ring.plus_closed [of J a "ringform.neg J b"]
    show "plus J a (ringform.neg J b) : carrier J" by auto

    from prems ring.times_closed [of J a r]
    show "times J a r : carrier J" by simp

    from prems ring.times_closed [of J r a]
    show "times J r a : carrier J" by simp
  qed
qed

lemma zero_ideal:
includes struct J
assumes "ring J"
shows "ideal J { zero J }"
proof (rule ideal.intro)
  show "ring J" by assumption
  show "ideal_axioms J {zero J}"
  proof (rule ideal_axioms.intro)
    fix a b r
    assume memb: "a ∈ {zero J}" "b ∈ {zero J}"
```

```
    from prems ring.exist_zero [of J]
      show subset: "a ∈ carrier J" by simp

    have "zero J ∈ {zero J}" by simp
    from prems this subset ring.exist_zero [of J]
         ring.plus_inverse [of J "zero J"]
      have "plus J (zero J) (ringform.neg J (zero J)) : {zero
J}" by simp
    from prems this show "plus J a (ringform.neg J b) : {zero
J}"
      by simp

    assume "r ∈ carrier J"
    from prems subset ring.exist_zero [of J]
         ring.times_closed [of J "zero J" r]
    show "times J a r : carrier J" by simp

    from prems subset ring.exist_zero [of J]
         ring.times_closed [of J r a]
    show "times J r a : carrier J" by simp
  qed
qed



locale quotient = ideal +
  fixes coset :: "'a => 'a set"
    and setsum :: "['a set, 'a set] => 'a set"
    and setmult :: "['a set, 'a set] => 'a set"
    and setneg :: "'a set => 'a set"
    and quotient :: "('a set) ringform"
    and cosetmult :: "['a set, 'a set] => 'a set"
    and generated_cosets :: "'a set => 'a set set"
```

```
  defines
    coset:    "coset j == {s ⊕ j | s. s ∈ I}"
and setsum: "setsum  S T == {s ⊕ t | s t . s ∈ S ∧ t ∈ T}"
and setmult:"setmult S T == {s ⊗ t | s t . s ∈ S ∧ t ∈ T}"
and setneg: "setneg  S ==   {⊖ s | s. s ∈ S}"
and generated_cosets: "generated_cosets S == { coset s | s. s
∈ S}"
and cosetmult: "cosetmult S T ==
         ⋃ (generated_cosets (setmult S T) )"
and quotient: "quotient == (|
    carrier = {coset r | r. r ∈ R},
    plus = setsum,
    zero = I,
    neg = setneg,
    times = cosetmult
    |)"

end
```

3.5 Rings with Axiomatic Classes

```
theory AxRings = Main :
```

3.5.1 Definition

In this third approach at defining rings in Isabelle, the construction of axiomatic classes seemed natural and canonical. The ring's underlying set is taken to be the entire collection of elements of the type, characterized by the constant `UNIV`. The axiomatic class is derived from four other classes: zero, plus, times, and minus. These provide the additive identity and our operations +, · and -. After most axiom names, `[iff]` is placed, indicating that the relation = should be taken to be symmetric.

```
axclass ring < zero, plus, times, minus
 p_commute           :  "a + b = b + a"
```

```
p_assoc        [iff]:  "(a + b) + c = a + (b + c)"
p_ident_r      [iff]:  "a + 0 = a"
p_inverse      [iff]:  "a + (- a) = 0"
t_assoc        [iff]:  "(a * b) * c = a * (b * c)"
dist_left      [iff]:  "a * (b + c) = (a * b) + (a * c)"
dist_right     [iff]:  "(b + c) * a = (b * a) + (c * a)"
minus_notation [iff]:  "a - b = a + -b"
```

3.5.2 Completing the Axioms

The axioms lend themselves to several corollaries. But first, two arithmetic properties of equations are made explicit, to ease several future proofs.

```
lemma eq_add: "a = b --> a + c = b + c"
proof
  assume "a = b"
  from this show "a + c = b + c"
    by auto
qed

lemma eq_subtract: "a + c = b + c --> (a::'a::ring) = b"
proof
  assume "a + c = b + c"
  from this have "(a + c) + (- c) = (b + c) + (- c)"
    by auto
  from this have "a + 0 = b + 0"
    by auto
  from this show "a = b"
    by auto
qed

lemma p_ident_l [iff]: "0 + a = (a::'a::ring)"
proof -
  have "0 + a = a + 0"
    by (simp only: p_commute)
  also have "... = a"
```

```
    by (simp only: p_ident_r)
  finally show ?thesis .
qed

lemma p_inverse_inverse [iff]: "- (- a) = (a::'a::ring)"
proof -
  have "- (- a) = 0 + - (- a)"
    by (simp only: p_ident_l)
  also have "... = a + - a + - (- a)"
    by (simp only: p_inverse)
  also have "... = a + (- a + - (- a))"
    by (simp only: p_assoc)
  also have "... = a + 0"
    by (simp only: p_inverse)
  also have "... = a"
    by (simp only: p_ident_r)
  finally show ?thesis .
qed
```

3.5.3 Properties of Rings

Next several properties of rings are proved. Some, but not all of these, have been converted to the locale definition of ring.

```
lemma t_zero_r [iff]: "a * 0 = (0::'a::ring)"
proof -
  have "a * 0 = a * 0 + 0"
    by (simp only: p_ident_r)
  also have "... = a * 0 + (a * 0 + - (a * 0))"
    by (simp only: p_inverse)
  also have "... = (a * 0 + a * 0) + - (a * 0)"
    by (simp only: p_assoc)
  also have "... = a * (0 + 0) + - (a * 0)"
    by (simp only: dist_left)
  also have "... = a * 0 + - (a * 0)"
    by (simp only: p_ident_r)
```

```
  also have "... = 0"
    by (simp only: p_inverse)
  finally show ?thesis .
qed

lemma t_zero_l [iff]: "0 * a = (0::'a::ring)"
proof -
  have "0 * a = 0 * a + 0"
    by (simp only: p_ident_r)
  also have "... = 0 * a + (0 * a + - (0 * a))"
    by (simp only: p_inverse)
  also have "... = (0 * a + 0 * a) + - (0 * a)"
    by (simp only: p_assoc)
  also have "... = (0 + 0) * a + - (0 * a)"
    by (simp only: dist_right)
  also have "... = 0 * a + - (0 * a)"
    by (simp only: p_ident_l)
  also have "... = 0"
    by (simp only: p_inverse)
  finally show ?thesis .
qed

lemma neg_right [iff]: "a * - b = - ((a::'a::ring) * b)"
proof -
  have "a * - b = a * (- b) + (a * b + - (a * b) )"
    by (simp add: p_inverse)
  also have "... = (a * (- b) + a * b) + - (a * b)"
    by (simp only: p_assoc)
  also have "... = a * ((- b) + b) + - (a * b)"
    by (simp only: dist_left)
  also have "... = a * (b + - b) + - (a * b)"
    by (simp only: p_commute)
  also have "... = a * 0 + - (a * b)"
    by (simp only: p_inverse)
  also have "... = 0 + - (a * b)"
```

```
    by (simp only: t_zero_r)
  also have "... = - (a * b)"
    by (simp only: p_ident_l)
  finally show ?thesis .
qed

lemma neg_left [iff]: "(- a) * b = - ((a::'a::ring) * b)"
proof -
  have "(- a) * b = (- a) * b + 0"
    by (simp only: p_ident_r)
  also have "... = (- a) * b + (a * b + - (a * b))"
    by (simp only: p_inverse)
  also have "... = ((- a) * b + a * b) + - (a * b)"
    by (simp only: p_assoc)
  also have "... = (- a + a) * b + - (a * b)"
    by (simp only: dist_right)
  also have "... = (a + - a) * b + - (a * b)"
    by (simp only: p_commute)
  also have "... = 0 * b + - (a * b)"
    by (simp only: p_inverse)
  also have "... = 0 + - (a * b)"
    by (simp only: t_zero_l)
  also have "... = - (a * b)"
    by (simp only: p_ident_l)
  finally show ?thesis .
qed

lemma t_neg_neg [iff]: "(- a) * (- b) = (a::'a::ring) * b"
proof -
  have "(- a) * (- b) = - (a * - b)"
    by (simp only: neg_left)
  also have "... = - (- (a * b))"
    by (simp only: neg_right)
  also have "... = a * b"
    by (simp only: p_inverse_inverse)
```

```
  finally show ?thesis .
qed

lemma sum_inverse [iff]: "-(a + b) = -(a::'a::ring) + -b"
proof -
  have "-(a + b) = -(a + b) + 0 + 0" by (simp only: p_ident_r)
  also have "... = -(a + b) + (a + -a) + (b + -b)" by (simp only:
p_inverse)
  also have "... = -(a + b) + a + (-a + b) + -b" by (simp add:
p_assoc)
  also have "... = -(a + b) + a + (b + -a) + -b" by (simp add:
p_commute)
  also have "... = -(a + b) + (a + b) + (-a + -b)" by (simp add:
p_assoc)
  also have "... = (a + b) + -(a + b) + (-a + -b)" by (simp add:
p_commute)
  also have "... = 0 + (-a + -b)" by (simp add: p_inverse)
  also have "... = (-a + -b)" by (simp add: p_ident_l)
  finally show ?thesis .
qed

lemma minus_dist_l [iff]: "a * (b - c) = ((a::'a::ring) * b)
- (a * c)"
proof -
  have "a * (b - c) = a * (b + - c)"
    by (simp only: minus_notation)
  also have "... = a * b + a * - c"
    by (simp only: dist_left)
  also have "... = a * b + - (a * c)"
    by (simp only: neg_right)
  also have "... = a * b - a * c"
    by (simp only: minus_notation)
  finally show ?thesis .
qed
```

```
lemma minus_dist_r [iff]: "(b - c) * a = (b * (a::'a::ring))
- (c * a)"
proof -
  have "(b - c) * a = (b + - c) * a"
    by (simp only: minus_notation)
  also have "... = b * a + (- c) * a"
    by (simp only: dist_right)
  also have "... = b * a + - (c * a)"
    by (simp only: neg_left)
  also have "... = b * a - c * a"
    by (simp only: minus_notation)
  finally show ?thesis .
qed
```

3.5.4 Additional Classes

New classes may also be created which derive from the axiomatic class defined above. One stipulates that multiplication is commutative,

```
axclass commutative_ring < ring
 t_commute:  "a * b = b * a"
```

and another asserts that the ring has a unity.

```
axclass ring_with_unity < ring, one
 unity_l   [iff]:  "1 * a = a"
 unity_r   [iff]:  "a * 1 = a"
```

which suggests several results.

```
lemma neg_with_unit_l [iff]:  "(- 1) * a = - (a::'a::ring_with_unity)"
proof -
  have "(- 1) * a = - (1 * a)"
    by (simp only: neg_left)
  also have "... = - a"
    by (simp only: unity_l)
  finally show ?thesis .
qed
```

```
lemma neg_with_unit_r [iff]:  "a * (- 1) = - (a::'a::ring_with_unity)"
proof -
  have "a * (- 1) = - (a * 1)"
    by (simp only: neg_right)
  also have "... = - a"
    by (simp only: unity_r)
  finally show ?thesis .
qed

lemma neg_ones_yield_pos [iff]:  "(- 1) * (- 1) = (1::'a::ring_with_unity)"
proof -
  have "(- 1) * (- 1) = 1 * (1::'a::ring_with_unity)"
    by (simp only: t_neg_neg)
  also have "... = 1"
    by (simp only: unity_l)
  finally show ?thesis .
qed
```

3.5.5 The Integers form a Ring

Once the ring definition has been established, we can give an example of a ring. The integers are convenient.

```
instance int :: ring
proof
  fix a b c
  show "a + b = (b::int) + a"
    by (rule zadd_commute)
  show "(a + b) + c = (a::int) + (b + c)"
    by (rule zadd_assoc)
  show "a + (0::int) = a"
    by (rule zadd_0_right)
  show "a + - a = 0"
  proof -
    have "a + - a = (- a) + (a::int)"
      by (simp only: p_commute)
```

```
    also have "... = (0::int)"
      by (rule zadd_zminus_inverse2)
    finally show "a + - a = (0::int)" .
  qed
  show "(a * b) * c = a * (b * c)"
    by (rule zmult_assoc)
  show "a * (b + c) = a * b + a * c"
    by (rule zadd_zmult_distrib2)
  show "(b + c) * a = b * a + c * a"
    by (rule zadd_zmult_distrib)
  show "a - b = a + - b"
    by auto
qed
```

3.5.6 Homomorphisms

Finally, tests are established to see if a map is homomorphic or isomorphic.

```
constdefs
 hom :: "('a::ring => 'b::ring) => bool"
        "hom phi == ALL a b. phi (a + b) = (phi a) + (phi b)
                    & phi (a * b) = (phi a) * (phi b)"
 iso :: "('a::ring => 'b::ring) => bool"
        "iso phi == (hom phi) & (ALL b. EX a. phi a = b)
                     & (ALL a b. phi a = phi b --> a = b)"
```

These permit some initial results about homomorphic maps.

```
lemma hom_all_additive:  "hom phi --> (ALL a b. phi (a + b) =
(phi a) + (phi b))"
proof
  assume "hom phi"
  from this show "(ALL a b. phi (a + b) = (phi a) + (phi b))"
    by (simp add: hom_def)
qed

lemma hom_spec_additive: "hom phi --> phi (a + b) = (phi a) +
```

```
(phi b)"
proof
  assume "hom phi"
  from this show "phi (a + b) = (phi a) + (phi b)"
    by (simp add: hom_all_additive)
qed


theorem hom_maps_zero:  "hom phi --> phi 0 = 0"
proof
  assume "hom phi"
  from this have eq: "phi 0 = phi (0 + 0)"
    by (simp only: p_ident_r)
  from prems hom_spec_additive [of phi 0 0] have "phi (0 + 0)
= (phi 0) + (phi 0)"
    by (simp add: hom_spec_additive)
  from eq this have "phi 0 = (phi 0) + (phi 0)" by auto
  from this have "0 + phi 0 = phi 0 + phi 0" by auto
  from this eq_subtract [of 0 "phi 0" "phi 0"] have "0 = phi 0"
by auto
  from this show "phi 0 = 0" by auto
qed

theorem ident_is_hom:  "hom (% x. x)"
proof -
  show ?thesis by (simp add: hom_def)
qed

theorem "hom (phi::('a::ring => 'a::ring)) --> hom (phi o phi)"
proof
  assume "hom phi"
  from this show "hom (phi o phi)" by (simp add: hom_def)
qed

theorem comp_is_hom:  — The composition of homomorphisms is homo-
```

```
morphic
  "((hom (phi::('a::ring => 'b::ring))) & (hom (chi::('b::ring
=> 'c::ring))))
    --> hom (chi o phi)"
proof
  assume "(hom phi) & (hom chi)"
  from this show "hom (chi o phi)" by (simp add: hom_def)
qed

end
```

3.6 Review

This has been a demonstration of how mathematical content could be encoded by direct translation to Isabelle with no intermediate assistance from computers. The reader is invited to compare these encodings to the code found in Sections 9.6, 9.7, and 9.8, which was created using a computer-assisted process which we will describe in the following chapters.

Chapter 4

A Framework for Document Processing

The work of this dissertation is done in the context of MathLang, a project conceived in 1999 which has had contributions from two researchers, four PhD students, four MSc students, and several undergraduates. One of the distinguishing characteristics of this project is a top-down approach to taking mathematical documents from natural language to formal, as opposed to most efforts which start with formal first principles and work towards a more natural representation of mathematics. This chapter describes the existing system, both in theory and implementation.

4.1 MathLang

MathLang is a project of the ULTRA[1] research group. The work is inspired by ideas of N.G. de Bruijn expressed while describing his Mathematical Vernacular [11]. In that paper there is described a division of labour that has come to be known as *de Bruijn's path* [19]. This path recommends, in principle, that in the very broad task of computerising a document,

[1]Useful Logics, Types, Rewriting, and their Automation, in the School of Mathematical and Computer Sciences at Heriot-Watt University: `http://www.macs.hw.ac.uk/ultra/`

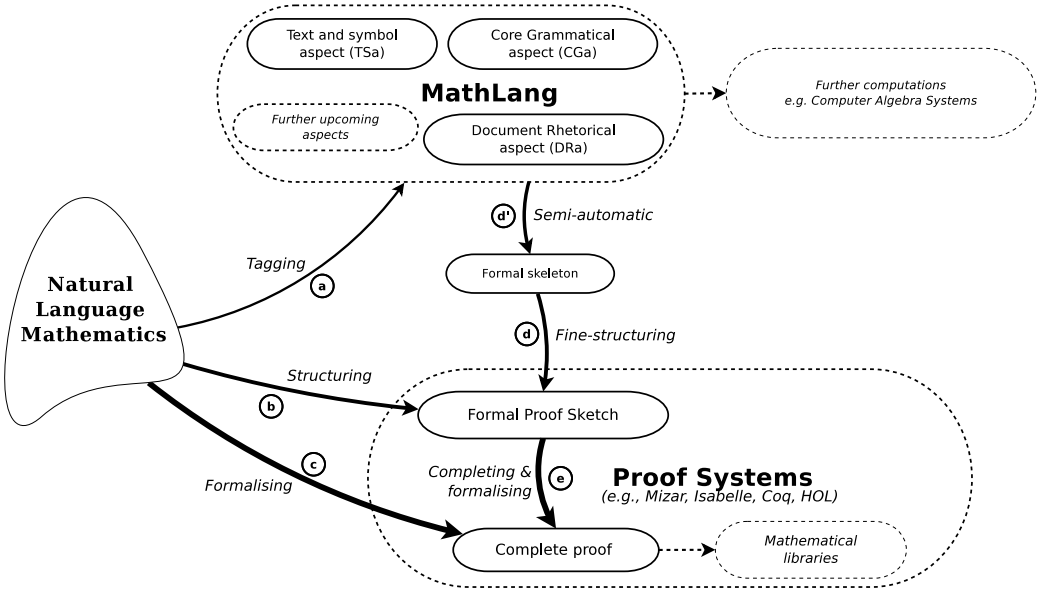


Figure 4.1 Paths of computerisation

the work should be stratified according to skill or technical knowledge required.

The inception of MathLang by Fairouz Kamareddine and J.B. Wells was in 1999 [24], with a vision of a multi-faceted system for computerising mathematics. By taking a gradual approach to this computerisation (see Figure 4.2), it is hoped that the painstaking task of formalising mathematical documents might be broken down into discrete phases, or *aspects*. In concept, MathLang will eventually provide translation to every prover and logical system, performing this task in many small steps which are easy to execute and analyse. In contrast, the provers by themselves currently make the translation in a single large step which is difficult to follow. Accordingly, breaking a task that was monolithic into many intermediate steps may lead to the discovery of the best possible path from natural language to any desired level of formalism. The "best path" is that which respects the mathematics as originally expressed and which takes into account the skills and expertise of users and authors. As MathLang has developed, its team has made every effort to evaluate existing technology and employ the tools and paradigms of others, where applicable, to avoid

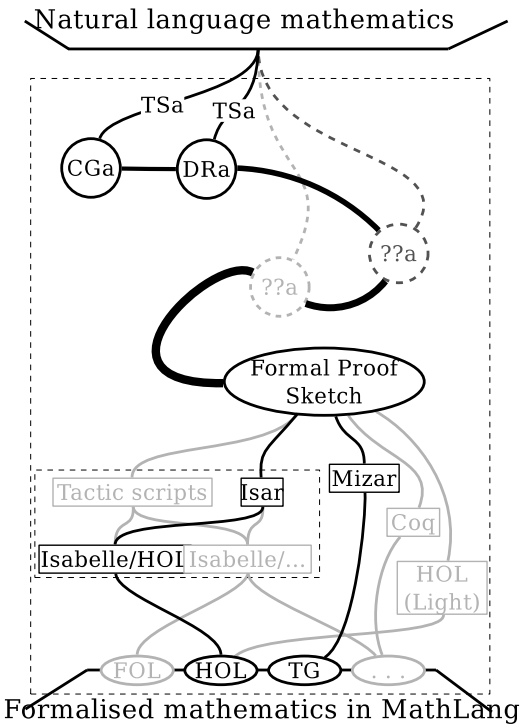


Figure 4.2 Paths one might follow in computerising a document.

duplication of effort.

MathLang is *not* a system for proof verification. This job has been excellently accomplished by many other systems. Rather, MathLang is a framework for computerising mathematics and translating what is written in a natural language into a form on which proof checkers can operate. With this goal in mind, however, development of the MathLang system and theory has a model which is opposed to the development of most proof assistants. Systems such as Mizar, Isabelle, and Coq in some sense take pure logic to be primary. From a very low level, development of the system is driven by a desire to simplify the process of expressing mathematics. MathLang, on the other hand, begins with mathematical text expressed in natural language. With each aspect added to the system comes a greater level of rigour.

For any mathematical document, there are many structures which can be analysed. Each of MathLang's aspects focuses on a distinct kind of

structure. Together these aspects seek to

1. capture mathematical knowledge with more flexibility than systems which create full formalisations,

2. extract structure from natural-language texts, and

3. stratify the verification of a document according to various metrics.[2]

Figure 4.2 visualises some paths one may follow between mathematics expressed in natural language and formalised mathematics. The goal is to eventually find an automatable path from top to bottom. The MathLang path has been roughly sketched for the proof checker Mizar and the aim of this PhD. is to construct it for the proof checker Isabelle.

As previously stated, the aim of MathLang is to computerise mathematics from the original natural language text all the way into fully formalised text in gradual steps. MathLang's way of doing so is to formalise the various steps into aspects. Although the gradual path to Mizar has been demonstrated on a number of examples, formalising all the needed aspects is not completed for this path. Instead, only three aspects on this path have been formalised. The rest of the path into Mizar, although it *has* been roughly defined, still needs a thorough formalisation and identification of the various aspects involved. Up to this point, the remaining path is given via hints and intuition. The team is working on spelling out these hints and experiences into fully blown aspects. What follows is a synopsis of each of those current aspects of MathLang which have been fully formalised and defined. The current aspects are the Text and Symbol aspect [18, 35], the Core Grammatical aspect [22, 35], and the Document Rhetorical aspect [20]. The former is explained in Chapter 5, the other two are described in the following sections.

[2]With MathLang, we wish to accommodate all mathematical documents, even when they are incomplete or contain errors. It should be possible to encode and process documents even when they are not correct.

term common mathematical objects like "2" or "the sequence S".

set sets of mathematical objects such as "$\mathbb{Q}$".

noun families of **term**s such as "sequence".

adjective defines new **noun**s from old. E.g., "infinite" is an **adjective** modifying the **noun** "sequence" to create a new **noun** "infinite sequence".

statement expressions like "$P(m)$" which describe Boolean statements or predicates.

declaration a new **term**, **set**, **noun**, **adjective**, or **statement**.

definition defines new symbols in mathematical texts.

step a group of mathematical assertions.

context preliminary assertions prior to a **step**.

Figure 4.3 The roles of MathLang's grammatical categories

4.2 The Core Grammatical aspect

The Core Grammatical aspect (CGa) is the aspect of MathLang that deals with encoding sentence-level mathematical content. It was developed as a refinement to the weak type theory (WTT) by Nederpelt and Kamareddine [23]. While writing, mathematics is expressed by mathematicians in natural language: either in sentences, such as,

"The square root of two is not rational"

or in formulæ, as,

$$\text{“}\sqrt{2} \notin \mathbb{Q}\text{”}$$

These ultimately are equivalent statements, but there is not a single symbol in common between the first sentence and the second. CGa provides methods for normalising such expressions. There are several important components to CGa:

1. every semantically-useful portion of the sentence is assigned a normalised symbol;
2. the symbols are created with a natural indication for hierarchy, ensuring that mathematical statements are unambiguous;
3. every symbol defined is assigned a grammatical type;
4. the hierarchy and grammatical types together provide a means to check for weak grammatical correctness; and
5. together with the Text and Symbol aspect (described in Chapter 5), it is easy to see the relationship between the computer-friendly normalised symbols and the original text.

Together, these components provide a computerised version of the text which is true to the text and provides certain guarantees of consistency. The remainder of this section describes the components in more detail.

Let us consider the shorter of the two examples above. The formula "$\sqrt{2} \notin \mathbb{Q}$" has four important pieces: there is the constant 2; the function $\sqrt{\ }$; the set membership relation $\in$; the slash through set membership, indicating negation; and the set of rational numbers $\mathbb{Q}$. CGa allows the user to assign computer-friendly symbols to each of these elements. In CGa we call these symbols *identifiers*, which are strings of arbitrary length. It is generally recommended that these identifiers are printable characters from the ASCII character set, to ensure straightforward translation to other systems.[3] For our current example, an author might choose identifiers that correspond in the following way:

$2 \longrightarrow$ `2` $\quad \sqrt{\ } \longrightarrow$ `sqrt` $\quad \in \longrightarrow$ `in` $\quad / \longrightarrow$ `not` $\quad \mathbb{Q} \longrightarrow$ `RAT`

These are the normalised computer-friendly identifiers in their direct correspondence with the components of the formula. In CGa the whole for-

[3]Certain characters are wise to avoid, such as `"`, `'`, and `:`, because they may cause mischief in systems like Isabelle and Mizar.

mula is expressed in the following composite expression, which is called the *interpretation* of the natural-language mathematics:

```
not(in(sqrt(2),RAT))
```

By analogy with application of functions in a C-like programming language, `2` is treated as a parameter to `sqrt`, `sqrt` and `RAT` are treated like parameters to `in`, and that is treated as the single parameter for `not`. This is one way that CGa represents the hierarchical priority of the components of the formula.

It is important to note that cosmetically-different-but-equivalent statements may be given identical computerised interpretations. Consider the other statement from the beginning of the section, "The square root of two is not rational." The following identifiers might be selected for this sentence:

'the square root of' ⟶ `sqrt` 'two' ⟶ `2`
'is' ⟶ `in` 'not' ⟶ `not` 'rational' ⟶ `RAT`

Followed by appropriate further parsing of the sentence, the identifiers are arranged in the interpretation `not(in(sqrt(2),RAT))`, which is precisely what was given for the previous formula.

In order to have a notion of grammatical role for these new identifiers, each one is assigned a type when it is introduced. Each symbol has zero or more parameters (each having its own grammatical category), and one grammatical category for the identifier itself. When declaring an identifier, overall type is expressed as a combination of these, in the following way:

⟨*ident*⟩(⟨*param-cat-1*⟩,⟨*param-cat-2*⟩,...,⟨*param-cat-n*⟩):⟨*ident-cat*⟩

If there are zero parameter types, the parentheses may be dropped. The constituent categories come from Table 4.3. The types for our current example are:

```
sqrt(term):term   2:term   RAT:set
in(term,set):stat   not(stat):stat
```

We can accordingly see that the interpretation `not(in(sqrt(2),RAT))` is well-typed, in that each identifier has the same number of parameters as its declaration, and the identifier categories of parameters match the parameter categories of their parents, in the order given in the declaration. This is what we mean by grammatical correctness.

This introduction to CGa is but a brief look, intended to provide sufficient background for the reader to understand the essentials when reading later chapters. For a thorough presentation of MathLang's Core Grammatical aspect, including details of the type system, see the PhD. dissertation of Manuel Maarek [35].

4.3 Text and Symbol aspect

The purpose of CGa is to encode a certain kind of structure commonly exhibited in mathematical texts. With that goal in mind, the Text and Symbol aspect (TSa) was created to provide two facilities:

1. streamlining the encoding process, and
2. providing a clear relationship between the natural-language text and the encoding.

Both are achieved through a boxed annotation process which creates the CGa encoding while the text is interactively annotated in a graphical word processing environment. Anecdotal testing has shown that new users find the process to be quite clear, and can begin annotating simple documents after only a short introduction to the concepts and method. This section describes TSa as it existed before work on this PhD commenced. We focus on its relationship to CGa as presented in Section 4.2.

So far we have seen how mathematical sentences and formulæ such as $\sqrt{2} \notin \mathbb{Q}$ may be assigned types and interpreted as computer-friendly expressions such as `not(in(sqrt(2),RAT))`. We further observed that cosmetically-different but semantically-identical content may be parsed

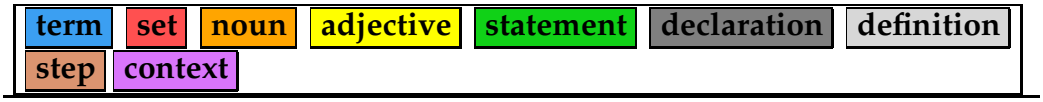


Figure 4.4 Colours associated with MathLang's grammatical categories

and converted into *identical* CGa expressions. As you will see in the following paragraphs, this becomes much more natural with the TSa annotating paradigm.

For every annotation we make, there are three important pieces of information. The foremost is the content of the annotation. The second is the identifier we have selected for computer-friendly CGa encoding. Finally, we need the resultant category from the type of the encoding. Let us start with the simplest annotation possible, a single symbol. The symbol and its type declaration are:

$$2 \qquad \texttt{2:term}$$

So we see that the content is 2, the interpretation is `2`, and its grammatical category is `term`. In order to create an annotation, we draw a box around the content. The box has the identifier in the upper-left corner, and we give a background colour to the box which corresponds to the grammatical category. The category/colour correspondence is given in Figure 4.4; in that table, we see that the colour corresponding to `term` is blue. From this analysis, we see that the appropriate annotation for 2 is:

$$\boxed{{}^{2}\,2}$$

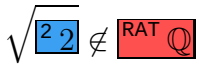
For larger expressions, we take the same approach, but many boxes will be nested, one within another. Consider the whole statement,

$$\sqrt{2} \notin \mathbb{Q}$$

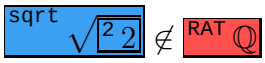
We take the annotations one mathematical concept or symbol at a time. The first one is easy, as we have done it above:

$$\sqrt{\boxed{{}^{2}\,2}} \notin \mathbb{Q}$$

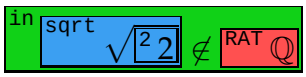
Continuing to look at atomic symbols, we consider $\mathbb{Q}$, which is `RAT:set` in CGa. The colour corresponding to `set` in Figure 4.4 is red, so we annotate this symbol as follows:



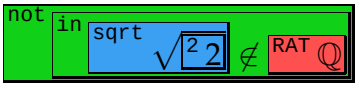
Furthermore, annotating $\sqrt{}$ is straightforward: it is a function applied to 2, so its annotation box should contain the annotation for `2`. The grammatical type of `sqrt` was noted above to be `term`, so it should be blue just as its parameter:



What next? A reasonable way to deal with the middle $\notin$ symbol is to first annotate the expression as $\sqrt{2} \in \mathbb{Q}$, then annotate its negation. Since $\in$ is an infix operator applied to its neighbours $\sqrt{2}$ and $\mathbb{Q}$, we draw this annotation box around the entire statement. Recalling from the previous section that $\in$ is represented as `in(term,set):stat`, we look at its grammatical category, `stat`, and see that this corresponds to the colour green. This gives us the following annotation (before dealing with negation):



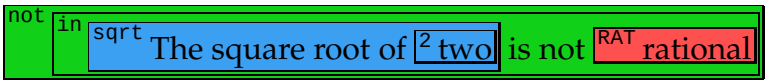
It was determined in the previous section that negation will be encoded as `not(stat):stat`, so that final annotation box is given thus:



Compare this completely-annotated expression with the encoding we arrived at before: `not(in(sqrt(2),RAT))`. We see that the nesting of boxes is the same as the nesting of parentheses, giving us the same structural representation of the mathematics. Furthermore, the TSa method of showing the interpretations demonstrates this nested hierarchy while showing the original text interleaved with the interpretations. This is

valuable, for instance, in the event that something in the original text is changed. Since the identifiers and types are immediately visible, it is more likely that they will be updated to be consistent with the altered text.

Before leaving this introduction to TSa, we will make a couple more observations about how text may be annotated. First, recall from Section 4.3 that different writing styles may lead to identical CGa interpretations. We showed how the formula $\sqrt{2} \notin \mathbb{Q}$ and the sentence, "The square root of two is not rational," may both be given the interpretation `not(in(sqrt(2),RAT))` if the author's intention is the same. With TSa, this correspondence is shown in the way we annotate the text. The latter sentence may be annotated in the following way, leading to an identical box hierarchy as its brother formula:



It all depends on the meaning intended by the author, and TSa is designed so as to provide an author with a straightforward way to make this intended meaning explicit.

4.4 The Document Rhetorical aspect

While CGa, with TSa, is tuned for encoding mathematical statements at the sentence level, the information it captures on a larger scale is less informative than we would like. One of the particular kinds of information that we wish to encode is the relationship between different rhetorical elements of the document; that is, the axioms, definitions, theorems, proofs, and similar units of argumentation, and their interdependence one upon the other. The principal reason that this is not captured well in CGa is that the structure of information encoded by CGa is strictly a tree, whereas (for instance) a proof may rely on several previous definitions and theorems which do not, in turn, directly rely on one another. Thus, this non-tree structure is better encapsulated as a graph. This structure is codified as

Structural roles: chapter, section, paragraph, and part

Mathematical roles: assertion, axiom, claim, conjecture, corollary, definition, example, exercise, lemma, proof, proposition, and theorem

Figure 4.5 DRa roles

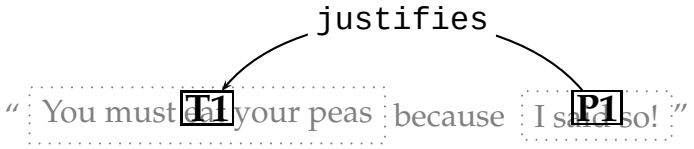
the Document Rhetorical aspect (DRa) of MathLang, which was developed by Krzysztof Retel under the supervision of Fairouz Kamareddine and J. B. Wells. It is defined in detail in Retel's dissertation [47]. This section gives a brief overview of DRa.

DRa works by identifying document units as nodes in a graph, and dependencies between the units as directed edges. There are two kinds of *role* that a unit may have: *structural* and *mathematical*. Structural roles include chapters and parts. These roles indicate typographical divisions in the document; they may not correspond directly to the rhetorical structure, but they exist to help a human reader find their way while reading. These elements may be useful in computer processing, however, since any mathematical units will be contained in structural ones. This organisation may permit the inference of other relationships in the text. *Mathematical* roles indicate units like theorems and proofs, which contribute directly to the development of a mathematical theory. A listing of all the kinds of roles shown in Figure 4.5.

To illustrate this aspect, the reader is invited to think back to childhood. There may have been a time at a family meal when you were served something you were reticent to consume. But when a child's taste buds are growing, parents are loath to listen to complaints. "You must eat your peas because I said so!" is a statement heard at the table of many a young family.

Consider that statement. "You must eat your peas" is an assertion. A theorem, if you will. In the same way, "I said so!" is a kind of justification, or proof, of the statement. If we call the former a theorem named **T1** and the latter a proof named **P1**, these two units may be depicted in the following way:

Here we see two units in the (extremely short) document, each having been defined one mathematical role and no structural role. The first, **T1**, has been assigned the role theorem and the second, **P1**, has role proof. While it is all well and good to assign roles to units, this becomes much more powerful when we connect units together to describe their relationships. We can relate them using RDF[4] triples as `[P1, justifies, T1]`. From this we might obtain a graph which looked like



It is worthwhile to note that, just as some children are less persuaded by this argument than others, DRa does *not* require the logic to be sound. It makes no assumptions about the veracity of the arguments being made: the system only requires that argumentation is acyclic and that some justification is offered when necessary for each claim. More rigorous validation, as with CGa, is left for later tools and aspects. Also, note that in practice it is rare to find theorems and proofs as brief as the above example. For instance, P1 could have been the bulk of a 200-page document and T1 could have been the statement, “Fermat’s Last Theorem holds.”

This example is very small, in order to distill relevant principles for discussion in later chapters. For a view of a larger example, look in Chapter 9, where there is to be found a detailed example taken through many steps, including a more detailed use of DRa (Section 9.4).

4.5 Continuing the Path to Proof Checkers

Reflecting back to Figure 4.2, we see that while the three aspects cover some ground, they are far from completing the computerisation path that we seek. DRa has largely been motivated by translation to Mizar, so let us consider the informal path which presently exists.

[4]Resource Description Framework, described in [33].

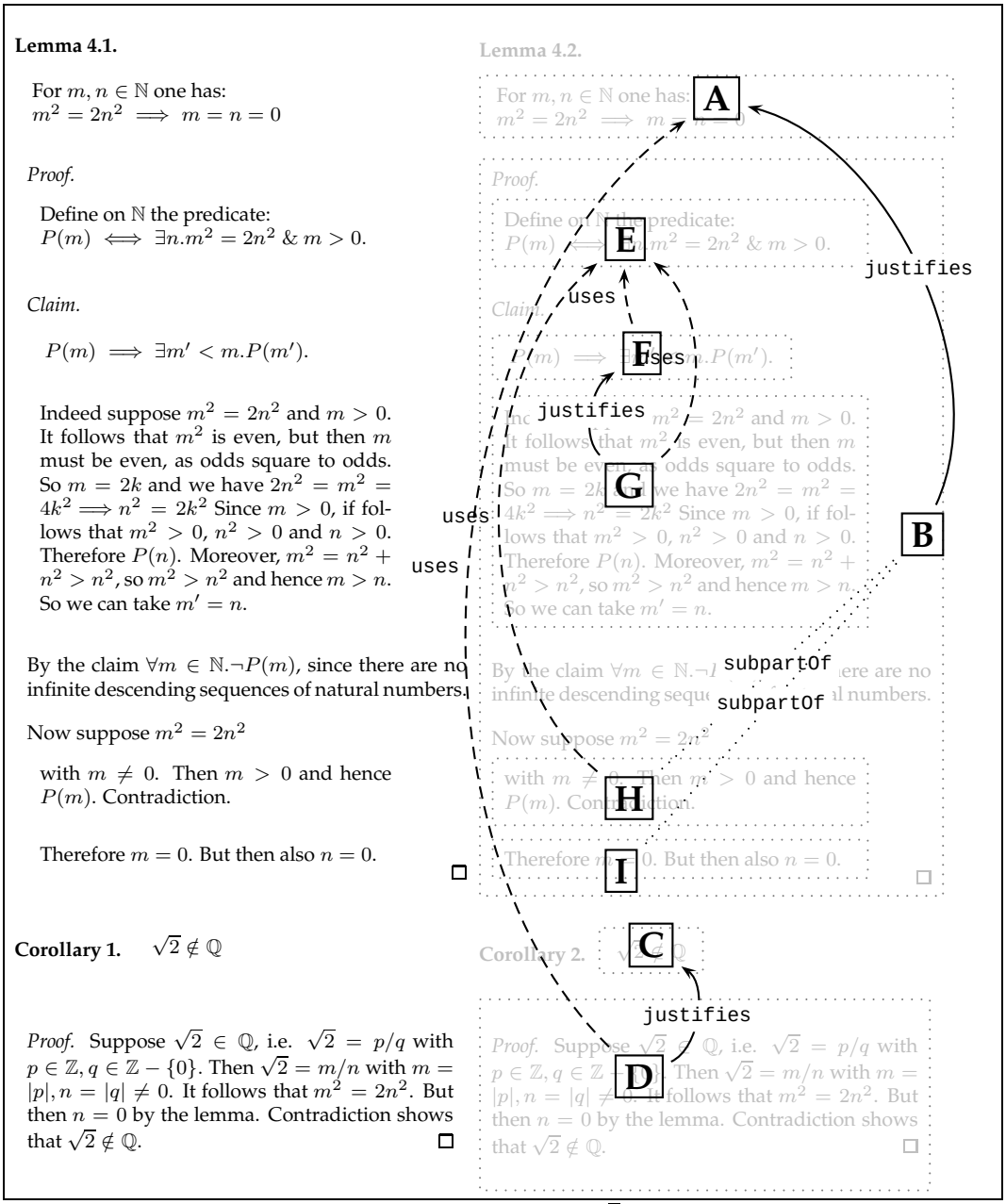
Given some text, we annotate it to the most complete extent possible using TSa, CGa, and DRa. Next, using the hints described in [19] we formulate a formal proof sketch (Sections 2.3 and 9.7) of the document in the Mizar language. After this skeleton has been created, it is necessary to fill in logical holes: steps and justifications which were considered frivolous in the informal, natural-language version of the document but which the Mizar proof checker finds essential to validate the mathematics being represented.

Figure 4.6 and Listing 4.1 are taken from [19]. Figure 4.6 shows a typical proof of the classic result, $\sqrt{2} \notin \mathbb{Q}$, written by Henk Barendregt. This proof appears in *The Seventeen Provers of the World* [55], a compendium of formal proofs in 17 systems such as Isabelle, Mizar, and Coq. In Figure 4.6 we find the dependency graph (DG), which is one result of applying the facilities of DRa to the example, and in Figure B.1 we see one possible CGa annotation of the same example.

In this case, we would like to convert the example to a proof in Mizar. We will not show the way to a complete formal proof – that requires a knowledge of Mizar which is beyond the scope of this paper. But we have a Mizar formal proof sketch of the result in Listing 4.1. Inspection of the listing will illustrate several important properties of the format of Mizar documents. The document is divided into two main parts: the environment-declaration and the text-proper. The former exists to refer to things (results, notations, definitions, etc.) needed from other Mizar files. It begins with `environ` and may be seen on lines 6–14 of the listing. Lines beginning with `::` are comments. After the keyword `begin`, the text-proper starts. This is where new mathematical results are formalized.

Listing 4.1: Encoding of the example from Figure 4.6 in the Mizar FPS

```
1:: This file is verified with the system version:
2:: Mizar verifier= 7.8.03,MML = 4.76.959
3::
4:: Created by Krzysztof Retel {retel@macs.hw.ac.uk}
5
6environ
7 vocabularies INT_1, SQUARE_1, MATRIX_2, IRRAT_1, RAT_1, ARYTM_3, ABSVALUE,
8   SEQM_3, FINSET_1;
```



The original text of Barendregt's version[55] of the proof of $\sqrt{2} \notin \mathbb{Q}$ is reproduced on the left hand side. The right hand side of the figure shows the automatically generated dependency graph for the text where relations between parts of the text are represented by visible arrows and graph nodes have specified (but not visible) mathematical structural roles.

Figure 4.6 Barendregt's version (without and with dependency graph) of the proof of the irrationality of $\sqrt{2}$

```
 9 notations INT_1, NAT_1, SQUARE_1, XXREAL_0, ABIAN, RAT_1, IRRAT_1, XCMPLX_0,
10   INT_2, SEQM_3, FINSET_1, REAL_1, PEPIN;
11 constructors INT_1, NAT_1, SQUARE_1, XXREAL_0, ABIAN, RAT_1, IRRAT_1,XCMPLX_0,
12    INT_2, SEQM_3, FINSET_1, PEPIN;
13 requirements SUBSET, NUMERALS, ARITHM, BOOLE, REAL;
14 registrations XREAL_0, REAL_1, NAT_1, INT_1;
15begin
16
17
18Lemma:  for m,n being Nat holds m^2 = 2*n^2 implies m = 0 & n = 0
19 proof
20   let m,n being Nat;
21   defpred P[Nat] means ex n being Nat st $1^2 = 2*n^2 & $1 > 0;
22   Claim: for m being Nat holds P[m] implies ex m' being Nat st m' < m & P[m']
23   proof
24     let m being Nat;
25     assume P[m];
26     then consider n being Nat such that
27     m^2 = 2*n^2 & m > 0;
28     m^2 is even ;
29::>           *4
30     m is even;
31::>         *4
32     consider k being Nat such that m = 2*k;
33::>                                      *4
34     2*n^2 = m^2
35::>           *4
36          .= 4*k^2;
37::>              *4
38     then n^2 = 2*k^2;
39     m > 0 implies m^2 > 0 & n^2 > 0 & n > 0;
40::>                                       *4,4,4
41     then P[n];
42::>         *4,4
43     m^2 = n^2 + n^2;
44::>                *4
45     n^2 + n^2 >  n^2;
46::>                 *4
47     then m^2 > n^2;
48::>               *4
49     then m > n;
50::>          *4
51     take m' = n;
52     thus thesis;
53::>           *4,4
54   end;
55   A2: for k being Nat holds not P[k]
56   proof
57     not ex q being Seq_of_Nat st q is infinite decreasing by Claim;
58::>                                                          *4
```

```
59      hence thesis;
60::>             *4
61    end;
62    assume A0: m^2 = 2*n^2;
63    per cases by A0;
64    suppose B1: m <> 0;
65      then m > 0;
66::>           *4
67      then P[m] by B1;
68::>             *4
69      then contradiction by A2;
70      hence thesis;
71    end;
72    suppose S1: m = 0;
73      then n = 0;
74::>           *4
75      thus thesis by S1;
76::>               *4
77    end;
78  end;
79
80Corollary:  sqrt 2 is irrational
81   proof
82     assume sqrt 2 is rational;
83     then ex p,q being Integer st
84     q <> 0 & sqrt 2 = p/q;
85::>                      *4
86     then consider m,n being Integer such that
87     A0: sqrt 2 = m/n and m = abs m & n = abs n & n <> 0;
88::>                                                    *4
89     m^2 = 2*n^2;
90::>          *4
91     n = 0 by Lemma;
92::>        *4
93     hence contradiction;
94::>                   *4
95   end;
96
97::> 4: This inference is not accepted
```

4.6 Review

In this chapter we have seen that MathLang is a framework, the architecture of which is divided into various aspects. The Core Grammatical aspect works on the sentence-level of the document's grammatical struc-

ture while the Document Rhetorical aspect works on a larger scale to trace argumentation throughout a paper. These two aspects were largely developed before the start of this PhD. In the following chapter we will see the Text and Symbol aspect, which this student helped to develop as a means of making the theory of CGa more manageable and accessible in the editing of documents.

Chapter 5

Syntax Souring

If all mathematicians expressed ideas in exactly the same way, formalisation would be a non-problem. However, each author has a unique style, with custom notations and different vocabularies. This makes for a richer experience when reading mathematical documents, but it does not help our formalisation efforts. The tools described in this chapter are constructed so as to accommodate a wide variety of writing styles and varying methods of expression. They allow the author to make the intention of the text explicit in ways that would be very difficult to automatically parse. The Text and Symbol aspect is a system for mapping the human-friendly content of the document text to structured symbols that the computer may then parse in interesting ways. The Document Rhetorical aspect, in turn, provides a similar way to mark the rhetorical structure of the document so that the relationships between elements such as axioms, proofs, and lemmas can be connected together. Once this has been laid out, we provide details of syntax souring, a part of TSa which allows authors to clarify the meaning of complicated mathematical expressions.

5.1 Extending the Text and Symbol aspect

A younger part of MathLang, the Text and Symbol aspect (usually denoted TSa) is driven by other aspects to provide methods for user input and for

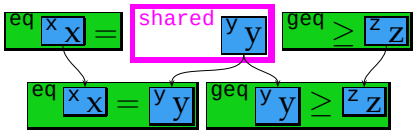


Figure 5.1 Sharing a term between two (in)equalities.

accommodating various writing styles. As part of this PhD, this work was done in collaboration with Manuel Maarek [35] under the supervision of Kamareddine and Wells. It was published at the Mathematical Knowledge Management conference in 2007 [18].

To understand TSa, it is perhaps instructive to expound on the example from the description of CGa in Section 4.2. In that description the phrase "the square of x" was annotated as "sqr the square of x x". With regard to CGa, the essential point was that the original text "x" was interpreted with the string `"x"` and given the type **term**, while the text "the square of ..." was interpreted with the string `"sqr"` and typed as **term**→**term**. The representation here of these interpretations and associations with boxes, super-scripted text, and colours has little to do with CGa. It does, in fact, fall under the jurisdiction of TSa.

This is because the scope of TSa is to comprehend the symbols and language put down by the mathematician and to disambiguate them for processing by one or more other MathLang aspects. Another component of TSa is the *syntax souring* facility, which gives users the ability to write naturally and concisely while providing the computer with instructions on the author's intent. The current features are explained in Section 5.4. The following is a short motivating example.

There are many situations when a mathematician is writing, perhaps recording an argument, and wishes to compare several terms under a sequence of relations. Often the resulting text will be rendered in the manner "$x = y \geq z$". This is problematic, however, in that while annotating such an expression one wishes to simultaneously create annotations amounting to "eq x $x = y$ y $\geq$ z z" and "x x $=$ geq y $y \geq$ z z". Unfortunately, these two **statement**s cannot overlap to both make use of the "y y" in the

middle. One solution is to use the `share` rewriting method from the souring facilities. With this approach, the expression would be annotated as follows, and be automatically rewritten as indicated in Figure 5.1. Read on for a more thorough description of `shared` and related tools.

5.2 Souring Annotations

The grammatical box annotations of Section 4.3 are guided by the style in which the original natural-language sentences were written. Mathematical writing styles are *uneven* and do not always fit such simplistic annotations. To adapt to any style, we need additional box annotations which help interpret the author's style. We believe it is necessary to separate grammatical and style annotations.

Mathematicians use natural language as a medium for communicating mathematical knowledge, but this language is hard for software to automate. Our group showed in [21] that MathLang has constructions that correspond to the way common mathematical justifications are structured. MathLang is automation-friendly and mimics the structure of justifications in natural-language texts. Therefore MathLang authoring does not require the user to alter or translate the document's knowledge for computerisation, although there is a need to adjust the writing style when encoding text directly into the core MathLang language. Because we regard our starting language, natural language, to be the *sweetest* for human readers, we call this modification *syntax souring*. This term describes the process of transforming natural language into syntactically formalised language (the core grammatical MathLang of [22]). The annotations introduced to perform a transformation of natural language to a core formal language are known as *souring annotation*.

Syntax sugaring

The notion of *syntax sugaring* is well known by programmers. Syntactic sugar is added to the syntax of programming languages to make it easier

to use by humans. Syntax sugaring lightens the syntax without affecting expressiveness.

$$\textit{programming language} + \textit{syntactic sugar} \longrightarrow \textit{Core programming language}$$

Syntactic sugar is there for the aid of humans as they write in a computer-friendly language. It is possible to write code without it, but adding sugar to a language makes it more expressive, more pleasant to use, and sometimes more maintainable (if the syntax sugar is well-designed). Undoing the sugaring to create expressions in the core language is extra work for the computer, however, and some people prefer to use the core language directly, to have a more precise idea of what their expressions mean.

Souring: dual of de-sugaring

Syntactic sugar is usually an additive for the syntax of formal language. *De-sugaring* is the process of getting rid of the sugared bits by replacing them with proper core syntax expressions.

$$\textit{natural language} + \textit{grammatical annotations} + \textit{syntactic sour bits} \longrightarrow \textit{Core programming language}$$

In our case the primary input is the mathematician's natural language which we want to extend for computer software use. *Souring* unfolds the sour bits to produce a *sour document*, i.e. a document which is formal enough to be understood by computer software. The original document and the sour one do not belong to the same type of document.

The *duality* between syntax sugaring and syntax souring resides in the fact that both are methods to humanise the authoring of rigid languages but have a different starting point (i.e., programming language for syntax sugaring and natural language for syntax souring). De-sugaring adapts rigid languages for human consumption. Souring rigidifies natural language for software use.

5.3 Denotational representation

We give here the denotational semantics for syntax souring. Denotational semantics are provided to describe the behaviour of a system in terms of the concrete notation used when interacting with the system. As such, they are offered as an aid to the prospective user of the system, providing a non-interactive window to see what the experience of using the system might be like. This is to be contrasted with the operational semantics presented in Sections 6.1 and 6.2, which would generally be of more interest to those who want to understand the internal workings of MathLang.

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 formula formed by symbols. This primary input corresponds to $\mathcal{D}_{\mathcal{F}}$ (formed by $\mathcal{F}$ individuals) in the operational system of Section 6.1. We add to this primary input, grammatical and souring annotations that wrap portions of the text. We already saw in Section 4.3 how we represent grammatical annotations. In this section we explain how we represent the souring annotations discussed in Section 5.2. We denote by T a portion of text which may include formula, grammatical annotations and souring annotations. We denote by A an arbitrary annotation.

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 4.3). Each instance of a grammatical annotation may get an attribute which corresponds to the grammatical annotation's interpretation given in Section 4.3. We represent grammatical annotations by a box whose background colour—according to the colour coding of Table 4.3—informs the grammatical category and whose interpretation is printed on the upper left-hand side of the box using `courier` typeface.

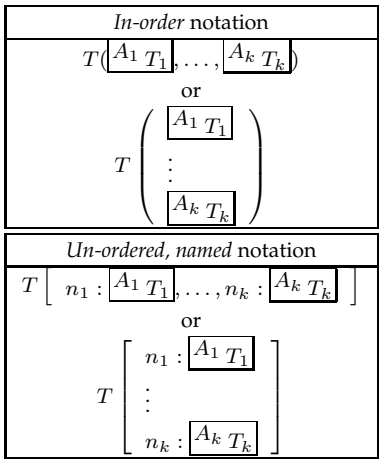
Here is for instance the term a annotated with a **term**-box with `"a"` as interpretation: $\boxed{\texttt{a}\ a}$. We use G, G', G_1, etc., to range over grammatical interpretations. Grammatical annotations correspond to $\mathcal{G}$ labels in the formal system presented in Section 6.1.

Souring annotations

Sour bits correspond to souring annotations. We denote them by a distinguishable font colour and a thicker box for the annotation they describe (i.e., $\boxed{\texttt{list}\ a, b, c}$). We define in the rest of this paper the following syntax souring annotations (which correspond to the elements souring labels $\mathcal{S}_u$ of Section 6.1): `position` i, `fold-right`, `fold-left`, `base`, `list`, `hook`, `loop`, `shared` and `map` (where i is a natural number).

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 two different notations for these parametrised texts: the *in-order* notation where arguments should appear in T in the same order as they appear in the pattern and the *un-ordered* notation where the order of arguments is unimportant. We denote such parametrised notation with $\boxed{A_1\ T_1}$, …, $\boxed{A_k\ T_k}$ being the arguments for T, as in the accompanying diagram. Sometimes, optional names $n_1, \ldots, n_k$ are used to indicate the appropriate argument order. The behaviour of parametrised text is reflected in the de-formatting function (Definition 6.8) and compatibility property (Definition 6.10) from Section 6.1.

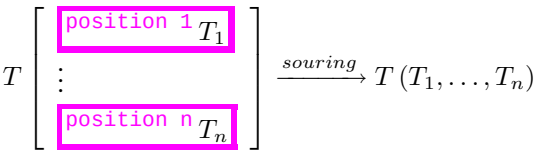


5.4 Souring transformations

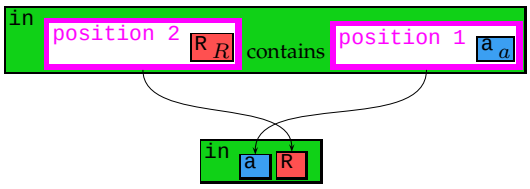
In this section we indicate how to use our souring annotations and describe the result of a souring transformation where the souring notation is unfolded to obtain a text where grammatical annotations are similar to those of Section 4.3. Such a document could then be checked according to the MathLang grammatical checker described in [22].

Re-ordering

`position i` When dealing with a natural language mathematical text, one regularly faces situations where two expressions holding similar knowledge are ordered differently. The re-ordering transformation corresponds to $\rightarrow_{pos}$ of Section 6.2. Considering the expression "a in R" from our supplement example, one can easily imagine the author using "R contains a" instead. The `position` souring annotation is meant for reordering inner-annotations. The souring rewriting function reorders the elements according to their position indices.



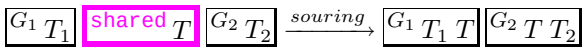
The expressions "a in R" and "R contains a" should both be interpreted as `in(a,R)` if `in` is the set membership relation. To indicate in the second expression that the order of the argument is not the "reading" order, we annotate R and a with `position 2` and `position 1`, respectively. It is common for binary symbols like $\subset$ to have a mirror twin like $\supset$. The `position` souring annotation usefully gives the same interpretation to twin symbols.



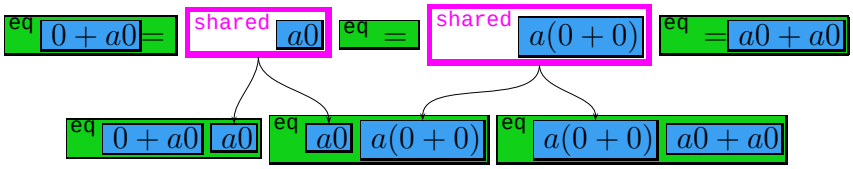
Sharing/chaining

`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.



The `shared` annotation indicates that an expression is to be used by both its preceding and following expressions. The shared expression is inserted at the end of the preceding expression and at the beginning of the following one. This transformation corresponds to $\hookrightarrow_{share}$ of Section 6.2.



This is adequate for many situations, but some formulae need a different way to duplicate parts of an expression. The document example we chose to computerise (see Appendix A) contains several sentences which are made easier to computerise by the use of sharing. The multiple equation

$$0 + a0 = a0 = a(0 + 0) = a0 + a0$$

is certainly the best example as it 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 pair of souring annotations `hook` and `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 $\hookrightarrow_{chain}$ of Section 6.2.

$$T\left(\begin{array}{l}\texttt{hook}\ T'\\ \texttt{loop}\end{array}\right) \xrightarrow{souring} T\begin{pmatrix}T'\\T'\end{pmatrix}$$

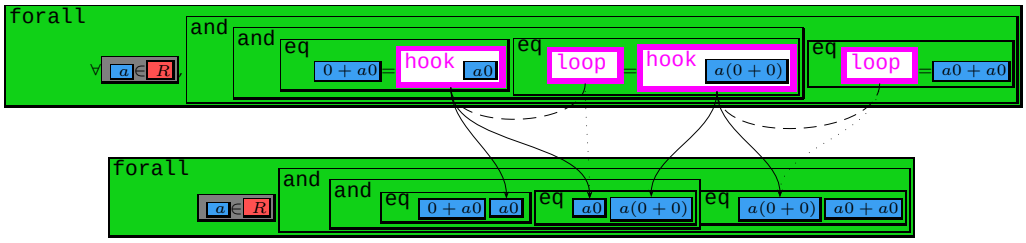
Let us see an example where a `shared` souring annotation could not have been used. If we consider the equation we used in the sharing example and decide to quantify this equation over a, we would obtain

$$\forall a \in R,\ 0 + a0 = a0 = a(0 + 0) = a0 + a0$$

which is effectively a shortcut for

$$\forall a \in R,\ 0 + a0 = a0 \wedge a0 = a(0 + 0) \wedge a(0 + 0) = a0 + a0$$

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`. Here is the way it would look using MathLang annotations:



List manipulations

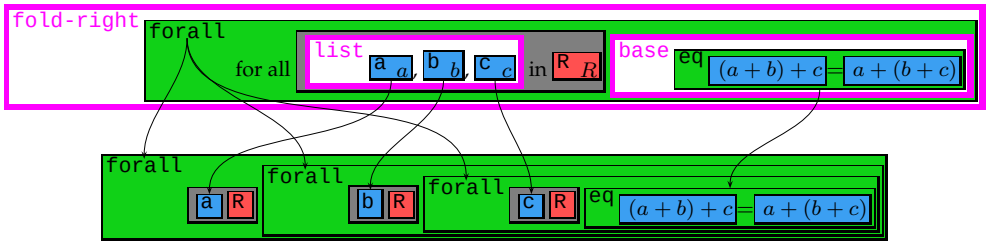
The list souring annotations indicate how lists of expressions have to be unfolded into MathLang interpretations. We define a list folding annotation, `fold-right`, and a mapping annotation, `map`.

$$\texttt{fold-right}\ T_f\begin{bmatrix} b : \texttt{base}\ T_b \\ l : \texttt{list}\ T_1 \ldots T_k \end{bmatrix} \xrightarrow{\text{souring}} T_f\begin{bmatrix} b : T_f\begin{bmatrix} b : T_f\begin{bmatrix} \cdots T_f\begin{bmatrix} b : T_b \\ l : T_k \end{bmatrix} \cdots \end{bmatrix} \\ l : T_2 \end{bmatrix} \\ l : T_1 \end{bmatrix}$$

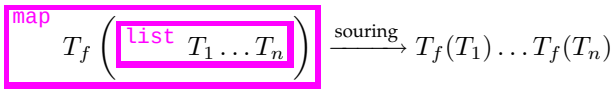
The `fold-right` souring annotation defines a pattern which is repeated for each element of the list argument. For each repeated pattern,

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. These transformations correspond to $\rightarrow_{fold}$ of Section 6.2.

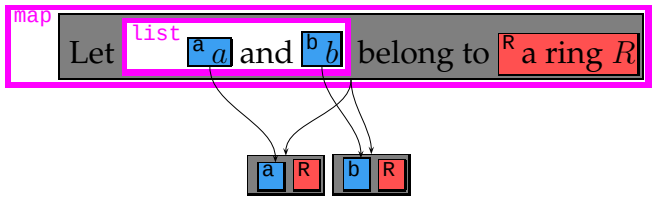
A major use of the `fold-right` souring annotation is to handle quantification over multiple variables. Considering the sentence "for all a, b, c in R *[...]* $(a + b) + c = a + (b + c)$", we would like to use one single `forall` instance for each variable a, b and c. We simply annotate the list of variables as such and the base equation as `base` and the souring unfolding creates a fully expanded interpretation on our behalf.



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. It corresponds to $\rightarrow_{map}$ defined in Section 6.2.



Similarly to folding, this souring annotation is useful for declarations, definitions or statements over several things. 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.



5.5 Review

This chapter has described how TSa works as a system for interleaving an original document and CGa encodings, including methods for reconciling difficult mathematical expressions for explicit comprehension. It also provided the operational notation for MathLang documents, which we use to define the details of this and other aspects of the framework. At the heart of TSa is the relationship between human-friendly documents – as written by mathematician authors – and computer-friendly annotations. The next chapter describes translation rules which are used to process these annotations once they are complete.

Chapter 6

Semantics and Implementations

In this chapter we present the details, both formal and practical, of the ideas which have been presented in earlier parts of the dissertation. In particular, we show the operational semantics of CGa and TSa, which were first presented in Sections 4.2, 4.3, and Chapter 5. Following this, there is some discussion of the implementations of certain specific parts of MathLang.

6.1 Operational System

In this section we define the operational system of MathLang. This definition is published in [18], with some updates and reformatting. Some readers may find it beneficial to keep in mind the definition of the XML XPath data model [10], as there exist strong conceptual parallels with the following definition.

Let $\mathbb{N}$ denote the natural numbers, use $(-;-)$ to denote ordered pairs, and let functions be sets φ of ordered pairs. Every function has a domain $\mathrm{dom}(\varphi) = \{a \mid \exists b.(a;b) \in \varphi\}$ and a range $\mathrm{ran}(\varphi) = \{b \mid \exists a.(a;b) \in \varphi\}$. A sequence is a function σ for which $\mathrm{dom}(\sigma) = \{n \mid 0 \leq n < k\}$ for some $k \in \mathbb{N}$. We write [] for the empty sequence and $[x_0, x_1, \ldots, x_n]$ for the sequence σ such that $\sigma(i) = x_i$ for each $i \in \mathrm{dom}(\sigma) = \{0, \ldots, n\}$. Upon that sequence is defined the metric $|\sigma| = n + 1$. We define σ_1, σ_2 to concatenate

σ_1 and σ_2 as the new sequence σ such that $\mathrm{dom}(\sigma_1, \sigma_2) = \{0, \ldots, |\sigma_1|+|\sigma_2|-1\}$ where $\sigma(i) = \sigma_1(i)$ for $i \in \mathrm{dom}(\sigma_1)$ and $\sigma(i) = \sigma_2(i)$ for $i-|\sigma_1| \in \mathrm{dom}(\sigma_2)$. Concatenation is associative. Moreover, $[], \sigma = \sigma$ and $\sigma, [] = \sigma$. For any set S, say $[S]$ denotes $\{\sigma \mid \mathrm{ran}(\sigma) \subseteq S\}$.

Building on to these basic mathematical concepts, we begin to define objects which are particular to the MathLang system. The reader who has already processed the contents of Chapters 4 and 5 will find many of the following terms to be familiar.

Let $\mathcal{L} = \mathcal{F} \cup \mathcal{G} \cup \mathcal{S}$ be a set of labels such that elements of $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{S}$ are formatting, grammatical, and souring labels, respectively. The set $\mathcal{F}$, of *formatting instructions*, varies according from rendering system to rendering system. We define $\mathcal{G} = \mathcal{C} \times \mathcal{I}$, where

$$\mathcal{C} = \{\textbf{term}, \textbf{set}, \textbf{noun}, \textbf{adj}, \textbf{stat}, \textbf{decl}, \textbf{defn}, \textbf{step}, \textbf{cont}\},$$

and contains identifiers for the primitive grammatical categories of Table 4.3. The set $\mathcal{I}$ consists of strings used for identifying abstract interpretations. We let ℓ, f, g, c and i range over $\mathcal{L}$, $\mathcal{F}$, $\mathcal{G}$, $\mathcal{C}$ and $\mathcal{I}$, respectively.

We let s range over $\mathcal{S} = \mathcal{S}_u \cup \mathcal{S}_i$ where $\mathcal{S}_u$ contains *souring identifiers* to be employed directly by the user:

$$\mathcal{S}_u = \{\texttt{fold-left}, \texttt{fold-right}, \texttt{map}, \texttt{base}, \texttt{list}, \texttt{hook}, \texttt{loop}, \texttt{shared}\} \cup (\{\texttt{position}\} \times \mathbb{N})$$

While $\mathcal{S}_i$ holds several identifiers used internally for rewriting:

$$\mathcal{S}_i = \{\texttt{hook-travel}, \texttt{head}, \texttt{tail}, \texttt{daeh}, \texttt{liat}, \texttt{right-travel}, \texttt{left-travel}\} \cup (\{\texttt{cursor}\} \times \mathbb{N})$$

Note that $\mathcal{S}_u$ and $\mathcal{S}_i$ are disjoint.

These are the preliminaries. Now, we put them together to define a MathLang document, the theoretical object upon which we are going to perform all kinds of operations. Documents have a tree structure, and this

is captured in the following definition by nesting sequences.

Definition 6.1 (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 $(d_1, d_2) \in \mathcal{D}$.

A MathLang document is an element of the set $\mathcal{D}$. In addition, we denote by $\mathcal{D}_{\mathcal{F}}$, $\mathcal{D}_{\mathcal{G}}$, $\mathcal{D}_{\mathcal{C}}$, $\mathcal{D}_{\mathcal{F}\cup\mathcal{G}}$, $\mathcal{D}_{\mathcal{G}\cup\mathcal{S}}$ and $\mathcal{D}_{\mathcal{F}\cup\mathcal{G}\cup\mathcal{S}}$ the sets of documents whose labels are restricted to the respective subscripted set. The variables d and d_n (where $n \in \mathbb{N}$) denote members of $\mathcal{D}$ unless otherwise noted.

Remark 6.2 (Notational convention)**.** We use $\ell\langle d\,\rangle$ to denote $[(\ell; d)]$. When not ambiguous, ℓ denotes $\ell\langle[]\rangle$. A box with black border and **coloured background**, $\boxed{{}^{i}\ d}$, is used to represent $(c; i)\langle d\rangle$ (a document with grammatical label), where the background colour of the box corresponds to c as shown in Table 4.3 (See Example 6.9, below). Similarly, a box with thick pink border and white background, $\boxed{{}^{s}\ d}$, is used to represent $s\,\langle d\rangle$, documents with souring labels.

When discussing documents, it is useful to talk about the relationship between them, particularly whether one document contains another in some meaningful sense. We next define a notion of *sub-document*, a relation which indicates whether a document can be pruned to create another specific document.

Definition 6.3 (Sub-document)**.** We define *sub-document*, denoted by $\sqsubset$, as the binary relation between documents such that:

$$d \sqsubset d \tag{SUB1}$$

$$d \sqsubset g\langle d_1\rangle \text{ if } d \sqsubset d_1 \tag{SUB2}$$

$$d \sqsubset (d_1, d_2) \text{ if } d \sqsubset d_1 \text{ or } d \sqsubset d_2 \tag{SUB3}$$

Remark 6.4. It is important to notice that 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 $d_1 \sqsubset d_2$, we have that $d_1 \not\sqsubset_{\mathcal{G}} \ell\langle d_2 \rangle$. It has been defined this way to expedite the definitions in Section 6.2.

Recall that documents are essentially nested, labeled sequences. Thus, the sub-document relation indicates whether a certain sequence is nested in another. In the same way, we may wish to test whether any of these nested sequences are labeled with a particular label. The following relation, label inclusion, provides a test for this.

Definition 6.5 (Label inclusion)**.** We define *label inclusion*, and we denote by $\widetilde{\in}$, the binary relation between a label and a document such that:

$$\ell \mathrel{\widetilde{\in}} \ell\langle d \rangle \tag{INC1}$$

$$\ell \mathrel{\widetilde{\in}} g\langle d \rangle \text{ if } \ell \neq g \text{ and } \ell \mathrel{\widetilde{\in}} d \tag{INC2}$$

$$\ell \mathrel{\widetilde{\in}} (d_1, d_2) \text{ if } \ell \mathrel{\widetilde{\in}} d_1 \text{ or } \ell \mathrel{\widetilde{\in}} d_2 \tag{INC3}$$

Remark 6.6. Note 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 $\ell_2 \mathrel{\widetilde{\in}} d$, we have that $\ell_2 \mathrel{\widetilde{\notin}} \ell_1\langle d \rangle$. As with Definition 6.3, noted in Remark 6.4, (INC2) is defined this way to expedite the definition of rewriting rules in Section 6.2.

It is worth noting that these notations have been developed for ease of reading, and particularly interactive annotation of texts. One of MathLang's biggest motivations is for humans to be able to type a mathematical text in a natural way on the computer, and then add the grammatical and souring information with ease. The prototype based on TEXMACS is described in detail in [35].

Formatting systems are treated as a set of formatting instructions $\mathcal{F}$, a blank formatting instruction ε, a concatenation operator $\bullet$, and a hole-filling function $\text{fill} : \mathcal{F} \times [\mathcal{F}] \to \mathcal{F}$, which takes two arguments, a formatting instruction f and a sequence of instructions σ. Instruction f may have

holes, denoted $\boxed{n}$, where $0 \leq n < |\sigma|$. The instruction f is rewritten so that each $\boxed{n}$ is replaced by $\sigma(n)$.

In order for us to convert these abstract documents of nested sequences to a nicely-rendered document that a human being might enjoy reading, we define the following function for rendering the documents. It traverses the tree, rendering the document label by label and formatting instruction by formatting instruction.

Definition 6.7 (Rendering functions)**.** Let $\mathrm{r} : \mathcal{D} \to \mathcal{F}$ be defined as

$$\mathrm{r}([]) = \varepsilon \tag{REN1}$$
$$\mathrm{r}(f\langle d\rangle) = \mathrm{fill}(f, [\mathrm{r}(d(0)), \ldots, \mathrm{r}(d(|d|-1))]) \tag{REN2}$$
$$\mathrm{r}((c; i)\langle d\rangle) = \boxed{{}^{i}\,\mathrm{r}(d)} \tag{REN3}$$
$$\mathrm{r}(s\langle d\rangle) = \boxed{{}^{s}\,\mathrm{r}(d)} \tag{REN4}$$
$$\mathrm{r}(d_1, d_2) = \mathrm{r}(d_1) \bullet \mathrm{r}(d_2) \tag{REN5}$$

where the background colour of the box given by (REN3) is the colour from Table 4.4 (i.e., $\mathrm{r}((\textbf{term}; i)\langle d\rangle) = \boxed{{}^{i}\,\mathrm{r}(d)}$, $\mathrm{r}((\textbf{set}; i)\langle d\rangle) = \boxed{{}^{i}\,\mathrm{r}(d)}$, etc.)

This function will produce a document which depicts every word and annotation. However, if the user is not trying to analyze the annotations, they can make the document cluttered and harder to read. Thus, we define a second function to extract the subdocument which consists only of formatting instructions (elements of $\mathcal{F}$). Recall that the annotation process is an additive one: when annotations are introduced, no information from the original document is lost. Accordingly, this function, which strips out all labels from $\mathcal{G}$ or $\mathcal{S}$, will produce a document that is equivalent to the original before any annotation work took place.

Definition 6.8 (Extract original document)**.** Usually, some $d \in \mathcal{D}$ consists of a "typical" mathematical text plus some information which is stored in the labels from $\mathcal{G} \cup \mathcal{S}$. For any document which has this property, it may

be useful to filter d with the function $\mathrm{od} : \mathcal{D} \to \mathcal{D}_{\mathcal{F}}$, defined as

$$\mathrm{od}([]) = [] \tag{OD1}$$

$$\mathrm{od}(\ell\langle d\rangle) = \begin{cases} \ell\langle \mathrm{od}(d)\rangle & \text{if } \ell \in \mathcal{F} \\ d & \text{otherwise} \end{cases} \tag{OD2}$$

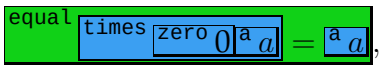
$$\mathrm{od}(d_1, d_2) = \mathrm{od}(d_1), \mathrm{od}(d_2). \tag{OD3}$$

It is then possible to obtain the mathematician's original text as $\mathrm{r}(\mathrm{od}(d))$.

The following example shows, given a small MathLang document, how the document may be rendered with annotations (thus depicting the work of the system's user). It also shows how the original document may be extracted, giving a representation of the document before annotation, and rendered to show that the document as originally composed is conserved even when annotations are added.

Example 6.9. In this example, formatting instructions are taken to be from the LaTeX typesetting system. Consider the document d given as

$$\llcorner\$\boxed{0}\$\urcorner\langle(\textbf{stat};\texttt{equal})\langle\llcorner\boxed{0}=\boxed{1}\urcorner\langle[(\textbf{term};\texttt{times})\langle [(\textbf{term};\texttt{zero})\langle 0\rangle, (\textbf{term};\texttt{a})\langle \texttt{a}\rangle]\rangle, (\textbf{term};\texttt{a})\langle \texttt{a}\rangle]\rangle\rangle\rangle.$$

The document will be rendered, $\mathrm{r}(d)$, as 

, while the filtered document $\mathrm{od}(d)$, $\llcorner\$\boxed{0}\$\urcorner\langle\llcorner\boxed{0}=\boxed{1}\urcorner\langle[\llcorner\boxed{0}=\boxed{1}\urcorner\langle 0, a, a]\rangle\rangle\rangle$, would be rendered as $0a = a$.

6.2 More Souring Details

In this section we formally define the souring rewriting rules given in Section 5.2 in terms of the operational semantics introduced in Section 6.1. This section was published previously [31] but has been expanded, with more details.

The motivation for souring is as follows: in the same way that syntactic sugar is added to a formal document to make it easier to read for

$$\begin{aligned}
\texttt{head}\langle d_1, d_2\rangle &\to_{\mathit{list}} d_1, \texttt{head}\langle d_2\rangle && \text{where } \texttt{list} \mathrel{\widetilde{\notin}} d_1\\
\texttt{tail}\langle d_1, d_2\rangle &\to_{\mathit{list}} d_1, \texttt{tail}\langle d_2\rangle && \text{where } \texttt{list} \mathrel{\widetilde{\notin}} d_1\\
\texttt{daeh}\langle d_1, d_2\rangle &\to_{\mathit{list}} d_1, \texttt{daeh}\langle d_2\rangle && \text{where } \texttt{list} \mathrel{\widetilde{\notin}} d_1\\
\texttt{liat}\langle d_1, d_2\rangle &\to_{\mathit{list}} d_1, \texttt{liat}\langle d_2\rangle && \text{where } \texttt{list} \mathrel{\widetilde{\notin}} d_1\\
\texttt{head}\langle g\langle d_1\rangle, d_2\rangle &\to_{\mathit{list}} g\langle \texttt{head}\langle d_1\rangle\rangle, d_2 && \text{where } \texttt{list} \mathrel{\widetilde{\in}} d_1\\
\texttt{tail}\langle g\langle d_1\rangle, d_2\rangle &\to_{\mathit{list}} g\langle \texttt{tail}\langle d_1\rangle\rangle, d_2 && \text{where } \texttt{list} \mathrel{\widetilde{\in}} d_1\\
\texttt{daeh}\langle g\langle d_1\rangle, d_2\rangle &\to_{\mathit{list}} g\langle \texttt{daeh}\langle d_1\rangle\rangle, d_2 && \text{where } \texttt{list} \mathrel{\widetilde{\in}} d_1\\
\texttt{liat}\langle g\langle d_1\rangle, d_2\rangle &\to_{\mathit{list}} g\langle \texttt{liat}\langle d_1\rangle\rangle, d_2 && \text{where } \texttt{list} \mathrel{\widetilde{\in}} d_1\\
\texttt{head}\langle \texttt{list}\langle g\langle d_1\rangle, d_2\rangle, d_3\rangle &\to_{\mathit{list}} g\langle d_1\rangle, d_3\\
\texttt{tail}\langle \texttt{list}\langle g\langle d_1\rangle, d_2\rangle, d_3\rangle &\to_{\mathit{list}} d_2, d_3\\
\texttt{daeh}\langle \texttt{list}\langle d_1, g\langle d_2\rangle\rangle, d_3\rangle &\to_{\mathit{list}} g\langle d_2\rangle, d_3\\
\texttt{liat}\langle \texttt{list}\langle d_1, g\langle d_2\rangle\rangle, d_3\rangle &\to_{\mathit{list}} d_1, d_3
\end{aligned}$$

Figure 6.1 Souring rewriting rules for lists and mapping.

humans, syntax souring is added to natural-language documents to make them easier for a computer to process. (Syntax sugaring makes the text or code better for humans, while syntax souring makes it worse for humans, but better for computers.) Before verifying a document, or processing it with any rules (such as those shown in Section 8.1), we typically *sour* the document by applying a function to the document (read on to Definition 6.13), then further processing the result. An example of a typical souring operation would be to convert $a = b = c$ to $(a = b) \wedge (b = c)$.

The souring system is conceived as a rewriting system, which is a set of rules which match and replace parameterized expressions. The rules of our rewriting system are shown in Figures 6.1, 6.2, and 6.3. A rewriting system can be thought of as a total function which maps some expressions to other expressions, and maps unmatched expressions to themselves. There are three concepts which are valuable properties for a rewriting system to have: compatability, reflexivity, and transitivity. These will make the rewriting system more robust and useful.

Compatibility is the property of the system which states that if some

$$\begin{aligned}
\texttt{hook}\,\langle d\rangle &\rightarrow_{\textit{chain}} d, \texttt{hook-travel}\,\langle d\rangle \\
\texttt{hook-travel}\,\langle d\rangle\,, \texttt{loop} &\rightarrow_{\textit{chain}} d \\
\texttt{hook-travel}\,\langle d_0\rangle\,, d_1, d_2 &\rightarrow_{\textit{chain}} d_1, \texttt{hook-travel}\,\langle d_0\rangle\,, d_2 \\
&\qquad \text{where } \texttt{loop} \mathrel{\widetilde{\notin}} d_1 \\
\texttt{hook-travel}\,\langle d_0\rangle\,, g\,\langle d_1\rangle &\rightarrow_{\textit{chain}} g\,\langle \texttt{hook-travel}\,\langle d_0\rangle\,, d_1\rangle \\
g\,\langle d_1, \texttt{hook-travel}\,\langle d_0\rangle\rangle &\rightarrow_{\textit{chain}} g\,\langle d_1\rangle\,, \texttt{hook-travel}\,\langle d_0\rangle \\
g_1\,\langle d_1\rangle\,, \texttt{shared}\,\langle d\rangle\,, g_2\,\langle d_2\rangle &\rightarrow_{\textit{share}} g_1\,\langle d_1, d\rangle\,, g_2\,\langle d, d_2\rangle
\end{aligned}$$

$$\begin{aligned}
\texttt{position}^i\langle d_1\rangle, \texttt{position}^j\langle d_2\rangle &\rightarrow_{\textit{pos}} \texttt{position}^j\langle d_2\rangle, \texttt{position}^i\langle d_1\rangle \\
&\qquad \text{where } j < i \\
\ell\langle \texttt{position}^1\langle d_1\rangle, d_2\rangle &\rightarrow_{\textit{pos}} \ell\langle d_1, \texttt{cursor}^1, d_2\rangle \\
\texttt{cursor}^i, \texttt{position}^{i+1}\langle d_1\rangle, d_2 &\rightarrow_{\textit{pos}} d_1, \texttt{cursor}^{i+1}, d_2 \\
\ell\langle d, \texttt{cursor}^i\rangle &\rightarrow_{\textit{pos}} \ell\langle d\rangle
\end{aligned}$$

Figure 6.2 Souring rewriting rules for duplicating and rearranging.

expression can be rewritten to another expression, we may also do the same rewriting when the original expression is found embedded in another, larger one. That is to say, containers do not affect the rewriting of expressions.

Definition 6.10 (Compatibility)**.** We define the following compatibility property for a rewriting rule $\rightarrow_n$:

$$d_1, d, d_2 \rightarrow_n d_1, d', d_2 \qquad \text{if } d \rightarrow_n d' \qquad \text{(COMP1)}$$

$$g\langle d\rangle \rightarrow_n g\langle d'\rangle \qquad \text{if } d \rightarrow_n d' \qquad \text{(COMP2)}$$

Note that (COMP2) is restricted to grammatical labels.

The property of *reflexivity* means that in a rewriting system $\rightarrow_n$ any document d may be rewritten to itself. That is, $d \rightarrow_n d$. The *transitivity* property states that if $d_1 \rightarrow_n d_2$ and $d_2 \rightarrow_n d_3$ then $d_1 \rightarrow_n d_3$. Additionally, recall that if φ' is the *closure* of a relation φ with respect to some de-

$$\texttt{fold-right}\langle d_0\rangle \rightarrow_{\mathit{fold}} \texttt{right-travel}\langle d_2\rangle, d_1 \quad \text{where} \begin{cases} d_0 \hookrightarrow_{\mathit{sour}} d_0' \\ \texttt{head}\langle d_0'\rangle \hookrightarrow_{\mathit{list}} d_1 \\ \texttt{tail}\langle d_0'\rangle \hookrightarrow_{\mathit{list}} d_2 \end{cases}$$

$$\texttt{right-travel}\langle d_1, d_2\rangle \rightarrow_{\mathit{fold}} d_1, \texttt{right-travel}\langle d_2\rangle \quad \text{where } \texttt{base} \mathrel{\widetilde{\notin}} d_1$$

$$\texttt{right-travel}\langle g\langle d_1\rangle, d_2\rangle \rightarrow_{\mathit{fold}} g\langle \texttt{right-travel}\langle d_1\rangle\rangle, d_2 \quad \text{where} \begin{cases} g \neq \texttt{base} \\ \texttt{base} \mathrel{\widetilde{\in}} d_1 \end{cases}$$

$$\texttt{right-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} d_2, \texttt{right-travel}\langle d_2\rangle \quad \text{where } \texttt{list} \sqsubset d_1$$

$$\texttt{right-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} \texttt{fold-right}\langle d_1\rangle$$

$$\texttt{fold-left}\langle d_0\rangle \rightarrow_{\mathit{fold}} \texttt{left-travel}\langle d_2\rangle, d_1 \quad \text{where} \begin{cases} d_0 \hookrightarrow_{\mathit{sour}} d_0' \\ \texttt{daeh}\langle d_0'\rangle \hookrightarrow_{\mathit{list}} d_1 \\ \texttt{liat}\langle d_0'\rangle \hookrightarrow_{\mathit{list}} d_2 \end{cases}$$

$$\texttt{left-travel}\langle d_1, d_2\rangle \rightarrow_{\mathit{fold}} d_1, \texttt{left-travel}\langle d_2\rangle \quad \text{where } \texttt{base} \mathrel{\widetilde{\notin}} d_1$$

$$\texttt{left-travel}\langle g\langle d_1\rangle, d_2\rangle \rightarrow_{\mathit{fold}} g\langle \texttt{left-travel}\langle d_1\rangle\rangle, d_2 \quad \text{where} \begin{cases} g \neq \texttt{base} \\ \texttt{base} \mathrel{\widetilde{\in}} d_1 \end{cases}$$

$$\texttt{left-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} d_2, \texttt{left-travel}\langle d_2\rangle \quad \text{where } \texttt{list} \sqsubset d_1$$

$$\texttt{left-travel}\langle d_1\rangle, \texttt{base}\langle d_2\rangle \rightarrow_{\mathit{fold}} \texttt{fold-left}\langle d_1\rangle$$

$$\texttt{map}\langle d\rangle \rightarrow_{\mathit{map}} [] \quad \text{where } \texttt{list} \sqsubset d$$

$$\texttt{map}\langle d_0\rangle \rightarrow_{\mathit{map}} d_1, \texttt{map}\langle d_2\rangle \quad \text{where} \begin{cases} d_0 \hookrightarrow_{\mathit{sour}} d_0', \\ \texttt{head}\langle d_0'\rangle \hookrightarrow_{\mathit{list}} d_1 \\ \texttt{tail}\langle d_0'\rangle \hookrightarrow_{\mathit{list}} d_2 \end{cases}$$

Figure 6.3 Souring rewriting rules for list operations.

sired property, then φ' contains all members of φ plus additional elements which give φ' that desired property.

Notation 6.11 (Reflexive transitive closure)**.** We denote by $\hookrightarrow_n$ the closure of $\rightarrow_n$ with respect to reflexivity and transitivity.

Finally, we are reminded that for any set of rewriting rules, there may be expressions which cannot be rewritten to other expressions, but may only be rewritten to themselves (because of reflexivity). We call any such expression a *normal form*, emphasizing that these are more normal (or stable) than expressions which may be rewritten. We regard normal forms to be highly-desirable endpoints for the rewriting process.

Notation 6.12 (Normal form)**.** We define the n-normal form relative to $\rightarrow_n$ and denote by NF_n the property on a document d such that no $\hookrightarrow_n$ rewriting can be applied to d.

Given these preliminaries and the rewriting rules given in Figures 6.1, 6.2, and 6.3, we may now establish an operational definition for syntax souring which is equivalent to the transformations described in Section 5.2.

Definition 6.13 (Souring)**.** The $souring$ rewriting rule, denoted by $\rightarrow_{sour}$ is defined as $d_0 \rightarrow_{sour} d_4$ where

$$
\begin{array}{ll}
d_0 \hookrightarrow_{share} d_1 & (d_1 \text{ being in a } NF_{share}) \\
d_1 \hookrightarrow_{chain} d_2 & (d_2 \text{ being in a } NF_{chain}) \\
d_2 \hookrightarrow_{pos} d_3 & (d_3 \text{ being in a } NF_{pos}) \\
d_3 \hookrightarrow_{list} d_4 & (d_4 \text{ being in a } NF_{list})
\end{array}
$$

Souring is a rewriting process. We may regard souring as a function $\mathrm{sour} : \mathcal{D} \rightarrow \mathcal{D}_{\mathcal{F} \cup \mathcal{G}}$. The souring of a document is the application of $\hookrightarrow_{sour}$ until NF_{sour} is reached.

Remark 6.14 (Ambiguous interaction of `chain` and `shared`)**.** The document-duplicating souring operations `chain` and `shared` have non-confluent

interaction. Consider the document

$$\ell_1\langle \texttt{hook}\langle \llcorner \texttt{a} \urcorner \rangle \rangle, \texttt{shared}\langle \texttt{loop}\langle\rangle, \texttt{hook}\langle \llcorner \texttt{b} \urcorner \rangle \rangle, \ell_2\langle \texttt{loop}\langle\rangle \rangle$$

If `chain`ing is performed before `sharing`, the resulting document will go through the two steps below.

$$\begin{aligned} &\ell_1\langle \texttt{hook}\langle \llcorner \texttt{a} \urcorner \rangle \rangle, \texttt{shared}\langle \texttt{loop}\langle\rangle, \texttt{hook}\langle \llcorner \texttt{b} \urcorner \rangle \rangle, \ell_2\langle \texttt{loop}\langle\rangle \rangle \\ &\quad \hookrightarrow_{\textit{chain}} \ell_1\langle \llcorner \texttt{a} \urcorner \rangle, \texttt{shared}\langle \llcorner \texttt{a} \urcorner, \llcorner \texttt{b} \urcorner \rangle, \ell_2\langle \llcorner \texttt{b} \urcorner \rangle \\ &\quad \hookrightarrow_{\textit{share}} \ell_1\langle \llcorner \texttt{a} \urcorner, \llcorner \texttt{a} \urcorner, \llcorner \texttt{b} \urcorner \rangle, \ell_2\langle \llcorner \texttt{a} \urcorner, \llcorner \texttt{b} \urcorner, \llcorner \texttt{b} \urcorner \rangle \end{aligned}$$

Otherwise, the souring rewriting process will proceed as follows, with the `sharing` rules applied first, then the `chain` rewriting performed.

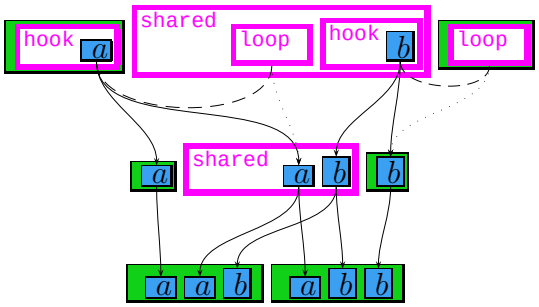
$$\begin{aligned} &\ell_1\langle \texttt{hook}\langle \llcorner \texttt{a} \urcorner \rangle \rangle, \texttt{shared}\langle \texttt{loop}\langle\rangle, \texttt{hook}\langle \llcorner \texttt{b} \urcorner \rangle \rangle, \ell_2\langle \texttt{loop}\langle\rangle \rangle \hookrightarrow_{\textit{share}} \\ &\quad \ell_1\langle \texttt{hook}\langle \llcorner \texttt{a} \urcorner \rangle, \texttt{loop}\langle\rangle, \texttt{hook}\langle \llcorner \texttt{b} \urcorner \rangle \rangle, \ell_2\langle \texttt{loop}\langle\rangle, \texttt{hook}\langle \llcorner \texttt{b} \urcorner \rangle, \texttt{loop}\langle\rangle \rangle \\ &\qquad \hookrightarrow_{\textit{chain}} \ell_1\langle \llcorner \texttt{a} \urcorner, \llcorner \texttt{a} \urcorner, \llcorner \texttt{b} \urcorner \rangle, \ell_2\langle \llcorner \texttt{b} \urcorner, \llcorner \texttt{b} \urcorner, \llcorner \texttt{b} \urcorner \rangle \end{aligned}$$

Although this is an interesting corner case, it is a heterogeneous use of the souring rules which may not be seen in the wild. It is hard to think of a case where such mixed annotations would be helpful. Indeed, any such usage may be disambiguated by the use of `chain` annotations alone, eliminating `shared` from the document entirely. In every case that the latter might be useful, an equivalent annotation may be performed using `chain` rules. We include `shared` because there are situations where it seems more natural than using the pair `hook` and `loop`, but in cases of ambiguity it may be better to stick to one kind of duplicating annotation.

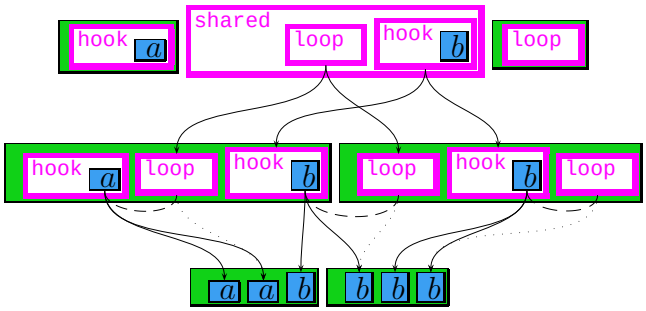
Before we leave this topic, here is the same issue illustrated using the denotational boxes employed in Section 5.4. The reader may find this example easier to read, and it may be illustrative of the relationships between the coloured denotational semantics and the above operational semantics.

Again, we have an expression which has been annotated with grammatical and souring boxes. On our first attempt, we apply the `chain`

souring first, then the `sharing` souring rules.



This results in an annotation which has an equal number of a boxes and b boxes. On the other hand, if we start with the exact same annotated document, but perform `shared` before `chain`, we end up with a different normal form:



Comparing this with the result of the previous souring effort, we see that the resulting expression has a different structure than the last. (More b boxes than a, among other things.) The actual expression that is produced is currently left up to the implementation, as it is believed that this kind of conflict is unlikely to be encountered in the annotation of actual documents. It suffices for the moment for the user to avoid such convoluted expressions, opting instead for simpler ones.

6.3 Adding TSa to the implementation

Before developing the concept of syntax souring and incorporating it into TSa, there was already in existence a plug-in for TEXMACS which allowed the user to perform the annotations for CGa. Furthermore, within the

MathLang type checker there were already the rules which are necessary for operating on normal grammatical annotations. These existing implementations were done by Manuel Maarek. They operate under a client/server relationship, with the TEXMACS plugin passing the annotated document to the type checker, which is running as a server. The server performs the souring and certain kinds of CGa-specific verification, then passes back the document to the plug-in client, possibly with feedback indicating the presence of errors in the annotation. The server/checker was originally written by Maarek in OCaml, but has more recently been reimplemented in Java by Christoph Zengler.

To provide the facilities in TEXMACS for using the souring annotations, we made several additions to the code. First, in the plug-in, there is a style sheet which describes the rendering of various MathLang annotations. We added the macros for rendering the new souring annotations to this style sheet. Secondly, in Scheme source code files, we defined menu items and keybindings for the user's benefit in performing annotations.

Finally, we added syntax souring rules defined in Section 6.2 were added to the OCaml code of the MathLang type checker. It is at this point, and not in the TEXMACS editor, that the souring takes place. As a result, at the current time it is very difficult for a user to verify (for sanity) the result of a souring annotation. Adding feedback to this end would be a valuable addition to the user interface. Such a feature would allow the user to see the annotation before and after souring, rather than simply trusting that the annotations are soured in the way the user expects. (The lack of this feature makes it harder to detect bugs, both in the implementation and the user's understanding of the tools provided.) See Section 9.2 for an idea of how these tools are used in practice while computerising a text.

6.4 Review

In this chapter we have seen the details of MathLang's operational semantics, including the formal details of the syntax souring rules which had

been introduced in Chapter 5, and had an overview of some parts of the MathLang implementation which was carried out in this PhD.

Chapter 7

Summary Trees

Up to this point, the tools of MathLang allow a person to take a mathematical text and add information so that the meaning of the document may be more accurately comprehended by a computer. In Chapter 4 we described the ways that information can be added to the document, and the benefits therein. This information is added through annotation. Chapter 5 shows how common idioms can complicate the annotation process and introduces workarounds to fit human expressions to the computer's requirements.

When all is done, and the tools have been utilized to annotate a document, we have a document which is the original text plus some information. This extra information provides details of the document's structure. One of the hard problems for mathematical knowledge management is extracting the structural information from documents as they are normally written by mathematicians. One of MathLang's primary contributions to this field has been to define a sensible expression of document structure, along with straightforward ways to insert the structural information into the document, so that the structural mark-up and the original document coexist.

The previously described components of MathLang (CGa, DRa, and TSa, including syntax souring) together provide the tools to develop this product: a document, written for humans, which is enriched by struc-

tural information that a computer may operate upon. Once we have that product, there are many interesting places we may wish to go. In this dissertation, we focus on two. The first, which we describe in this chapter, is simply to better comprehend (as humans) the structure which is exhibited in the document. The second is to translate the document into the language of a theorem prover.[1]

Our approach to the second goal is described in Chapter 8. For the moment, in this chapter, we focus on the goal of comprehending structure. What follows is an approach to make it easier for a human to comprehend the structure of the document. In our approach, we regard the document as a tree and create summaries of document trees. These summaries are crafted so that some aspects of the document structure are emphasized, and others are elided.

Annotated documents often exhibit self-similar structure, although the structure varies between one part of the document and the next. If a certain part of the document is highly self-similar, it is likely that this part of the document may be dealt with using only one or two recursive rules. When the structure changes, however – even to a *different* self-similar pattern – we will probably need different rules to translate the new structure that is being expressed.

The tree summary algorithms described in the following sections operate on document annotations to eliminate basic self-similarity so that the viewer of the summarised tree may focus on the places where the structure changes. With changes noted, it is easier to gain an idea of the number of rules which may be required to cover the translation of the document. Furthermore, if one compares the summary trees for two different documents, it may also be possible to note where the rule sets that would cover the respective documents might overlap. This is useful when the set of existing rules covers one document, but not another. Then one can see which parts of the new document are covered by the existing rules, and which parts require new rules.

The material in this chapter is previously unpublished.

[1] In our case, we choose Isabelle as the target for such translation.

7.1 Documents as Trees

When talking about a *tree*, we mean a directed, connected graph which satisfies certain additional properties. Several trees are visible in Figure 8.5. A tree has nodes and edges. An *edge* connects two nodes and has a direction; the node towards which the edge points is called the *head* of the edge, and the other node is called the *tail*. When the tree has any nodes, it has a distinguished node which we call the root. This *root* node has the property that it is the tail of all attached edges. Every node which is not the root node is the head of exactly one edge; it is the tail of any other edge which may be attached. Finally, a node may have a *name*, which is a non-unique identifier for the node.

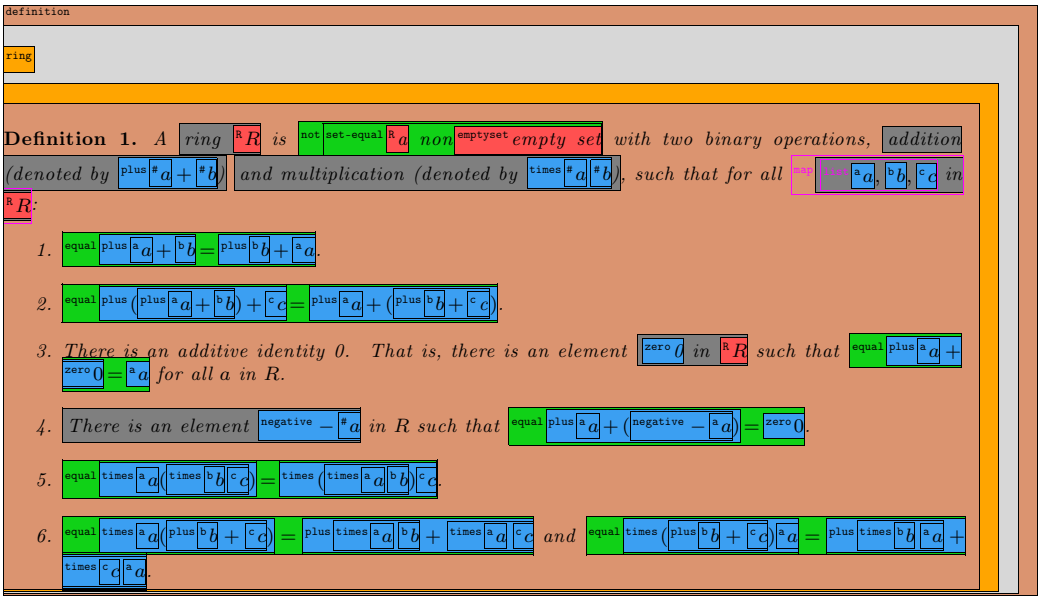
Looking at the definition of MathLang documents in Section 6.1, we see that the recursive nature of the definition lends itself very handily to being regarded as a tree. However, there are many nodes in the original tree which are not necessary for us to consider when looking at the structure expressed in MathLang annotations. Thus, before doing any kind of summarizing operations, the following function is used to extract just the grammatical labels. Refer to Sections 6.1 and 6.2 for notation details.

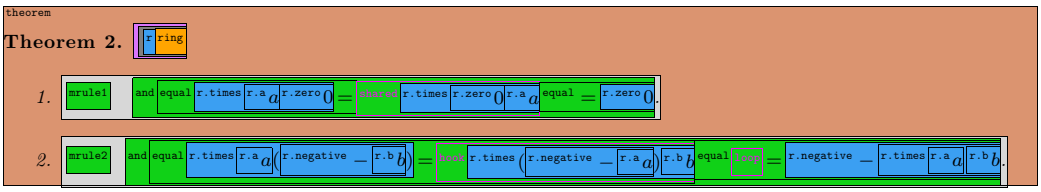
Definition 7.1 (Grammatical subtree)**.** Let $\mathrm{gst} : \mathcal{D} \to \mathcal{D}_{\mathcal{C}}$ be defined as:

$$\mathrm{gst}([]) = [] \tag{GST1}$$
$$\mathrm{gst}((c;i)\langle d\rangle) = c\langle \mathrm{gst}(d)\rangle \tag{GST2}$$
$$\mathrm{gst}(f\langle d\rangle) = \mathrm{gst}(d) \tag{GST3}$$
$$\mathrm{gst}(s\langle d\rangle) = \mathrm{gst}(\mathrm{sour}(s\langle d\rangle)) \tag{GST4}$$
$$\mathrm{gst}(d_1, d_2) = \mathrm{gst}(d_1), \mathrm{gst}(d_2) \tag{GST5}$$

Note that this function yields an element of $\mathcal{D}_{\mathcal{C}}$, with labels only from $\mathcal{C}$, the set of grammatical labels. Depending on how the author annotated the original document, the resulting document may correspond to a single tree, or it may be more akin to a sequence of disjoint trees. As we shall see in Section 8.3, this is not so important for our purposes. The process

Rings





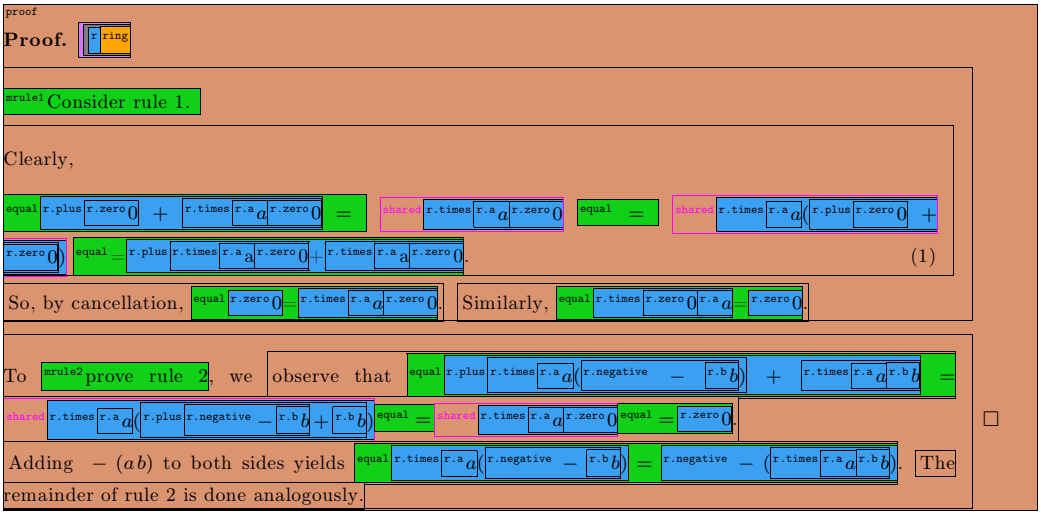


Figure 7.1 An annotated document.

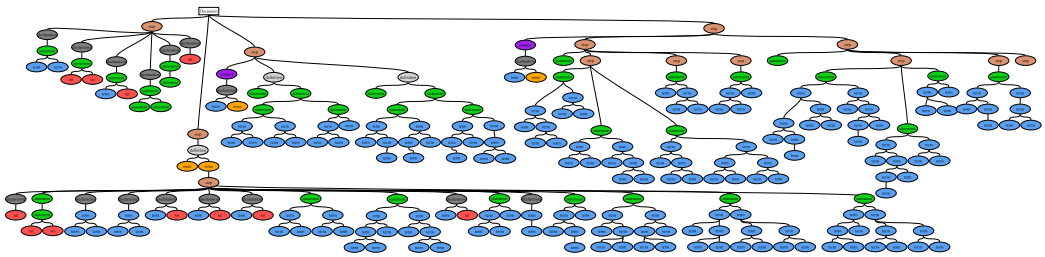


Figure 7.2 The annotations from Figure 7.1 depicted as a tree.

of using summary trees to identify needed rules is more concerned with tree fragments than the whole trees, themselves. Since translation rule creation is very much a manual process at this time, the summary trees are created to provide an alternate view of the tree for human eyes. When creating these rules, a human will be particularly interested in looking at small clusters of nodes (which will, in isolation, constitute a tree) to see what the relationships are between specific nodes and their descendants.

7.2 Summary Operations

When we create a summary of a tree, we are consolidating nodes in a specific way. There are two summary operations defined. One consolidates sibling nodes, and the other consolidates parents and children.

Definition 7.2 (Horizontal summary)**.** Let the function $\mathrm{hs} : \mathcal{D} \to \mathcal{D}$ be defined in the following way:

$$\mathrm{hs}([]) = [] \tag{HS1}$$

$$\mathrm{hs}(\ell\langle d_1\rangle, d, \ell\langle d_2\rangle) = \mathrm{hs}(\ell\langle d_1, d_2\rangle, d) \tag{HS2}$$

$$\mathrm{hs}(d_1, d_2) = \mathrm{hs}(d_1), \mathrm{hs}(d_2) \tag{HS3}$$

$$\mathrm{hs}(\ell\langle d\rangle) = \ell\langle \mathrm{hs}(d)\rangle \tag{HS4}$$

Intuitively, we see that this function takes a sequence of documents and concatenates any sub-documents which have identical labels, labeling the new sub-document with the same label as was used before. To see what

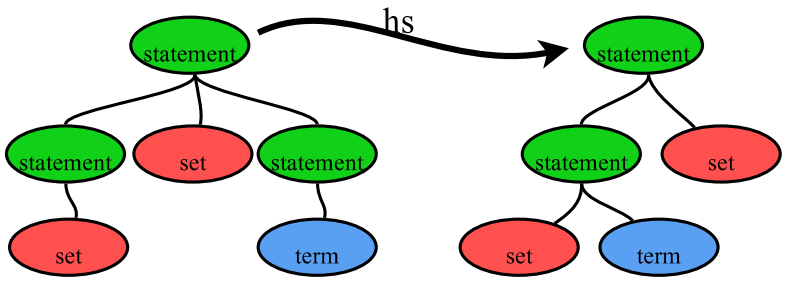


Figure 7.3 Horizontal summary function hs example

the function does, consider the following example. Given the MathLang document (shown with a simple representation of the corresponding tree)

$$\textbf{stat}\langle\textbf{stat}\langle\textbf{set}\rangle, \textbf{set}, \textbf{stat}\langle\textbf{term}\rangle\rangle$$ 

we regard it as the tree on the left-hand side of Figure 7.3. The root node is a **stat**, with three children. Two of its children are **stat**, as well. These are combined into one **stat**, which inherits the **set** and **term** children of the former **stat** nodes. This results in the following document and tree:

$$\textbf{stat}\langle\textbf{stat}\langle\textbf{set}, \textbf{term}\rangle, \textbf{set}\rangle$$ 

Figure 7.3 shows how this transition works with a more detailed view of the trees involved. Figure 7.5 shows another example of hs in action.

So, this is the way horizontal summaries are created. When looking at any given node in the horizontal summary of a tree, we know that for every child node, the original node had one or more children with that label. This is useful because certain classes of CGa expression can be processed with translation rules which operate on nodes that have children with a variety of grammatical types. Regardless of the number of arguments, or the order the types appear in, the translation rule behaves in a similar way. See Section 8.1 for examples of this, particularly rules (DEFn), (TYn), (PFXn), and (PFn).

While horizontal summaries do pare down the tree, a human in the process of creating translation rules for a large document may benefit from

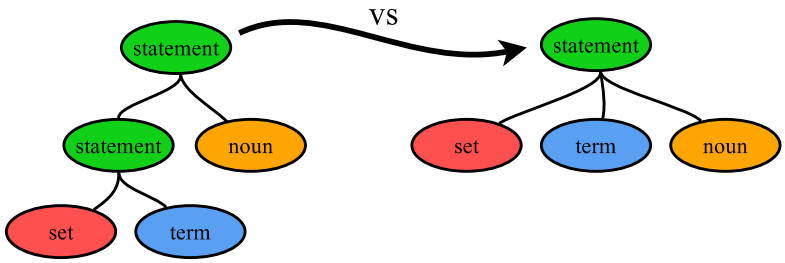


Figure 7.4 Vertical summary function vs in action

even more consolidation. In this event, they may find the following summary helpful.

Definition 7.3 (Vertical summary)**.** Let the function $\mathrm{vs} : \mathcal{D} \to \mathcal{D}$ be defined in the following way:

$$\mathrm{vs}([]) = [] \tag{VS1}$$
$$\mathrm{vs}(\ell\langle d_1, \ell\langle d\rangle, d_2\rangle) = \mathrm{vs}(\ell\langle d_1, d, d_2\rangle) \tag{VS2}$$
$$\mathrm{vs}(d_1, d_2) = \mathrm{vs}(d_1), \mathrm{vs}(d_2) \tag{VS3}$$
$$\mathrm{vs}(\ell\langle d\rangle) = \ell\langle \mathrm{vs}(d)\rangle \tag{VS4}$$

As hs consolidates sibling nodes, the function vs identifies nodes which have the same label as the parent node. It then removes the child and inserts the child's children as children of the parent. By way of example, consider this document (with tree representation):

$$\textbf{stat}\langle \textbf{stat}\langle \textbf{set}, \textbf{term}\rangle, \textbf{noun}\rangle$$ 

Looking at the tree, we see that the root node is a **stat**, and it has **stat** and **noun** for children. When calculating the vertical summary, the child **stat** is deleted, and its children (**set** and **term**) are inserted in its stead. This gives us the document (with tree):

$$\textbf{stat}\langle \textbf{stat}\langle \textbf{set}, \textbf{term}, \textbf{noun}\rangle\rangle$$ 

The whole operation is shown in Figure 7.4. Another example can be seen

in Figure 7.5.

Finally, whenever summaries are discussed, it is common for the tree to be summarized both vertically and horizontally. For ease of discussion, we establish the following notion. When we say "summary tree" without qualification, we mean the tight summary:

Definition 7.4 (Tight Summaries)**.** By *tight summary* we mean the result of a horizontal summarisation of a vertical summary of a tree. That is, given a tree t, the tight summary of t is $\mathrm{hs}(\mathrm{vs}(t))$. This produces a tree which is usually much smaller than either a horizontal or vertical summary alone, and is usually smaller than the original tree. Figure 7.5 depicts an example of a tight summary.

Note 7.5. It is believed to be true that $\mathrm{hs}(\mathrm{vs}(\mathrm{hs}(t))) = \mathrm{hs}(\mathrm{vs}(t))$ for any tree t, but this has not been proved.

7.3 Algorithms for Programming

When using summary trees in a programming environment where the trees to be summarised are already expressed as trees (rather than an abstract MathLang document), it is helpful to have an algorithm to create the summaries, particularly in the event that one is using a procedural programming language which does not have convenient facilities for pattern-matching. The following section provides such algorithms, along with proofs that they achieve the summary properties we seek.

Algorithm 1 describes a function HORIZSUMMARY which creates a horizontal summary of a tree. To start, `tree` is the tree that we want to summarise, and the `summary` tree is the container for the summary we are creating. This algorithm should create a summary of the subtree of `tree` which lies under the node called `parent`. The initial `parent` should match some node in `tree`. The initial `children` should be the nodes which are actually children of this matched node.

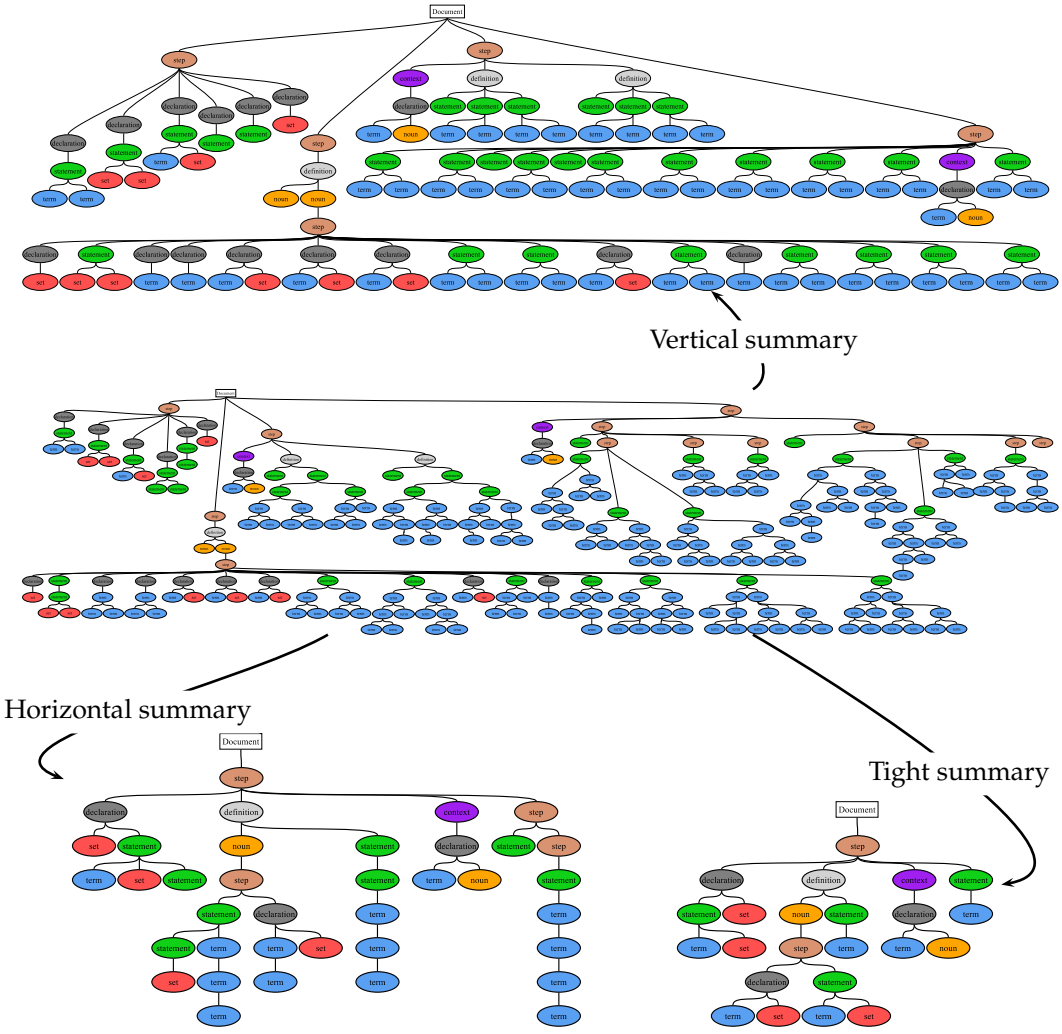


Figure 7.5 Various summaries of the tree shown in Figure 7.2.

Algorithm 1 HORIZSUMMARY(`tree`, `summary`, `parent`, `children`)

Require: `tree`: tree to summarise, `summary`: a tree, `parent`: node from `summary`, and `children`: list of nodes.
Ensure: Tree which is horizontal summary of `tree`.

```
Let used be a list of nodes mapped from node classes.
Let grandchildren be a list of lists of nodes mapped from node classes.
for all child ∈ children do
    if child does not equal any of the used nodes then
        Let new be a fresh node st. new=child.
        Add new to used nodes under its class.
        Add a list of child's children to grandchildren under child's
        class.
    else
        Let name be the class of the current child.
        Let gc be the list in grandchildren mapped from name.
        Add this child's children to gc.
for all node ∈ used do
    Add node to summary as child of parent
    Let name be the node class.
    Let gcs be the list of grandchildren mapped under name.
    Execute HORIZSUMMARY(tree, summary, node, gcs)
return summary.
```

Lemma 7.6. Given a tree T, for any node $n \in$ HORIZSUMMARY(T) no two immediate children of n will have the same name.

Proof. In each iteration of the algorithm, one node called `parent` is considered and the list `used` is constructed from nodes which are immediate children of `parent`. Iterating over these children, a node is added to `used` iff it does not have the same name as any of the nodes in `used`. Therefore, no two nodes in that list will have the same name. Since `used` is duplicated to form the children in the tree HORIZSUMMARY(T) which we are constructing, we conclude that for any node in HORIZSUMMARY(T), there are no two immediate children with the same name. □

Algorithm 2 describes a function VERTSUMMARY which creates a vertical summary of a tree. It will make a summary of a subtree of the original.

Algorithm 2 VERTSUMMARY(`tree`, `summary`, `parent`, `children`)

Require: `tree`: tree to summarise, `summary`: a tree, `parent`: node from `summary`, and `children`: list of nodes.
Ensure: Tree which is vertical summary of `tree`.

```
for all child ∈ children do
    Let name be the class of the child.
    if parent = child then
        for all gc which is a child of child do
            Add gc to children.
            Remove child from children.
    else
        Let newnode be a new node st. newnode = child.
        Add newnode to summary as child of parent.
        Get gcs be the nodes which are children of child.
        Execute VERTSUMMARY(tree, summary, newnode, gcs).
return summary.
```

The initial summary should be empty except for one node, the initial parent. The initial parent should match the root of the subtree you wish to summarise (usually the root of the entire tree). The initial children should be the actual children from the original of that root.

Lemma 7.7. Given a tree T, there is no node n in VERTSUMMARY such that n has the same name as any of its children.

Proof. When deciding if the children of a node $n \in T$ should be duplicated in VERTSUMMARY(T), a child is duplicated iff its name is not the same as n. If the names in T are not the same, then the names in VERTSUMMARY(T) will also not be the same. Rather, if they are the same then the child is purged from consideration for VERTSUMMARY(T) and its own children's names are compared with that of n as if they were children of n. In the same way, these 'adopted' children are included in VERTSUMMARY(T) if their names differ from n, but are otherwise ignored, next comparing their own children with n. Thus every child node, either adopted or original, whose name is the same as n's, is deleted. Therefore n will have no child in VERTSUMMARY(T) whose name matches its own. □

7.4 Review

Summary trees reduce the complexity of MathLang documents so that important hierarchical trends in the document structure are easier for a human to note. These trends are valuable when creating the translation rules described in Chapter 8 because it is important for the set of rules to be comprehensive, covering the entire document.

Chapter 8

Rules for Translating Documents

Section 6.1 described the operational representation of MathLang documents. The documents are stored as an assembly of labels, each of which has a particular role (formatting, grammatical, or souring). These documents are natural-language text and formulae which have been annotated to make the structure of the text more explicit. While the annotations, themselves, are useful for certain kinds of verification and comprehension of the text, the mathematician can use this information to reformat the document into another form. In this section, we give a set of translation rules which could be recursively applied to a MathLang document, and easily extended to be applied to other documents. This set of rules converts some of the information of the document to Isabelle syntax. Sections 8.1 and 8.2 were published in [31].

8.1 Rules for conversion to Isabelle

In this section, we describe a set of rules which are sufficient to translate the document in Figure 9.3 to the language of Isabelle. These are given to show how rules can be created to cover different cases in MathLang documents. Suppose that d is a document which has been soured (see Definition 6.13). Then we apply translation rules $\mathcal{T} : \mathcal{D}_{\mathcal{G}} \to \mathcal{D}_{\mathcal{F}}$ which are mutually recursive as partial translations of d into the syntax of Isabelle.

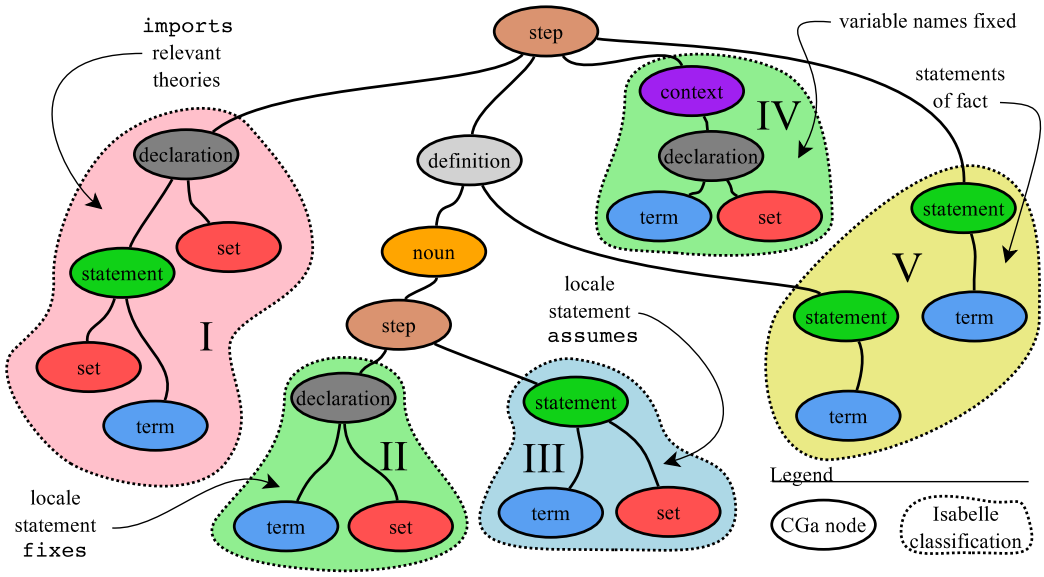


Figure 8.1 Tree depicting kinds of CGa node hierarchies from Figure 9.3, with suggested translations to Isabelle.

These are defined from the top down: each rule may rely on other rules which are defined later in the section. Figure 8.3, in Section 8.2, shows the translation given by the rules.

In the first rule, `(*name*)` is an Isabelle comment which should be replaced with a name for the theory. Similarly, `(*theories*)` is a list of other theories which contain required prior knowledge. Constructing this list of theories is a problem which this PhD. does not attempt to address. For the current work, we leave this task to an Isabelle expert, to fill in the blanks. The root document tree may be translated by the following rule.

$$\mathcal{T}_{\text{root}}(d) = \text{fill}(\llcorner\texttt{theory (*name*) imports (*theories*) begin 0 end}\urcorner, [\mathcal{T}_{\text{main}}(d)]) \quad \text{(ROOT1)}$$

This inserts the main frame for the theory and then invokes the (MAINn) rules shown in figure 8.2. When the main text contains a **definition** annotation surrounding **noun**s, this kind of annotation may be translated with the (DEFn) rules shown in Figure 8.2.

$$\mathcal{T}_{\text{main}}\left(\boxed{\texttt{preface}\ d}\right) = \llcorner\urcorner \tag{MAIN1}$$

$$\mathcal{T}_{\text{main}}\left(\boxed{\texttt{definition}\ d}\right) = \mathcal{T}_{\text{def}}(d) \tag{MAIN2}$$

$$\mathcal{T}_{\text{main}}\left(\boxed{\texttt{theorem}\ \boxed{i\ []}\ \boxed{i'\ []}, d}\right) = \mathcal{T}_{\text{thm}}(i', d) \tag{MAIN3}$$

$$\mathcal{T}_{\text{main}}\left(\boxed{\texttt{proof}\ d}\right) = \mathcal{T}_{\text{pf}}(d) \tag{MAIN4}$$

$$\mathcal{T}_{\text{main}}(d_1, d_2) = \mathcal{T}_{\text{main}}(d_1) \bullet \mathcal{T}_{\text{main}}(d_2) \tag{MAIN5}$$

$$\mathcal{T}_{\text{def}}\left(\boxed{i\ \boxed{i\ []}\ \boxed{i'\ \texttt{props}\ d}}\right) = \text{fill}\left(\llcorner\texttt{locale 0=1}\urcorner, [i, \mathcal{T}_{\text{def}}(d)]\right) \tag{DEF1}$$

$$\mathcal{T}_{\text{def}}\left(\boxed{\boxed{i\ []}\ \boxed{i'\ []}}\right) = \text{fill}\left(\llcorner\texttt{fixes 0::"'r" assumes "0:1"}\urcorner, [i, i']\right) \tag{DEF2}$$

$$\mathcal{T}_{\text{def}}\left(\boxed{i\ d}\right) = \text{fill}\left(\llcorner\texttt{fixes 0::"1"}\urcorner, \left[i, \mathcal{T}_{\text{ty}}\left(\boxed{i\ d}\right)\right]\right) \tag{DEF3}$$

$$\mathcal{T}_{\text{def}}\left(\boxed{i\ []}\right) = \text{fill}\left(\llcorner\texttt{fixes 0::"'r set"}\urcorner, [i]\right) \tag{DEF4}$$

$$\mathcal{T}_{\text{def}}\left(\boxed{i\ d}\right) = \text{fill}\left(\llcorner\texttt{assumes "1"}\urcorner, \left[\mathcal{T}_{\text{pfx}}\left(\boxed{i\ d}\right)\right]\right) \tag{DEF5}$$

$$\mathcal{T}_{\text{def}}(d_1, d_2) = \mathcal{T}_{\text{def}}(d_1) \bullet \mathcal{T}_{\text{def}}(d_2) \tag{DEF6}$$

Figure 8.2 Rules for translating various parts of a document.

Within (DEFn), we have two additional rules referenced, $\mathcal{T}_{\text{ty}}$ and $\mathcal{T}_{\text{pfx}}$. The former gives the type signature for an expression while the latter gives the prefix (Polish notation) interpretation of an annotation's identifiers. We define them one at a time.

For $\mathcal{T}_{\text{ty}}$ we use $\boxed{i\ d} \in \left\{\boxed{i\ d}, \boxed{i\ d}\right\}$ (term or set) where $i \in \mathcal{I}, d \in \mathcal{D}$. As stated, this rule extracts the type signature for the given expression.

$$\mathcal{T}_{\text{ty}}\left(\boxed{i\ d}\right) = \text{fill}\left(\llcorner\texttt{0=>1}\urcorner, \left[\mathcal{T}_{\text{ty}}(d)\,, i\right]\right) \tag{TY1}$$

$$\mathcal{T}_{\text{ty}}\left(\boxed{i\ d}, d'\right) = \text{fill}\left(\llcorner\texttt{0=>1}\urcorner, \left[\mathcal{T}_{\text{ty}}(d')\,, \mathcal{T}_{\text{ty}}\left(\boxed{i\ d}\right)\right]\right) \tag{TY2}$$

$$\mathcal{T}_{\text{ty}}\left(\boxed{i\ []}\right) = \llcorner\texttt{'r}\urcorner \qquad \mathcal{T}_{\text{ty}}\left(\boxed{i\ []}\right) = \llcorner\texttt{'r set}\urcorner \tag{TY3}$$

Example 8.1. When (DEF3) is applied to the annotated expression

the result of the translation is

```
8   fixes   plus   :: "'r => 'r => 'r"
```

where all three of the symbols in the type signature are `'r` because the three inner boxes were all **terms**.

If, on the other hand, we want to convert several boxes – again, for $i \in \mathcal{I}, d \in \mathcal{D}$ we have $\boxed{{}^{i}\ d} \in \left\{\boxed{{}^{i}\ d}, \boxed{{}^{i}\ d}, \boxed{{}^{i}\ d}\right\}$ (term, set, or statement) – the following rules turn the boxes into a prefix notation that is Isabelle-friendly, although it is not perfect (See Note 8.3, below).

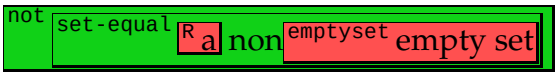
$$\mathcal{T}_{\text{pfx}}\left(\boxed{{}^{i}\ []}\right) = i \tag{PFX1}$$

$$\mathcal{T}_{\text{pfx}}\left(\boxed{{}^{i}\ d}\right) = \text{fill}\left(\llcorner \boxed{0}\ \ \boxed{1} \urcorner, \left[i, \mathcal{T}_{\text{pfx-inner}}(d)\right]\right) \tag{PFX2}$$

$$\mathcal{T}_{\text{pfx-inner}}\left(\boxed{{}^{i}\ []}, d'\right) = \text{fill}\left(\llcorner \boxed{0}\ \ \boxed{1} \urcorner, \left[i, \mathcal{T}_{\text{pfx-inner}}(d)\right]\right) \tag{PFX3}$$

$$\mathcal{T}_{\text{pfx-inner}}\left(\boxed{{}^{i}\ d}, d'\right) = \text{fill}\left(\llcorner (\boxed{0}\ \ \boxed{1})\ \ \boxed{2} \urcorner, \left[i, \mathcal{T}_{\text{pfx-inner}}(d)\ , \mathcal{T}_{\text{pfx-inner}}(d')\right]\right) \tag{PFX4}$$

Example 8.2. We see that the annotated expression



which may then be manipulated to

```
7   assumes "not (set-equal R emptyset)"
```

Note 8.3. It is possible, on a case-by-case basis, to translate expressions such as `emptyset` to the more Isabelle-friendly {}, or even

```
equals zero (times a zero)
```

to

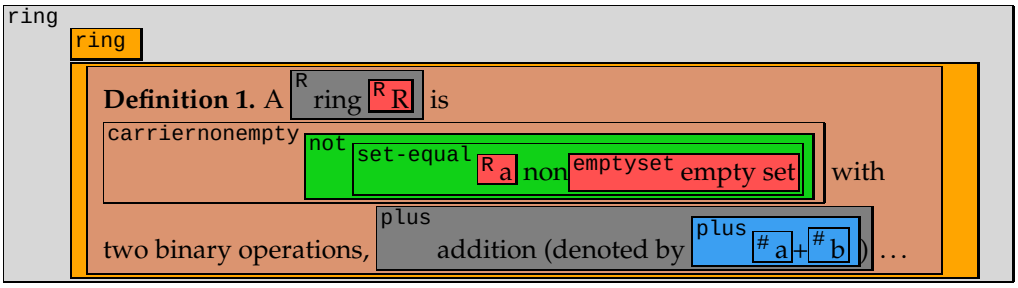
```
zero = a * zero
```

Mathematical concept	Possible MathLang	Isabelle
'Normal' equality	`equals`	*infix* =
Equality of sets	`set-equals`	*infix* =
Empty set	`emptyset`	{}
Set membership	`in`	*infix* :
Logical NOT	`not`	~
Logical AND	`and`	&
Universal quantifier	`forall`	ALL...,...

Table 8.1 Concepts which may be mapped to common Isabelle symbols.

but this kind of automated translation may not be useful or even desirable for the user. We leave it, for the moment, to future work.

Example 8.4. To illustrate the way that $\mathcal{T}_{\text{def}}$, $\mathcal{T}_{\text{ty}}$, and $\mathcal{T}_{\text{pfx}}$ work together, note



would be translated into

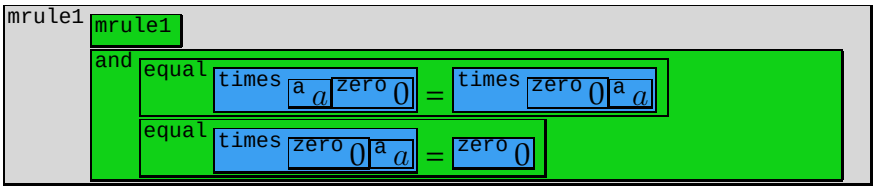
```
locale ring =
  fixes   R :: "'r set"
  assumes "not (set-equal R emptyset)"
  fixes   plus  :: "'r => 'r => 'r"
```

The final rules, $\mathcal{T}_{\text{thm}}$ and $\mathcal{T}_{\text{pf}}$, are for translating parts of the document where theorems are defined and proofs are expressed. The first is a **definition** containing two **statement**s. This first inserts the code framing the theorem statement, then inserts a prefixed interpretation of the annotations identifiers, as with (DEFn).

$$\mathcal{T}_{\text{thm}}\left(p, \boxed{{}^{i}[]\ {}^{i}d}\right) =$$
$$\text{fill}(\llcorner\texttt{theorem (in 0) 1: "2"}\urcorner, [p, i, \mathcal{T}_{\text{pfx}}\left(\boxed{{}^{i}d}\right)]) \quad \text{(THM1)}$$

$$\mathcal{T}_{\text{thm}}(p, (d_1, d_2)) = \mathcal{T}_{\text{thm}}(p, d_1) \bullet \mathcal{T}_{\text{thm}}(p, d_2) \quad \text{(THM2)}$$

Example 8.5. If the above rule is applied to the theorem in Figure 9.3,



it would result in the output

```
28 theorem (in ring) mrule1:
29 shows "and (equal (times a zero) (times zero a))
30            (equal (times zero a) zero)"
```

The final rules are filled in as follows. We note that in Isabelle, theorems pass their locale information on to their associated proof. Thus, although we see the declaration of a ring as context for both theorems and definitions (as denoted $\boxed{r\ \texttt{ring}}$ in Figure 9.3), and this is necessary for MathLang's internal type checking, we do not need this information in the translation. Thus, (PF1) returns an empty string.

$$\mathcal{T}_{\text{pf}}\left(\boxed{d}\right) = \llcorner\urcorner \quad \text{(PF1)}$$
$$\mathcal{T}_{\text{pf}}\left(\boxed{{}^{i}d}\right) = \text{fill}\left(\llcorner\texttt{have "0"}\urcorner, \left[\mathcal{T}_{\text{pfx}}\left(\boxed{{}^{i}d}\right)\right]\right) \quad \text{(PF2)}$$

Example 8.6. This will translate $\boxed{\texttt{equal zero 0} = \texttt{times a zero 0}}$ to the code

```
40 have "equal zero (times a zero)"
```

```
theory (* name *)
imports (* theories *)
begin

locale ring =
  fixes   R :: "'r set"
  assumes "not (set-equal R emptyset)"
  fixes   plus  :: "'r => 'r => 'r"
  fixes   times :: "'r => 'r => 'r"
  fixes   a :: "'r"
  assumes "a : R"
  fixes   b :: "'r"
  assumes "b : R"
  fixes   c :: "'r"
  assumes "c : R"
  assumes "equal (plus a b) (plus b a)"
  assumes "equal (plus (plus a b) c) (plus a (plus b c))"
  fixes   zero :: "'r"
  assumes "zero : R"
  assumes "equal (plus a zero) a"
  fixes   negative :: "'r => 'r"
  assumes "equal (plus a (negative a)) zero"
  assumes "equal (times a (times b c)) (times (times a b) c)"
  assumes "equal (times a (plus b c)) (plus (times a b) (times a c))"
  assumes "(times (plus b c) a) (plus (times b a) (times c a))"


theorem (in ring) mrule1:
shows "and (equal (times a zero) (times zero a))
           (equal (times zero a) zero)"

theorem (in ring) mrule2:
shows "and (equal (times a (negative b)) (times (negative a) b))
           (equal (times (negative a) b) (negative (times a b)))"

have "equal (plus zero (times a zero)) (times a zero)"
have "equal (times a zero) (times a (plus zero zero))"
have "equal (times a (plus zero zero))
            (plus (times a zero) (times a zero))"
have "equal zero (times a zero)"
have "equal (times zero a) zero"

have "equal (plus (times a (negative b)) (times a b))
            (times a (plus (negative b) b))"
have "equal (times a (plus (negative b) b)) (times a zero)"
have "equal (times a zero) zero"
have "equal (times a (negative b)) (negative (times a b))"

end
```

Figure 8.3 Isabelle code created using rules from Section 8.1 on annotations in Figure 9.3.

8.2 Resulting Code

With the aid of the rules from Section 8.1, the Isabelle code in Figure 8.3 may be constructed (again based on the annotations of the small ring theory in Figure 9.3). The rules described in this section are sufficient to translate the document given in Figure 9.3 to Isabelle syntax, and even to get the user very close to a formal proof sketch, but the rules as defined are only sufficient for an extremely small subset of examples. It is not difficult to find a new document for which the translation rules give us an Isabelle-like text which is an insufficient representation of the original mathematics. The translation shown in Figure 8.3 shows several specific drawbacks:

1. The document does not successfully pass through the Isabelle system for several reasons. There are some trivial things, like the theory name on Line 1, which are simple to add but are not easily provided by an intelligent system.

2. Providing the list of imported theories, also, is difficult for a person who does not know the existing libraries nor how to search them for relevant information.

3. The main failure of the resulting locale definition is the form of expressions such as equality. On a case-by-case basis, such things could be converted (in the case of `equal` and `set-equal`, an infix '=' would satisfy Isabelle nicely), but it is hard to say that such transformations would be generally useful without being highly context-sensitive.

4. The relationship between theorems and proofs is not ordered well. In the original text, it makes perfect sense for the author to write what are essentially two theorems, then prove them in the same order. However, Isabelle requires proofs to directly follow their assertions, and the fact that lines 36–41 should be moved just after line 30 is not addressed well. It may not even be immediately evident to the human eye that it is these lines, exactly, which should be associated

with theorem `mrule1`. There is the smaller matter that these proofs should be surrounded with `proof` ... `qed` pairs, but this issue goes hand-in-hand with the aforementioned problem of discerning which proof lines go with which theorem.

The major hurdle, however, is that for Isabelle to find this theory correct, it requires much more information. None of the proof claims (`have "..."`) are justified, and there are significant holes in the reasoning. This is largely due to the fact that the original author simply left many holes which would be evident to a human reader, considering them unnecessary. When this theory file is developed to a point at which Isabelle is completely satisfied, it is approximately 10 times longer. See Section 9.8 for an example of the formalism with details filled in.

While these problems are significant, we believe that the current end-result has merit. One of the major benefits is that this can be performed by a mathematician who knows little-to-nothing about Isabelle. The (very incomplete) theory in Figure 8.3 can then be given to an Isabelle expert for development into a robust theory. This way, they have a starting point in Isabelle syntax, which may save them time in understanding the intent of the document. This is a different way of operating than the status quo (shown in Chapter 3), where an Isabelle expert takes the original natural language text and converts it by hand directly to an Isabelle theory.

8.3 Extending the Translation Rule Set

MathLang cannot automatically discover the rules that are used to translate documents from MathLang annotations to Isabelle syntax. The rules need to be crafted manually by comparing annotated documents with target formal Isabelle documents. In order to write good rules, it is necessary to be very familiar with the parts of Isabelle which are used in the target document. Once good rules have been written, other users of MathLang may use these to generate skeletons for Isabelle documents while knowing little-to-nothing about Isabelle, itself.

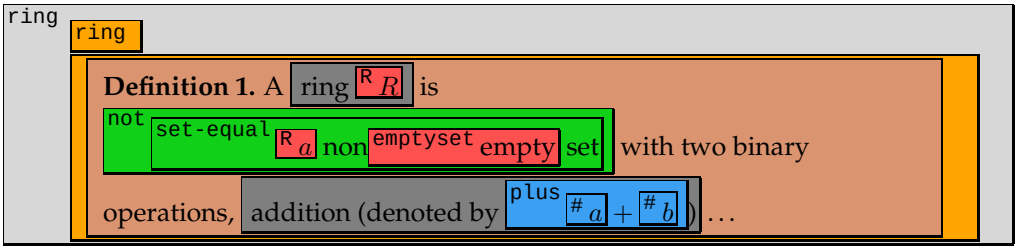


Figure 8.4 Annotated excerpt.

After inspection of the annotations and the target, the rule creator may see a way to rewrite the annotations into the desired Isabelle expression, resulting in a partial rule set. The rule set may be expanded by inspecting more of the document. As the rule set grows, rules may be consolidated into more general ones. The end result should be a set of rules which can be applied recursively to the annotations, automatically producing a document in the syntax of the target system, which can then be filled in (by hand, at this point) without too much alteration to make a formal document. This section is previously unpublished.

To see how these rules might be created, we look at a small excerpt of our sample document shown in Figure A.1:

> **Definition 1.** A ring R is a nonempty set with two binary operations, addition (denoted by $a + b$)...

This is a small sample, which can be seen with CGa annotations in Figure 8.4. We want to create rules which use these annotation boxes as input, producing text that is something like the following code, taken from the listing in Section 9.8:

```
5  locale ring =
     fixes                R :: "'r set"
7    assumes nonempty:    "R \<noteq> {}"
     fixes                plus  :: "'r => 'r => 'r"
```

We start small. First, look at the boxes R which annotate the phrase "ring R" in the text and compare it to the line, **fixes** `R :: "'r set"`. We know from experience with Isabelle that the `"'r set"` part is the type for

the symbol R, specifically a set parameterized by an arbitrary type. The red box R has the grammatical type **set**, with similar semantics, so we might try defining a translation function $\mathcal{T}$ in the following way.[1]

$$\begin{aligned}
\mathcal{T}\left([]\right) &= \llcorner\urcorner \\
\mathcal{T}\left(\boxed{^{i}\,[]}\right) &= \llcorner\texttt{'r set}\urcorner \\
\mathcal{T}\left(\boxed{^{i}\,d}\right) &= \text{fill}\left(\llcorner\texttt{fixes 0 :: "1"}\urcorner, \left[i, \mathcal{T}\left(\boxed{^{i}\,d}\right)\right]\right)
\end{aligned}$$

So far, so good. Next, look further down Figure 8.4 and compare the annotations plus # + # marking "addition (denoted by $a + b$)" with the formal line of Isabelle code, **fixes** `plus :: "'r => 'r => 'r"`. This is very similar to the previous annotation, so we should be able to modify the rules so that they produce the appropriate code no matter which annotation is fed in.

First, note that the type `'r => 'r => 'r` is the type of a function with two parameters, the final `'r` being the type of the function's output, the first two the type of the parameters. In CGa annotations, an identifier of # means that an unspecified parameter is being provided. Given this information, we can create new rules to make the required translation.

$$\begin{aligned}
\mathcal{T}\left([]\right) &= \llcorner\urcorner \\
\mathcal{T}\left(\boxed{^{i}\,[]}\right) &= \llcorner\texttt{'r}\urcorner \\
\mathcal{T}\left(\boxed{^{i}\,d}\right) &= \text{fill}\left(\llcorner\texttt{fixes 0 :: "1"}\urcorner, \left[i, \mathcal{T}\left(\boxed{^{i}\,d}\right)\right]\right) \\
\mathcal{T}\left(\boxed{^{i}\,d}\right) &= \text{fill}\left(\llcorner\texttt{0 => 1}\urcorner, \left[\mathcal{T}\left(d\right), \mathcal{T}\left(\boxed{^{i}\,[]}\right)\right]\right) \\
\mathcal{T}\left(d, \boxed{^{i}\,[]}\right) &= \text{fill}\left(\llcorner\texttt{0 => 1}\urcorner, \left[\mathcal{T}\left(d\right), \mathcal{T}\left(\boxed{^{i}\,[]}\right)\right]\right)
\end{aligned}$$

Applied to the annotation plus # + #, this should give us the code we desire. Now, looking at these two sets of rules, it may be possible to combine them in a natural way. If we use the box $\boxed{^{i}\,d}$ to indicate either $\boxed{^{i}\,d}$ or $\boxed{^{i}\,d}$ (whichever is currently applicable), we may obtain the following rule

[1] See Section 6.1 for definitions of various notations.

set, which covers both the previous annotations and gives the appropriate code for either case.

$$\mathcal{T}\left(\boxed{^{i}\,[]}\right) = \llcorner\texttt{'r set}\urcorner$$
$$\mathcal{T}\left(\boxed{^{i}\,[]}\right) = \llcorner\texttt{'r}\urcorner$$
$$\mathcal{T}\left(\boxed{^{i}\,d}\right) = \text{fill}\left(\llcorner\texttt{fixes 0 :: "1"}\urcorner, \left[i, \mathcal{T}\left(\boxed{^{i}\,d}\right)\right]\right)$$
$$\mathcal{T}\left(\boxed{^{i}\,d}\right) = \text{fill}\left(\llcorner\texttt{0 => 1}\urcorner, \left[\mathcal{T}\left(d\right), \mathcal{T}\left(\boxed{^{i}\,[]}\right)\right]\right)$$
$$\mathcal{T}\left(d, \boxed{^{i}\,[]}\right) = \text{fill}\left(\llcorner\texttt{0 => 1}\urcorner, \left[\mathcal{T}\left(d\right), \mathcal{T}\left(\boxed{^{i}\,[]}\right)\right]\right)$$
$$\mathcal{T}\left([]\right) = \llcorner\urcorner$$
$$\text{where } \boxed{^{\texttt{i}}\,d} \in \left\{\boxed{^{\texttt{i}}\,d}, \boxed{^{\texttt{i}}\,d}\right\}$$

Note that in the third rule, the expression $\mathcal{T}\left(\boxed{^{i}\,d}\right)$ may be applied to either the annotation $\boxed{^{i}}$, where $d = []$, or $\boxed{^{i}\,\boxed{\#}\,\boxed{i}}$, where $d = \boxed{\#}\,\boxed{i}$. The rules are matched in the order they are defined to resolve any ambiguity in the definition.

Thus, we have created rules which may be used to translate two small parts of the sample document. Further work along similar lines will allow a person to eventually build up a rule set like the one described in Section 8.1, which can be applied to an entire document.

8.4 Review

The translation rules described in this chapter are an example of how CGa annotations can be converted into the languages of theorem provers to provide a starting point for full formalisation of mathematical documents. We may consider this set of translation rules as a proof-of-concept for the exercise; much work is still needed to create a set of translation rules which would cover a variety of mathematical documents, and it is not certain that any one collection of rules would be adequate for broad collections of mathematical topics. These represent one possible use out of many for

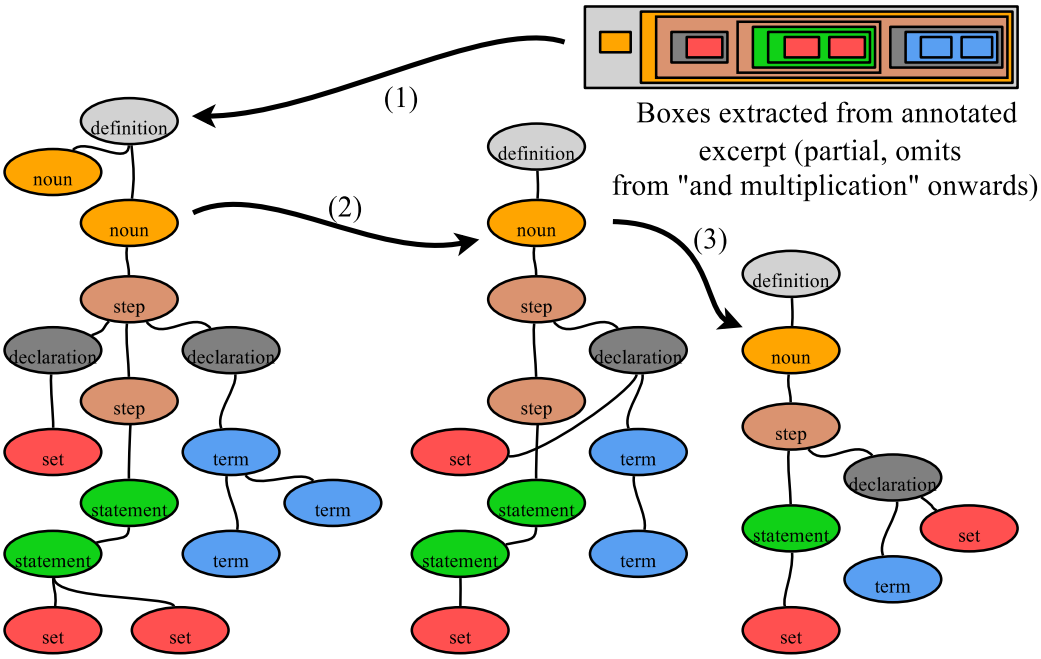


Figure 8.5 Tree and summaries corresponding to excerpt in Figure 8.4.

MathLang annotations. The discovery of appropriate translation rules is not trivial. It is a process that takes insight and requires significant knowledge of the target system (e.g., Isabelle).

Chapter 9

A Stroll down the Path of Document Computerisation

MathLang is a large framework designed to solve a large problem: computerising mathematical texts. In order to achieve this, and to accommodate the needs of different people who have different computerisation needs, the process of computerising is broken down into a number of steps, organised into aspects. The steps are described in detail in Chapters 4, 5, 7, and 8. In each chapter, the descriptions were augmented with small examples. This chapter pulls these examples together into a cohesive whole.

This chapter describes working examples, from a human-friendly sample text taken all the way to fully-verified formal Isabelle code. The example is shown in its different states, and it is possible to see the steps needed to take the example from one state to the next. We discuss the usefulness of each step as we walk through the experience of computerizing each mathematical document.

9.1 Original Text

The following text was taken from an undergraduate textbook on abstract algebra [12], from chapter 12, entitled "Introduction to Rings". It took

this author about 13 minutes to type up the example in a suitable form for using with MathLang. Upon examination, the reader may note that it consists of three major parts: the definition of this algebraic object, a small theorem about multiplication, and a proof of the theorem. Noting these divisions will help the discussion which follows. While reading, take note of which parts seem obvious to you as a reader, and which require some effort to follow. Take a moment to consider what information might need to be added for a computer to follow the reasoning.

Definition 9.1 (Ring)**.** A *ring* R is a non-empty set with two binary operations, addition (denoted by $a+b$) and multiplication (denoted by ab), such that for all a, b, c in R:

1. $a + b = b + a$.
2. $(a + b) + c = a + (b + c)$.
3. There is an additive identity 0. That is, there is an element 0 in R such that $a + 0 = a$ for all a in R.
4. There is an element $-a$ in R such that $a + (-a) = 0$.
5. $a(bc) = (ab)c$.
6. $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$.

Our first theorem shows how the operations of addition and multiplication intertwine.

Theorem 1 (Rules of Multiplication)**.** Let a, b, and c belong to a ring R. Then

1. $a0 = 0a = 0$.
2. $a(-b) = (-a)b = -(ab)$.

Proof. Consider rule 1. Clearly,

$$0 + a0 = a0 = a(0 + 0) = a0 + a0.$$

So, by cancellation, 0=a0. Similarly, 0a=0.

To prove rule 2, we observe that $a(-b) + ab = a(-b + b) = a0 = 0$. So, adding $-(ab)$ to both sides yields $a(-b) = -(ab)$. The remainder of rule 2 is done analogously. □

We consider this to be a typical sample of mathematical writing. There is a definition wherein a kind of object is defined, bestowed with certain kinds of structure via a list of constraints. Knowledge about the structure is extended with a theorem, and the assertions of the theorem are justified with an argument that uses the constraints in the definition. Certain knowledge and ability is assumed of the reader: "sets," "binary operators," were probably defined earlier in the text, or they may be assumed to be common knowledge. Both items (1) and (2) of the theorem are chained equalities, and in both cases the second half is left as an excercise to the reader. The parts of the proof which are written down are compact and implicitly use transitivity of equals, cancellation, and other shortcuts that the reader is expected to see without assistance.

These are all reasonable assumptions to make for a human reader who has already completed part of an undergraduate curriculum and has already progressed through 11 chapters of the same textbook. But what happens when the text is given to a computer? The machine will have all kinds of trouble, from parsing the different formulae and parts of the text (Is the numbered list part of the definition? What about the sentence after it?) to comprehending scope (Is the a in the first part of the definition the same a as in item 3? What is it that $-(ab)$ is added to, in the proof?) and understanding just how the arguements in the proof are justified by the definition (out of the 6 items in the definition, which ones are needed to support the proof of rule 2?).

These are only scratching the surface of the complexities of computerizing mathematical documents. Furthermore, the issues will vary depending on why the document is being computerized. Is the computer helping with verification? Accessibility? Navigation? Teaching? Publishing? Calculation? The less ambiguous a document's encoding, the easier it will be

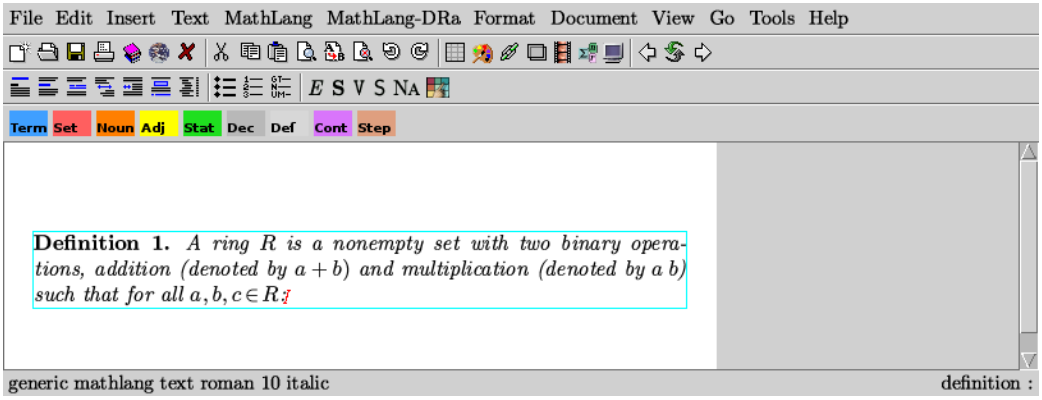


Figure 9.1 Typing the beginning of a document into TEXMACS

to meet these needs. MathLang is designed to help encode the document and then move it in one or more of these directions. In the following sections, we will demonstrate how the existing MathLang system is used on this particular example.

9.2 Adding CGa Annotations to a Document

As the name suggests, the Core Grammatical aspect (CGa) is designed to describe the grammatical structure of mathematical content in a document. This means, as described in Section 4.2, that the annotations made in this aspect will primarily be made at the sentence-level and smaller. It is normal for single words and individual variables to be annotated. Annotating this example, both with the kinds of annotations described in this section and also Section 9.3, took 65 minutes.

For an author to use the facilities of CGa in a document, A reasonable work flow might be as follows. We have written a plugin for the TEXMACS scientific word processor. This allows a user to interact with the document, add and visualise annotations, and send the document with annotations to a server for correctness checking. Informal tests have shown so far that the environment is straightforward for new users: it is easy to learn to use, and the different functions of the system are clear.

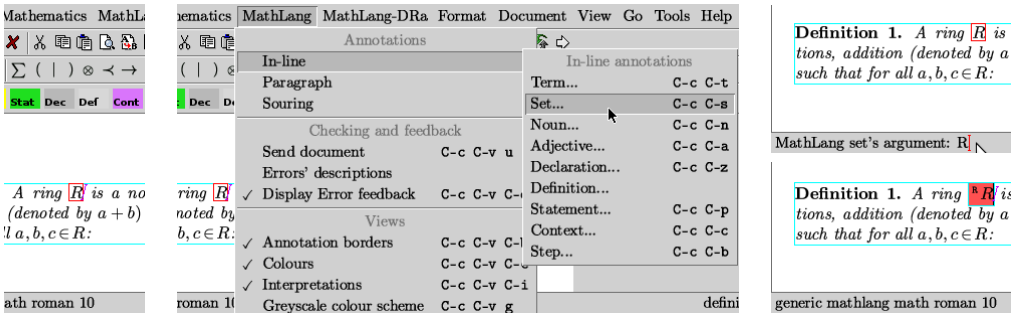


Figure 9.2 Annotation steps: select variable, choose annotation type from menu, enter interpretation argument, and the annotation is complete.

An author annotating a document might proceed in the following way. While it is possible to annotate a document after the composition of the human-friendly text is complete, we will show a document being annotated as it is written down. We start by writing the beginning of the definition, as in Figure 9.1.

There are parts of this sentence which should be annotated. It is important to note that the annotation is a computer-friendly expression of the author's intent, so there is often more than one 'correct' annotation for a given document. To annotate part of the sentence, we select the text in TEXMACS, then choose the appropriate type of annotation using the plugin. The plugin has menus and keybindings, and for certain common annotations, there are buttons on the TEXMACS toolbar.

In our current example, we start small and expand. Let us begin by annotating a single variable. We choose the first one, R, in the example. As can be seen in Figure 9.2, the variable is selected using the normal text-selection features of the operating system. In this case the user clicks on the MathLang menu, chooses Annotations, In-line, and selects Set..., which is an appropriate type for this variable. This leads TEXMACS to prompt the user (in the status bar) for an argument, which should be the interpretation for the variable as was described in Section 4.2. The annotation is then complete. We see the original text, R, surrounded by a box with a red background which contains a superscripted R. The red background indicates the type (set) of the annotation, and the superscripted R is the

interpretation which was entered.

9.3 TSa Concerns

The Text and Symbol aspect has two distinct components at the current time, both of which are designed to accommodate the peculiarities of human mathematical writing. One part, which is described in Section 4.3, describes a method for depicting annotations to a document, and is directly related to the aspect CGa. The other part is syntax souring, which is described in Chapter 5. This part is devoted to unraveling certain kinds of syntactic sugar which are deeply engrained into the writing patterns of mathematicians.

For our current example, the first part is dealt with neatly in Section 9.2. The TEXMACS annotation process uses this part of TSa directly to make the annotation process as clear and straightforward as possible. This part of TSa is tightly interwoven with the method and ethos of MathLang annotation, for CGa if not entirely for DRa. Thus, the previous section provides ample illustration of this part of TSa in practice. We have not, however, seen how this example might utilise syntax souring. Read on.

Up to the point where the example left off in the previous section, the text is nicely suited to annotations for CGa. At the end of the current excerpt, however, we see the following formula: $a, b, c \in R$. This kind of expression is very common. It means that we are introducing three variables, a, b, and c, and they are declared to be members of the set R. This provides a context in which future discussion of these variables is restricted. But this is, in fact, syntactic sugar for the mathematician to mean that $a \in R$, $b \in R$, and also $c \in R$. Because of the way CGa is defined, only the latter, expanded version of the math may be annotated. Accordingly, we need some way of converting the shorter $a, b, c \in R$ into the more verbose $a \in R, b \in R, c \in R$. Since this is all for the benefit of the computer, we to not need to worry about changing the English sentence into something grammatically correct. We only need to create annotations

Rings

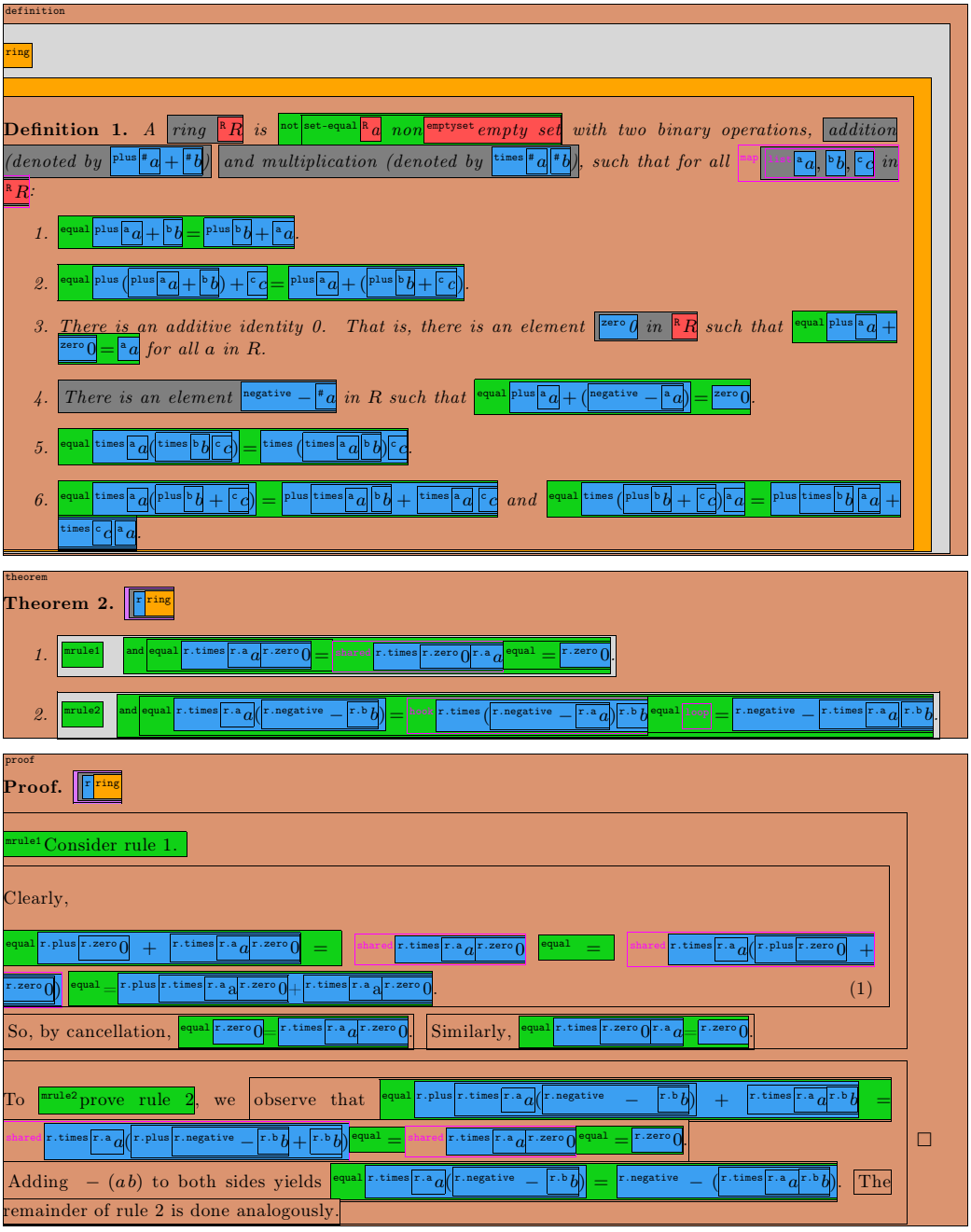
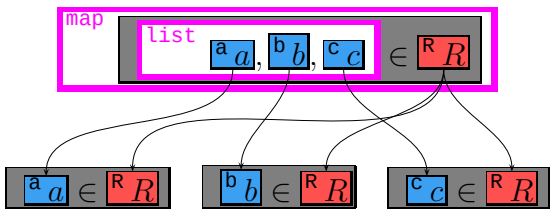


Figure 9.3 Completed CGa annotation of ring theory text

that faithfully convey the meaning of the original mathematics. This is exactly the purpose of souring.

Souring annotations, in the current implementation of MathLang, are slightly different in appearance from the regular CGa annotations. CGa annotations have content, type, and interpretation. The type is conveyed by the colour of the box surrounding the content, and the interpretation is included as superscripted text in the upper-left corner of the annotation box. On the other hand, souring annotations are always depicted with a box of white background and pink outline. The kind of souring being performed is denoted in the upper-left corner, in the same manner as CGa annotations.

An appropriate souring annotation for the expression $a, b, c \in R$ is mapping. The concept of mapping is (as described in Section 5.4) as follows: One has a function f, and a list $\{a, b, \ldots, n\}$, and the function is applied to the elements of the list to produce a list $\{f(a), f(b), \ldots, f(n)\}$. For the souring operation of mapping, we make several assumptions about the function and associated list. First, the function has exactly one parameter, and the parameter is used in a single place in the function. Second, the function is given in an anonymous (unnamed) way with the list provided instead of a parameter. Third, instead of an explicit parameter, the paramater's position is indicated by the presence of the argument list. In other words, in the case of the expression $a, b, c \in R$, the function is $f(x) = x \in R$ and the list is $\{a, b, c\}$. We use the souring annotation list to simultaneously denote the list of arguments and the position of the parameter. Thus, the annotation we choose to use in this circumstance is shown on the following two lines. The first is the annotation as marked by the user, the second is the result of MathLang souring rewriting rules:



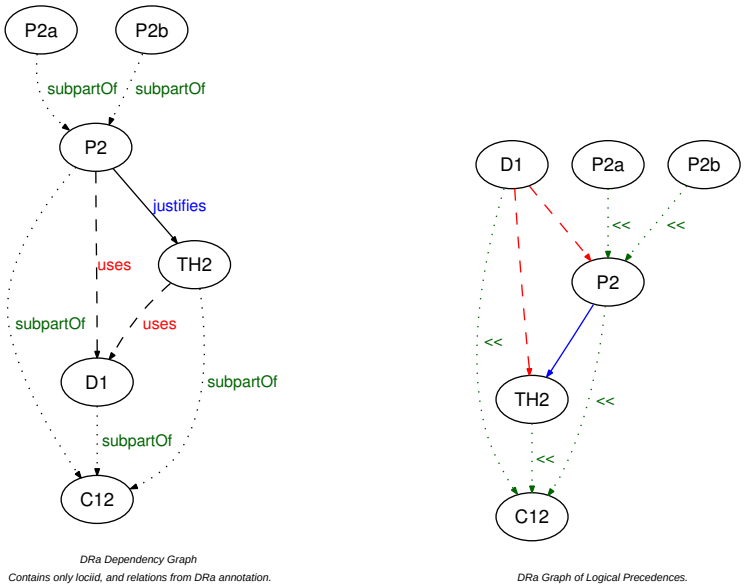


Figure 9.4 Two graphs automatically generated from DRa annotations: The dependency graph (DG) and the graph of logical precedence (GoLP)

This is only one example of how syntax souring is used. For more examples and a more thorough and rigorous treatment of the methods used, see Chapter 5. The `map/list` annotations seen here are useful in other situations, and there are other kinds of souring annotations for use in a variety of scenarios.

9.4 DRa Annotation of Text

The Document Rhetorical aspect of MathLang is used to capture the relationships between different portions of a text as an author builds formulae, definitions, arguments, and examples into a cohesive theory. Processing DRa annotations to a document can flag problems such as circular reasoning and poorly-supported theorems, and aid in the navigation or restructuring of a text. Annotation is performed by identifying interesting components of the document, and then indicating their relationships, one

to another. It takes about 5 minutes to annotate the document in the way described in this section.

The graphs in Figure 9.4 are automatically generated to show relationships within the document according to the DRa annotation. The first graph is the dependency graph which illustrates the relationships as defined by the user. The second is the graph of logical precedence, which is an altered graph illustrating the actual rhetorical structure of the document. Figure 9.5 (the DG of which is equivalent to the graph in Figure 9.4) provides a slightly different view of the dependency graph. By laying each node over the portion of the text that it denotes, it is much easier to see how the structure of the graph is derived.

In the normal workflow of DRa one works to develop the dependency graph, then automatically derives the graph of logical precedence from this. The author proceeds in the following way. Perusing the text, a portion is selected so that it is, in some respect, self-contained. (This may be a single sentence, or an entire chapter. Read on for the kinds of portions we mean.) This portion of text is then given a name. Considering Figure 9.5, the reader may note that the three main parts of the document (definition, theorem, and proof) are each given a short identifier (**D1**, **TH2**, and **P2**, respectively). In this case the user also identified the entire excerpt as a chapter (**C12**), and annotated two halves of the proof (**P2a** and **P2b**) which are focused on separate multiplication rules from the theorem.

Note that not all of the annotations deal directly with the rhetorical structure of the document: the entire body of this example was annotated as a chapter, which is usually considered an organisational (or even typographical) entity rather than a part of the argumentation structure. In fact, there are two classes of narrative entity which in turn characterise *division elements* (or portions of text such as chapters, section, and paragraphs) and *mathematical units* (of which theorem, axiom, and corollary are examples). More information about this distinction may be found in [20].

After this, the user may indicate the relationships between various annotated parts of the text. In our example, the chapter is divided into our three primary parts, and this is indicated by the "`subpartOf`" edge con-

Definition 9.1.

A *ring* R is a nonempty set with two binary operations, addition (denoted by $a + b$) and multiplication (denoted by ab), such that for all a, b, c in R:

1. $a + b = b + a$.
2. $(a + b) + c = a + (b + c)$.
3. There is an additive identity 0. That is, there is an element 0 in R such that $a + 0 = a$ for all a in R.
4. There is an element $-a$ in R such that $a + (-a) = 0$.
5. a(b c)=(a b)c.
6. $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$.

Theorem 1.

Rules of Multiplication Let a, b, and c belong to a ring R. Then

1. $a0 = 0a = 0$.
2. $a(-b) = (-a)b = -(ab)$.

Proof.

Consider rule 1. Clearly,

$$0 + a0 = a0 = a(0 + 0) = a0 + a0.$$

So, by cancellation, $0 = a0$. Similarly, $0a = 0$.

To prove rule 2, we observe that $a(-b) + ab = a(-b + b) = a0 = 0$. So, adding $-(ab)$ to both sides yields $a(-b) = -(ab)$. The remainder of rule 2 is done analogously.

□

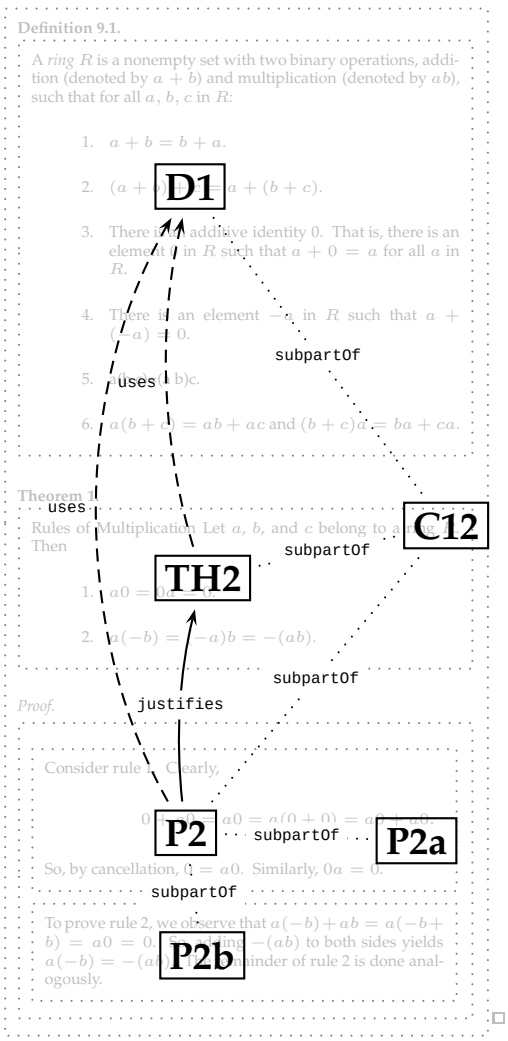


Figure 9.5 Comparison of original (CML) text with a dependency graph.

necting each to **C12**. The main proof **P2** is similarly related to its constituents **P2a** and **P2b**. Finally, it is indicated that **P2** should justify the claim of **TH2**, and that both of these make use of the definition **D1** in some fashion.

The differences between a dependency graph and a graph of logical precedence are minor, but significant. The dependency graph, oft abbreviated DG, is intended to show the relations between parts of a document in the most natural way. By contrast, the graph of logical precedence (or GoLP) is meant to delineate the order of annotated portions of text in a sequence of reasoning steps [20]. The latter is effectively a transformation of the former in which edge labels are rewritten and the direction of certain edges are reversed (namely uses, inconsistentWith and exemplifies). Ultimately, the difference in topology between DG and GoLP provides the ability to check various important properties. The DG, for example, can verify that some justification has been offered for each proof or corollary, and inversely that any axiom has no associated proof. On the other hand, argumentation issues such as circular reasoning may be flagged if the GoLP contains one or more cycles.

9.5 Encoding of Text in Plain MathLang Syntax

Once the document has been encoded using the current aspects of MathLang, the document may be translated to the following. This is the *plain* syntax which was originally created for development of MathLang, before TSa was worked out to provide a more text-oriented way of computerizing documents. The plain syntax encoding can be automatically generated from the annotations made in the previous sections. Under normal circumstances, it would not be used or seen by a user of MathLang, but it can be useful for troubleshooting and making new developments.

The first section of plain encoding establishes a namespace for DRa. It then identifies the node names that will be used for the annotation of the document and assigns a role to each as a division element or mathematical

unit. Thirdly, it records the edges between certain nodes, identifying each of them with the type of relation which is being made.

```
[! default namespace "dra"
    "http://www.macs.hw.ac.uk/ultra/mathlang/document-context"]
[dra:maindesc xml:id="desc1" about="g.mathlang.xml"
  [dra:description about="C12"
      hasStructuralRhetoricalRole="chapter"]
  [dra:description about="D1" hasMathRhetoricalRole="definition"]
  [dra:description about="TH2" hasMathRhetoricalRole="theorem"]
  [dra:description about="P2" hasMathRhetoricalRole="proof"]
  [dra:description about="P2a" hasMathRhetoricalRole="proof"]
  [dra:description about="P2b" hasMathRhetoricalRole="proof"]
  [dra:relation xml:id="r1" src="P2" anch="TH2" type="justifies"]
  [dra:relation xml:id="r2" src="TH2" anch="D1" type="uses"]
  [dra:relation xml:id="r3" src="P2" anch="D1" type="uses"]
  [dra:relation xml:id="r4" src="D1" anch="C12" type="subpartOf"]
  [dra:relation xml:id="r5" src="P2" anch="C12" type="subpartOf"]
  [dra:relation xml:id="r6" src="TH2" anch="C12" type="subpartOf"]
  [dra:relation xml:id="r7" src="P2a" anch="P2" type="subpartOf"]
  [dra:relation xml:id="r8" src="P2b" anch="P2" type="subpartOf"]
]
```

The next few lines are pure CGa. The annotation uses the symbols `equal`, `in` and `and` but does not define them in the text. Since they are not built in to MathLang, they must be declared before the code which corresponds to the original document. This would not be the case if the symbols had been introduced in the original text. In that situation their declarations would be made in a more natural way. The *symbol*(*type-list*):*type* expression is the manner in which **declaration**s (from Table 4.3) are expressed in this syntax. Similarly, {/} pairs indicate a **step** annotation.

```
{
 equal(term,term):stat;
 in(term,set):stat;
 and(stat,stat):stat;
};
```

The following lines of code begin the encoding of the actual text. The first line indicates the start of the node annotation which denotes the entire excerpt as a chapter called "**C12**". Then the definition of a ring (the first of the three main parts) is given, surrounded by a DRa annotation indicating labelling it as "**D1**".

As an example of the correspondence between the following code and

the annotated document shown in Figure 9.3, let us consider the introduction of the ring operation called addition. In the CGa rendering of the annotated document, the introduction is shown as

while the corresponding line in the code below reads

```
plus(term,term):term;
```

The "plus" on the gray **declaration** box, as stated in Section 5.1, is optional. The "plus" in the plain syntax line comes from the interpretation of the blue **term** box. Within this box, there are two others which have "#" interpretations, meaning they are placeholders. Accordingly, the parentheses contain term twice. The grammatical category after the colon is the return type of the newly-declared plus symbol.

```
[dra:node dra:nodeId="C12" |
{
 [dra:node dra:nodeId="D1" |
  ring:=Noun {
   R:set;
   plus(term,term):term;
   times(term,term):term;
   a:R;
   b:R;
   c:R;
    equal(plus(a,b),plus(b,a));
    equal(plus(plus(a,b),c),plus(a,plus(b,c)));
  {
    zero:R;
    equal(plus(a,zero),a);
    in(a,R);
  };
  {
    negative(term):term;
    equal(plus(a,negative(a)),zero);
  };
    equal(times(a,times(b,c)),times(times(a,b),c));
    equal(times(a,plus(b,c)),plus(times(a,b),times(a,c)));
    equal(times(plus(b,c),a),plus(times(b,a),times(c,a)));
  }
 ];
```

As the above is the first part of the text, the definition, the next block of code matches with the statement of the theorem, which in the DRa annotation was called **TH2**.

```
[dra:node dra:nodeId="TH2"|
{
  r:ring;
  a:r.R;
  b:r.R;
  "mrule-1":=and(equal(r.times(a,r.zero),r.times(r.zero,a)),
      equal(r.times(r.zero,a),r.zero));
  "mrule-2":=and(equal(r.times(a,r.negative(b)),
      r.times(r.negative(a),b)),equal(r.times(r.negative(a),b),
      r.times(a,b)));
} ];
```

The code is concluded with the plain encoding of the proof of the theorem.

```
 [dra:node dra:nodeId="P2" |
 {
  [dra:node dra:nodeId="P2a" |
  {
   {
     equal(r.plus(r.zero,r.times(r.a,r.zero)),
         r.times(r.a,r.zero));
     equal(r.times(r.a,r.zero),
         r.times(r.a,r.plus(r.zero,r.zero)));
     equal(r.times(r.a,r.plus(r.zero,r.zero)),
         r.plus(r.times(a,r.zero),r.times(a,r.zero)));
   };
     equal(r.zero,r.times(a,r.zero));
     equal(r.times(r.zero,a),r.zero);
  } ];
  [dra:node dra:nodeId="P2b" |
  {
   {
     equal(r.plus(r.times(a,r.negative(b)),r.times(a,b)),
         r.times(a,r.plus(r.negative(b),b)));
     equal(r.times(a,r.plus(r.negative(b),b)),r.times(a,r.zero));
     equal(r.times(a,r.zero),r.zero);
   };
     equal(r.times(a,r.negative(b)),r.negative(r.times(a,b)));
  } ];
 } ];
} ];
};
```

Again, under normal use of the MathLang system, this code format is

never seen or even generated by the user. It is simply a concise format for expressing the CGa and DRa information in as uncluttered a way as possible. However, for exploring the possibilities for future developments, it is sometimes the most straightforward language to work with.

9.6 Translating CGa/DRa Annotations to Isabelle Syntax

Once we have the document annotated in CGa and DRa annotations, it is possible to produce code in the syntax of Isabelle which corresponds to the annotations made in the document. This is done by taking the document tree and applying rules like those described in Chapter 8. The rules specifically mentioned in that chapter are sufficient to translate the current example document to Isabelle syntax, but in order to properly translate a variety of documents, the rule set would need to be expanded. The issue of expanding translation rule sets, along with an approach for creating new rules, is discussed in Chapter 7.

The rules are defined so that the main rule is applied to the root of the document, and the rules are applied recursively on the document tree. The following is the result of applying rules to the ring theory example as annotated in Figure 9.3:

```
theory (* name *)
imports (* theories *)
begin

locale ring =
  fixes   R :: "'r set"
  assumes "not (set-equal R emptyset)"
  fixes   plus  :: "'r => 'r => 'r"
  fixes   times :: "'r => 'r => 'r"
  fixes   a :: "'r"
  assumes "a : R"
  fixes   b :: "'r"
  assumes "b : R"
  fixes   c :: "'r"
  assumes "c : R"
  assumes "equal (plus a b) (plus b a)"
  assumes "equal (plus (plus a b) c) (plus a (plus b c))"
```

```
18    fixes   zero :: "'r"
      assumes "zero : R"
20    assumes "equal (plus a zero) a"
      fixes   negative :: "'r => 'r"
22    assumes "equal (plus a (negative a)) zero"
      assumes "equal (times a (times b c)) (times (times a b) c)"
24    assumes "equal (times a (plus b c)) (plus (times a b) (times a c))"
      assumes "(times (plus b c) a) (plus (times b a) (times c a))"
26

28  theorem (in ring) mrule1:
    shows "and (equal (times a zero) (times zero a))
30             (equal (times zero a) zero)"

32  theorem (in ring) mrule2:
    shows "and (equal (times a (negative b)) (times (negative a) b))
34             (equal (times (negative a) b) (negative (times a b))"

36  have "equal (plus zero (times a zero)) (times a zero)"
    have "equal (times a zero) (times a (plus zero zero))"
38  have "equal (times a (plus zero zero))
               (plus (times a zero) (times a zero))"
40  have "equal zero (times a zero)"
    have "equal (times zero a) zero"
42
    have "equal (plus (times a (negative b)) (times a b))
44              (times a (plus (negative b) b))"
    have "equal (times a (plus (negative b) b)) (times a zero)"
46  have "equal (times a zero) zero"
    have "equal (times a (negative b)) (negative (times a b))"
48
    end
```

The result is a file which is legal Isabelle syntax, but is not formally correct. There are many gaps in the reasoning, and there may be certain parts of the document, particularly in proofs, which are not in the best order for formalization. However, this file may be a good starting point for an Isabelle expert to begin when formalizing the document. In the next section, we show how the process of filling in the theory might proceed.

9.7 Formal Proof Sketch of Ring Theory in Isabelle/Isar

The barriers to entry for a user to begin formalizing theories in most any proof assistant are, first, learning the syntax, general principles, and idiosyncracies of the system; second, learning what the system requires for justification; and third, learning where all these justifications may be found in the bowels of the assistant's library of mathematical theories. In automatically generating Isabelle syntax, we hope to aid a new user in gaining familiarity with the idioms of the new system. The other two barriers, it is generally agreed, require a deeper knowledge of the system.

The idea of a proof sketch was first put forth by Freek Wiedijk in the context of Mizar, and he soon expanded the idea to Isabelle in collaboration with Marcus Wenzel [54]. We describe them in more detail in Section 2.3, but for the present context, understand that formal proof sketches are defined, essentially, to be formal documents which permit certain errors. Specifically, errors relating to justification.

The following is a translation of the above code into the language of Isabelle/Isar. It is a complete formal proof sketch as discussed in Section 2.3 and also [54]. To facilitate the translation we took hints from the text which are similar to those described in [19]. Filling in the details from the text shown in Section 9.6 took about 12 minutes.

```
5  locale ring =
     fixes                R :: "'r set"
7    assumes nonempty:   "R \<noteq> {}"
     fixes                plus  :: "['r,'r] => 'r"
9    fixes                times :: "['r,'r] => 'r"
     assumes pluscomm:   "[| a:R; b:R |] ==> plus a b = plus b a"
11       and plusassoc:  "plus (plus a b) c = plus a (plus b c)"
     fixes                zero :: "'r"
13   assumes zeroinR:    "zero : R"
         and zeroisid_r: "a:R ==> plus a zero = a"
15   fixes                negative :: "'r => 'r"
     assumes negcancels: "a:R ==> plus a (negative a) = zero"
17   assumes timescomm:  "[| a:R; b:R; c:R |] ==>
                             times a (times b c) =
19                             times (times a b) c"
         and distleft:   "[| a:R; b:R; c:R |] ==>
```

```
21                          times a (plus b c) =
                              plus (times a b) (times a c)"
23       and distright:  "[| a:R; b:R; c:R |] ==>
                          times (plus b c) a =
25                          plus (times b a) (times c a)"
    assumes plusclosed: "[| a:R; b:R |] ==> plus a b : R"
27      and timesclosed: "[| a:R; b:R |] ==> times a b : R"
```

The Isar command **sorry** tells Isabelle to skip a proof-in-progress and treat the goal under consideration to be proved. It causes the proof text to be an incorrect document, but allows the user to skip over portions that have not been justified to the satisfaction of the proof checker. As the reader can see, almost every step translated from the MathLang encoding is unsatisfactory and must be left for later justification.

```
47  theorem (in ring)
    assumes "a:R" and "b:R"
49  shows
      "times a zero = times zero a \<and> times zero a = zero
51     \<and>
       times a (negative b) = times (negative a) b
53       \<and> times (negative a) b = negative (times a b)"
    proof −
55      have "plus zero (times a zero) = times a zero"
          sorry
57      also have "... = times a (plus zero zero)"
          sorry
59      also have "... = plus (times a zero) (times a zero)"
          sorry
61      also have "zero = times a zero"
          sorry
63      have "times a zero = zero"
          sorry
65
        have "plus (times a (negative b)) (times a b)
67                                    = times a (plus (negative b) b)"
          sorry
69      have "times a (plus (negative b) b) = times a zero"
          sorry
71      have "times a zero = zero"
          sorry
73
        have "times a (negative b) = negative (times a b)"
75        sorry

77      show ?thesis sorry
    qed
```

9.8 Formalisation of Ring Theory in Isabelle/Isar

The following is the fully-formalised counterpart to the above code. Unlike the formal proof sketch, this document is a sound formalisation in Isabelle/HOL, which completely justifies the assertions made rather than temporarily sidestepping them with **sorry** commands. It has been developed entirely by hand; no automation for this transition has been yet established. This process is still much more intensive than the previous steps, and takes nearly four and a half hours to complete.

The text begins with the definition of a ring through the Isabelle/Isar `locale` construct. It is mostly the same as the definition given in the above sketch, with the addition of one small axiom, `negclosed`, which was not stated in the original text.

```
locale ring =
  fixes                R :: "'r set"
  assumes nonempty:   "R \<noteq> {}"
  fixes                plus  :: "['r,'r] => 'r"
  fixes                times :: "['r,'r] => 'r"
  assumes pluscomm:   "[| a:R; b:R |] ==> plus a b = plus b a"
      and plusassoc:  "plus (plus a b) c = plus a (plus b c)"
  fixes                zero :: "'r"
  assumes zeroinR:    "zero : R"
      and zeroisid_r: "a:R ==> plus a zero = a"
  fixes                negative :: "'r => 'r"
  assumes negcancels: "a:R ==> plus a (negative a) = zero"
  assumes timescomm:  "[| a:R; b:R; c:R |] ==>
                          times a (times b c) =
                            times (times a b) c"
      and distleft:   "[| a:R; b:R; c:R |] ==>
                          times a (plus b c) =
                            plus (times a b) (times a c)"
      and distright:  "[| a:R; b:R; c:R |] ==>
                          times (plus b c) a =
                            plus (times b a) (times c a)"
  assumes plusclosed: "[| a:R; b:R |] ==> plus a b : R"
     and timesclosed: "[| a:R; b:R |] ==> times a b : R"
      and negclosed:  "a:R ==> negative a : R"
```

We omit some minor lemmas to focus on the formalisation of the main result. First, the theorem statement is given. The reader may note that this is precisely the same as the statement from the previous proof sketch.

```
theorem (in ring)
```

```
71  assumes "a:R" and "b:R"
    shows
73    "times a zero = times zero a \<and> times zero a = zero
       \<and>
75     times a (negative b) = times (negative a) b
         \<and> times (negative a) b = negative (times a b)"
```

Naturally, it is the proof of this theorem that sees the most significant changes from proof skeleton to complete formalisation. Effort has been made to make use of the proof elements above, but it is in fact difficult to achieve for several reasons. The proof goal stated above is essentially the conjunction of four statements which, taken together, encapsulate the chained equalities of the original theorem. Below they are proved separately, as facts `A`, `B`, `C`, and `D`, then these facts are used to justify the main proof goal.[1]

The goals are proved out of their originally-stated order because facts `B` and `D` are respectively simpler than `A` and `C`. The former are useful in simplifying the proofs of the latter. One consequence of this is that statements such as `have "times a zero = zero"`, which were present in the formal proof sketch and are (sometimes partially) preserved in the full formalisation, may appear in a different order than that indicated in the proof sketch.

```
    proof −
78    have B [simp]: "times zero a = zero"
      proof −
80      from prems zeroinR timesclosed [of zero a]
            zeroisid_r [of "times zero a"]
82        have "times zero a = plus (times zero a) zero" by auto
        also from prems this zeroinR timesclosed [of zero a]
84          negcancels [of "times zero a"]
          have "... = plus (times zero a)
86          (plus (times zero a) (negative (times zero a)))"
          by auto
88      also from this plusassoc [of "times zero a" "times zero a"
            "negative (times zero a)"]
90        have "... = plus (plus (times zero a) (times zero a))
            (negative (times zero a))" by auto
92      also from prems zeroinR distright [of a zero zero]
          have "... = plus (times (plus zero zero) a)
94                                          (negative (times zero a))"
```

[1] In Isabelle/Isar, the current proof goal may be cited by the keyword `?thesis`.

```
      by simp
96    also from zeroinR zeroisid_l [of zero]
      have "... = plus (times zero a) (negative (times zero a))"
98      by simp
      also from prems zeroinR timesclosed [of zero a]
100       negcancels [of "times zero a"]
        have "... = zero" by simp
102   finally show ?thesis .
    qed
104 have A: "times a zero = times zero a"
    proof −
106   have "plus zero (times a zero)
                              = plus (times a zero) (times a zero)"
108   proof −
        from prems zeroinR timesclosed [of a zero]
110         zeroisid_l [of "times a zero"]
          have "plus zero (times a zero) = times a zero" by auto
112     also from zeroinR zeroisid_l [of zero]
          have "... = times a (plus zero zero)" by auto
114     also from prems zeroinR distleft [of a zero zero]
          have "... = plus (times a zero) (times a zero)" by auto
116     finally show ?thesis .
      qed
118   from prems zeroinR timesclosed [of a zero]
        this plus_cancels [of zero "times a zero" "times a zero"]
120     have "zero = times a zero" by auto
      from this B show "times a zero = times zero a" by auto
122 qed
    have D: "times (negative a) b = negative (times a b)"
124 proof −
      have "plus (times (negative a) b) (times a b) = zero"
126   proof −
        from prems negclosed [of a] distright [of b "negative a" a]
128       have "plus (times (negative a) b) (times a b)
                                      = times (plus (negative a) a) b"
130       by auto
        also from prems negclosed [of a] pluscomm [of "negative a" a]
132         negcancels [of a]
          have "... = times zero b" by auto
134     also have "... = zero"
        proof −   −− "This is slightly modified from the proof for B"
136       from prems zeroinR timesclosed [of zero b]
            zeroisid_r [of "times zero b"]
138         have "times zero b = plus (times zero b) zero" by auto
          also from prems this zeroinR timesclosed [of zero b]
140           negcancels [of "times zero b"]
            have "... = plus (times zero b)
142           (plus (times zero b) (negative (times zero b)))"
            by auto
144       also from this plusassoc [of "times zero b" "times zero b"
```

```
            "negative (times zero b)"]
146       have "... = plus (plus (times zero b) (times zero b))
            (negative (times zero b))" by auto
148     also from prems zeroinR distright [of b zero zero]
          have "... = plus (times (plus zero zero) b)
150                                    (negative (times zero b))"
          by simp
152     also from zeroinR zeroisid_l [of zero]
          have "... = plus (times zero b) (negative (times zero b))"
154       by simp
        also from prems zeroinR timesclosed [of zero b]
156         negcancels [of "times zero b"]
          have "... = zero" by simp
158     finally show ?thesis .
      qed
160   finally show ?thesis .
    qed
162 from this
      have "plus (plus (times (negative a) b) (times a b))
164                                          (negative (times a b))
                  = plus zero (negative (times a b))"
166     by auto
    from this plusassoc
168     have "plus (times (negative a) b) (plus (times a b)
                                        (negative (times a b)))
170                 = plus zero (negative (times a b))"
      by auto
172 from prems this timesclosed [of a b]
      negcancels [of "(times a b)"] negclosed [of a]
174     zeroisid_r [of "times (negative a) b"]
      timesclosed [of "negative a" b]
176   have "times (negative a) b = plus zero (negative (times a b))"
      by auto
178 from prems this timesclosed [of a b] negclosed [of "times a b"]
      zeroisid_l [of "negative (times a b)"]
180   have "times (negative a) b = negative (times a b)" by auto
    from prems this show ?thesis by auto
182 qed
  have C: "times a (negative b) = times (negative a) b"
184 proof −
    have "plus (times a (negative b)) (times a b) = zero"
186   proof −
      from prems negclosed [of b] distleft [of a "negative b" b]
188     have "plus (times a (negative b)) (times a b)
                                    = times a (plus (negative b) b)"
190     by auto
      also from prems this negclosed [of b] negcancels [of b]
192       pluscomm [of "negative b" b]
        have "... = times a zero" by auto
194     also from this A B
```

```
        have "... = zero" by auto
196     finally show ?thesis by auto
      qed
198   from this
        have "plus (plus (times a (negative b)) (times a b))
200                                              (negative (times a b))
                          = plus zero (negative (times a b))"
202     by auto
      from prems this plusassoc timesclosed [of a b]
204       negcancels [of "times a b"]
        have "plus (times a (negative b)) zero
206                                = plus zero (negative (times a b))"
        by auto
208   from this zeroisid_l [of "negative (times a b)"]
        zeroisid_r [of "times a (negative b)"] prems
210     timesclosed [of a b] negclosed [of "times a b"]
        negclosed [of b] timesclosed [of a "negative b"]
212     have "times a (negative b) = negative (times a b)" by auto
      from this D
214     have "times a (negative b) = times (negative a) b" by auto
      from this show ?thesis by auto
216  qed
    from A B C D show ?thesis by auto
218 qed
```

Reflection upon the similarities between this formalisation and the original proof provided in [12] indicates important differences between typical mathematical communication and rigorous formalisation. One of these differences is the way that the original author did not concern himself with noting the transitivity of equality. In the original proof, part of the theorem is stated as "$a0 = 0a = 0$", and the author is satisfied with proving that $a0 = 0$. This is a point that mathematicians are comfortable to overlook, but through which a theorem prover must be guided. Discovery of ways to accommodate this sort of discrepancy of justification requirements will be important in future research.

9.9 Continuing the example

Although this example has been taken from the original text to complete formalisation in Isabelle/HOL, it is still of interest to carry it from the same MathLang encoding through a formal proof sketch for Mizar to a full for-

malisation in Mizar. Translation to Mizar is the original path explored in the development on DRa, so it is important to understand intricacies of this translation process (and the Mizar language itself) before attempting to model future MathLang developments on the work done in DRa.

9.10 Review and Analysis of Effort

In this chapter we have shown a sample text being annotated and processed in a variety of ways. The text was annotated, translated, and ultimately formalized. This represents a fundamental change from more established use of proof assistant systems. To gain much benefit from a system such as Isabelle or Mizar, one must gain a fairly deep knowledge of the system before benefit may be gleaned from one's efforts. In this chapter, a process has been outlined wherein each step provides some incremental benefit to the user. Thus, there are payoffs at each step along the way, not just at the end when the entire document is formalized.

The particular breakdown of the efforts for our current example can be summarized as follows. Given a text in a book, this can be typed in to the computer to obtain a suitable form for computerization in 13 minutes. This gives us a digital, nominally structured, typeset document with sections, formulae, and so on, which is suitable for presentation to human beings. The initial annotation in CGa using the tools of DRa can be done in a little more than an hour. At this stage, the computer can provide the benefits of grammatical type checking, and verifies that the annotations we have provided are reasonable and sane. In a similar vein, DRa annotations are done in only a few minutes, providing us with the opportunity to check the rhetorical structure of the document. Benefits here also include navigation and viewing graphs of the document's structure.

Translation of the document to the syntax of Isabelle is automatic, and gives us a structure in which to build out our formal document. It presents the theory in a natural form for filling in details. Building this up to a formal proof sketch, where Isabelle can verify for us that the syntax and

structure we have used are legitimate, only takes a few minutes, as well. In all, to get it to this point, an Isabelle document which lacks justification details, takes 90 minutes, or 1.5 hours. Filling in this formal proof sketch to produce a complete formal theory takes an additional 4.5 hours, meaning that, using the tools of MathLang, taking the example from textbook to formal document takes 6 hours. Most of this time is spent in filling in the details of proofs. Definitions and theorem statements are roughly satisfactory, but there is far less information in the textbook proof than Isabelle requires to see the validity of the arguments.

Translating the same text directly from the textbook to fully-formal Isabelle takes roughly 6 hours, as well. However, it is worth noting that after only a quarter of this time, the MathLang process will provide the user with specific benefits, but when the user is 90 minutes into the straight-to-Isabelle process, they only have a partial formal document.

It is worth noting that the times given here are obtained from an attempt at the procedure by this author, who is by now quite familiar with the example at hand. It would take a proportionately longer time to annotate and formalize a text which he had not previously examined.

Chapter 10

Concluding Thoughts

MathLang is a project with a very wide scope. The developments in this dissertation are put forth as an incremental step towards the larger goal. We want computers to be more useful to mathematicians. They should aid the user in verifying what is written. Computers should guide the user in comprehending texts, particularly in the context of a larger corpus of work. They should be able to translate and convert documents into whatever form a user requires. Additionally, in those areas where computer facilities are not yet mature, computers should provide guidance and hints to speed the work of a human expert in a task.

10.1 Work Accomplished

In this PhD, there have been several notable developments. Some have been directly building on existing portions of MathLang, while others extend the project in new directions. The primary contributions are as follows, in order of appearance in this dissertation.

Integrating natural-language input with CGa annotation.

When this PhD. was just starting, Manuel Maarek had already completed CGa and was working on TSa. I helped with the development and implementation of the TSa annotation scheme which is described in Section 4.3

and demonstrated in Section 9.2. The annotation method there described allows the user to provide explicit type information directly interleaved with the original natural-language document. Before TSa, the type information had to be composed in a text file separately from the original document. Having them together in this way gives us greater confidence that the computerisation of the document is faithful to the original, as composed by the mathematician.

Tools for making expressions more computer-friendly.

Syntax Souring is presented in Chapter 5 and again in Section 6.2. The methods of syntax souring are designed to accommodate certain expressions used by mathematicians in ordinary writing which are not comprehended easily by existing portions of MathLang. In *souring* these expressions, MathLang is undoing the syntactic sugar which is employed by mathematicians in their every-day writing. We established the procedures of souring, established the appearance of annotations for users to employ souring, and implemented the procedures as a part of the MathLang CGa grammar type checker.

Operational semantics for MathLang.

In Chapter 6 we defined precise notation for the operation of the MathLang system, particularly CGa and TSa. Starting from basic definitions, we define what a MathLang document is, and define a number of functions and relations which should be useful to anyone who wishes to gain a deeper understanding of the system, perhaps for the purpose of implementing its workings.

A tree-compressing method to elide self-similar structure.

When you need to see the structure of a document, rendering it in tree form can be very illuminating. At the same time, these trees can be very

big, with much repetition and self-similar structure. This can make it difficult to locate the places in the tree where interesting structural changes occur. Summary trees are one way of eliding the repetitive parts of the tree to allow a human being to focus on the places where structural patterns change. They were developed during this PhD, and are presented in Chapter 7.

An approach at automatically producing Isabelle code.

The final major contribution to this PhD. is the approach to translating MathLang documents into the language of Isabelle, presented in Chapter 8. Normal use of MathLang creates annotations as described in Sections 4.3, 9.2 and 9.3, producing CGa type information as described in Section 4.2. It is not hard for an Isabelle expert to convert this information into Isabelle code, forming the skeleton for an Isabelle theory. The CGa type information is never enough for a fully-formal Isabelle document, but it can be a good starting point for expansion. We have here presented an example set of rules for automating this MathLang-to-Isabelle translation process, and provide guidelines for creating further rules to cover a wider range of documents.

10.2 Future Work

There are many directions that this project could be extended in the future. Here are a few possibilities.

Adding automation for mundane tasks.

The activity of annotating documents is generally agreed to be a straightforward but tedious task. Smaller mathematical expressions may be frequently repeated, and the annotation of these can be quite tedious when found in large documents. Anecdotal tests have indicated operation of the system is easy to comprehend, but the system may be harder to use

as documents become complex. Users may easily become bogged down in the details of annotation, distracting them from higher-level authoring duties. This sort of thing might be easily automated. In general, further investigation is necessary to ensure that the present – or any future – implementation provides maximal assistance to the mathematical user. Easing the tedium of repeated annotations is part of the goal, but the system as currently implemented could benefit from the insight of an expert in usability.

Implement MathLang in other standard systems.

The current implementation of MathLang makes use of certain internal representations for MathLang documents, but there is nothing special or sacred about the particulars of the format, except in regards to convenience and efficiency. There are established standards, such as OpenMath and OMDoc, which provide means for document representation along with a framework for describing things such as the particulars for CGa, TSa, and DRa. Expressing the existing system in terms of an OpenMath content dictionary [28, §22.3.2] or an OMDoc module [28, Ch. 10] would be a useful step towards building connections between the current MathLang system and other software packages.

Extend TSa to DRa, improve use in CGa.

The annotation of some common mathematical constructions was made possible through the developments of syntax souring [18] and others have been made easier than before. However, between the two other aspects, CGa had an almost exclusive focus. DRa could benefit from input generalisation and disambiguation that has improved the user experience for CGa.

This is currently under implementation. In addition to providing standard ways to display DRa nodes and connections, the implementation of TSa for DRa is planned to include facilities for exploring and display-

Natural language or CGa	Isabelle language feature
definition annotations	locales
declaration annotations	**fix**(**es**) statements
declaration with **context**	**fix**(**es**) or **assume**(**s**) statements
statement annotations	**have** statements
"*This* is denoted by *that*."	pretty-printing syntax declarations

Table 10.1 Possible correspondences between MathLang annotations and Isabelle features.

ing automatically generated DG and GoLP. It remains to be seen whether there are parts of DRa which will require or benefit from the souring rules which have been developed for CGa. Furthermore, it will be important when developing further aspects in MathLang to be attentive to the potential needs that a fledgling aspect will have from the Text and Symbol aspect.

Look for corresponding idioms in MathLang and Isabelle.

In beginning the formalisation of ring theory in Isabelle via MathLang and formal proof sketches, several potential correspondences between CGa and Isar have been noted. They are summarised in Table 10.1.

Develop DRa hints for translation towards Isabelle.

Currently DRa is a system motivated largely by translation of documents to the language of the Mizar proof checker [19]. As such, the facilities for describing the narrative structure of a document are well-developed. However, using these tools to then create a skeleton for a Mizar formalisation is less mature. Presently there is a selection of hints which provide clues for the transformation from a DRa graph to a Mizar skeleton, but these have not been formalised in any way. A point of current development in the MathLang group is to refine these hints into reliable rules with the potential for automation.

In the same way, it seems a fruitful endeavour to develop similar hints

(and eventually rigorous rules) for the transformation of DRa-annotated documents to skeletons for Isabelle/Isar. It will be necessary to examine the existing Mizar hints and clues to see if they can be modified to operate towards Isabelle; or better, if they might be generalised to apply to both languages. An important goal of the MathLang project is to allow the user to avoid committing to any one theorem prover or logical foundation for as far as possible along the computerisation path. Thus, any possible generalisations to accommodate more logics and verifiers will be welcome.

Furthermore, all hints that have been recommended to date for Mizar operate at a rather holistic level, since they take DRa annotations completely into account, but only a subset of CGa. Hints for certain DRa graph features paired with CGa **step** annotations provide guidance for the main proof skeleton while the CGa preamble[1] is used to refine the user's search through the Mizar Mathematical Library to develop an environment for the article.

Develop DRa hints as aspect.

Once a sufficient body of hints and guides from DRa or CGa have been collected, an important next step will be to synthesise them into a new aspect of MathLang. This will involve generalising the new rules as much as possible and proving their suitability in the target language, be it Isabelle/Isar or some other target system. There may be several distinct aspects to be formulated in the space between DRa and Isabelle/Isar. These may focus on such issues as overall narrative structure, more detailed justification requirements, or selecting existing results to use in justification from the existing library.

Develop MathLang library (with support system).

The ring theory example used in this report is a very small and limited example of mathematics. In order to help the MathLang project grow and

[1] The set of symbol declarations at the beginning of the document which are included because the original text does not introduce them.

develop, it is necessary to expand annotation efforts to include samples of mathematics which are very different, and samples which are very large. It is important to experiment with mathematics based on different foundations (ZF set theory or the simply typed λ-calculus, to name only two), mathematics from different fields (e.g., number theory, topology, and numerical analysis), and mathematics which use different methods (such as proof by contradiction, cases, or induction). There are several immediate possibilities, and the group is always looking for new and interesting examples.

Just as important as the exercise of annotation, however, is the accumulation of a library of MathLang-encoded documents, and documents which have been formalised with the aid of MathLang. Firstly, giving the system a way to refer to other documents for prior work will greatly improve the structure of extensive annotation efforts, and reduce duplication of data and the potential for error due to copying code from one document to another. In addition, it will provide a repository for examples, to which an author can refer to see how others have chosen to annotate certain mathematical expressions.

In earlier stages of MathLang, and in other projects [50], "Grundlagen der Analysis[2]" by Edmund Landau [32] has been considered an important text, and it is worthwhile to update the computerisations of this work with an annotated version which makes use of the latest additions to MathLang. In addition, because of its extensive use of proof by induction, "An Introduction to the Theory of Numbers" by G.H. Hardy and E.M. Wright [14] may prove to be a fruitful tome.

Expand CGa and TSa to cope with more mathematics.

There are some known mathematical constructs for which a satisfactory annotation has not yet been found, and there are surely others which have not yet come to the attention of this development team. The fact is that mathematical language is very expressive and flexible, presenting all

[2] "Foundations of Analysis".

kinds of interesting and unusual cases to challenge the computer system. Many common ways of expressing mathematics may currently be captured in MathLang, but there are others which elude us. Current known troublesome expressions for the system include

- good handling of expressions with omitted terms, such as
  - $\overbrace{x + \ldots + x}^{n \text{ times}}$
  - $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$, and
  - $\frac{1}{1+\frac{1}{1+\cdots}}$;
- proper treatment of proof by induction; as well as
- satisfactory treatment of relations such as modular equivalence e.g., $-1 \equiv 2 \pmod 3$.

Naturally, these are only a few cases which have been brought up in conversation with colleagues. We desire both to discover appropriate ways to accommodate these kinds of mathematics, as well as find new mathematical examples which we have not yet attempted to encode with the tools of MathLang.

One hereunto-unexplored area of mathematics, for MathLang, is geometry. The facilities of CGa, etc. do not currently accommodate diagrams in any meaningful way. It would be interesting to explore how the ideas of annotating text might be extended to annotating diagrams. In a similar vein, proofs without words are fascinating expressions of mathematical reasoning, but similarly defy our current methods for computerisation.

10.3 Review

This concludes our presentation of this PhD. student's current contributions to the MathLang project. We have seen an overview of the scientific ecosystem in which MathLang is developed, as well as the pre-existing

parts of MathLang. We have described the new developments of MathLang (and reviewed them in this chapter), and touched on a range of possibilities for taking the project forward. MathLang has a great deal of potential, and should continue to grow in the coming years.

Appendix A

Ring theory example

The accompanying three figures show three different views of the same document. This example is a brief selection from [12], and provides a definition of an algebraic ring with some brief corollaries. It was chosen because it concisely exhibits most of our developments in a very accessible text. The observant reader will note that, although the examples throughout the main body of Section 9 were drawn from this example, the annotations chosen in the main text of [18] are not in every case consistent with the corresponding annotations in this supplement. In particular, such deviations usually pertain to the logical interpretation. (E.g., "0" is annotated $^{0}0$ in one example and $^{\mathrm{zero}}0$ in another.)

There are two essential observations to be made here. Firstly, while it is essential to maintain consistency in the naming of logically equivalent entities within a given document, the system is intended to be flexible. It is built in the object-oriented paradigm, as explained in [22], and is intended to give much of the same flexibility afforded to the users of modern programming languages.

The first of the views is completely without any grammatical or souring annotation. This is the form in which one would enter the text before giving any consideration to grammatical categorisation. The second view is after the document has undergone a full annotation. The third view differs from the second only inasmuch as a toggle has been switched to

Rings

Definition 1. *A ring R is a nonempty set with two binary operations, addition (denoted by $a+b$) and multiplication (denoted by ab), such that for all a, b, c in R:*

1. $a+b=b+a$.
2. $(a+b)+c=a+(b+c)$.
3. *There is an additive identity 0. That is, there is an element 0 in R such that $a+0=a$ for all a in R.*
4. *There is an element $-a$ in R such that $a+(-a)=0$.*
5. $a(bc)=(ab)c$.
6. $a(b+c)=ab+ac$ *and* $(b+c)a=ba+ca$.

Theorem 2.

1. $a0=0a=0$.
2. $a(-b)=(-a)b=-ab$.

Proof.

Consider rule 1.

Clearly,

$$0+a0=a0=a(0+0)=a0+a0. \tag{1}$$

So, by cancellation, $0=a0$. Similarly, $0a=0$.

To prove rule 2, we observe that $a(-b)+ab=a(-b+b)=a0=0$.

Adding $-(ab)$ to both sides yields $a(-b)=-(ab)$. The remainder of rule 2 is done analogously. □

Figure A.1 Ring theory text taken from *Contemporary Abstract Algebra* [12].

enable the display of the logical interpretations (introduced in Section 4.3) for each box.

A.1 Fully-Formalised Ring Theory

The following code is a completely formalised Isabelle document which corresponds roughly to the text in Figures A.1 and A.3.

```
theory Gallianfull
imports Main
begin
```

Rings

Definition 1. *A ring R is a nonempty set with two binary operations, addition (denoted by $a+b$) and multiplication (denoted by ab), such that for all a, b, c in R:*

1. $a+b=b+a$.
2. $(a+b)+c=a+(b+c)$.
3. *There is an additive identity 0. That is, there is an element 0 in R such that $a+0=a$ for all a in R.*
4. *There is an element $-a$ in R such that $a+(-a)=0$.*
5. $a(bc)=(ab)c$.
6. $a(b+c)=ab+ac$ *and* $(b+c)a=ba+ca$.

Theorem 2.

1. $a0=0a=0$.
2. $a(-b)=(-a)b=-ab$.

Proof.

Consider rule 1.

Clearly,

$$0+a0=a0=a(0+0)=a0+a0. \tag{1}$$

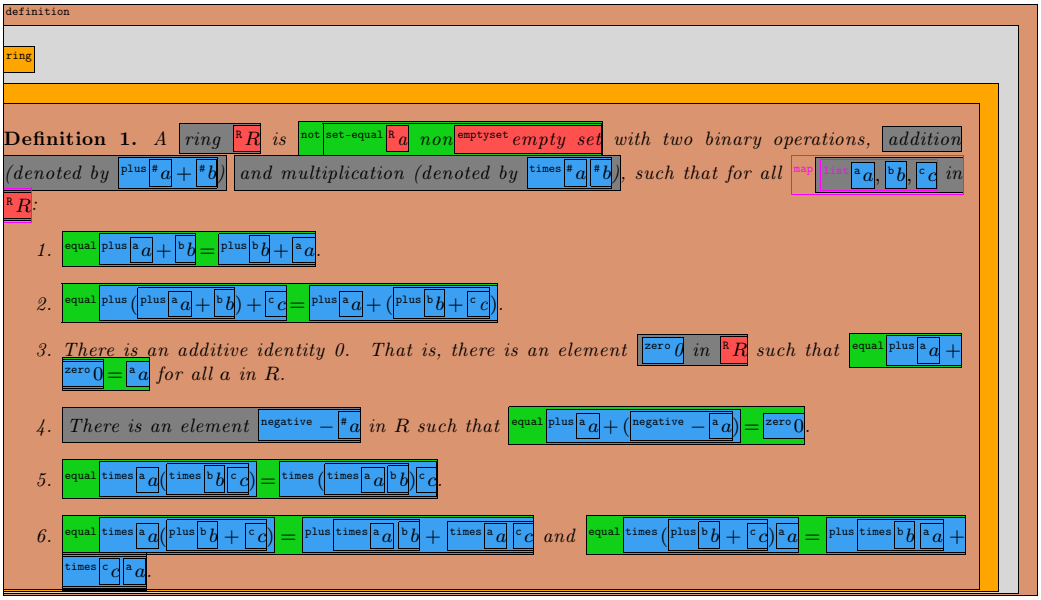
So, by cancellation, $0=a0$. Similarly, $0a=0$.

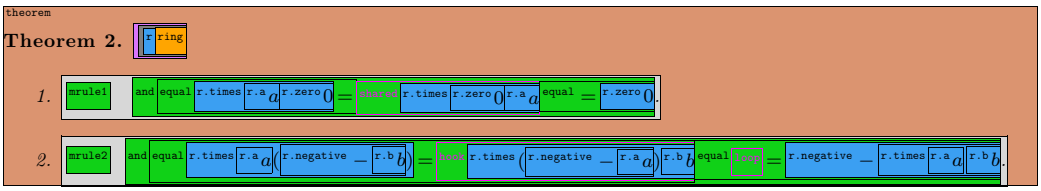
To prove rule 2, we observe that $a(-b)+ab=a(-b+b)=a0=0$.

Adding $-(ab)$ to both sides yields $a(-b)=-(ab)$. The remainder of rule 2 is done analogously. □

Figure A.2 Ring theory text with coloured annotations.

Rings





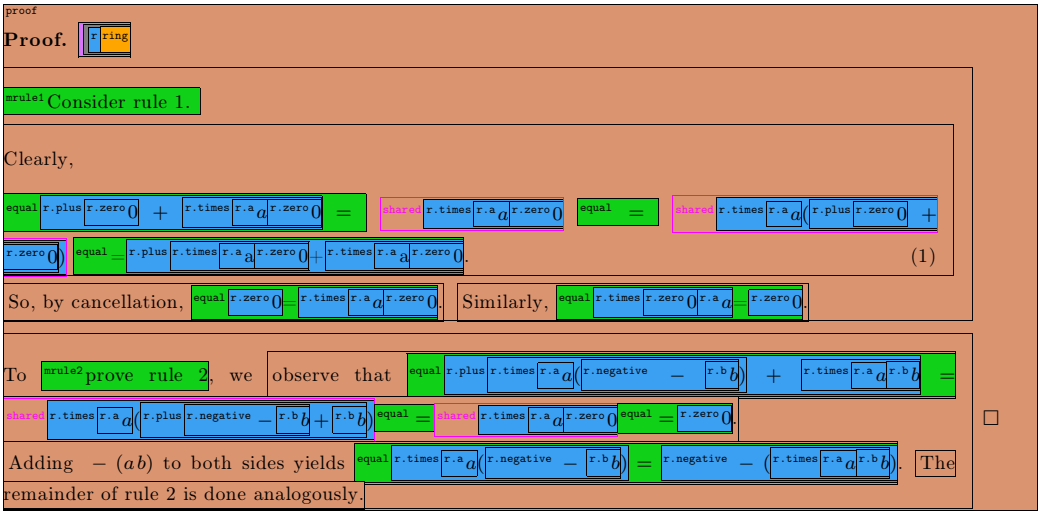


Figure A.3 Ring theory text with complete CGa annotation

```
4
locale ring =
6   fixes                R :: "'r set"
    assumes nonempty:   "R \<noteq> {}"
8   fixes                plus  :: "'r => 'r => 'r"
    fixes                times :: "'r => 'r => 'r"
10  assumes pluscomm:   "[| a:R; b:R |] ==> plus a b = plus b a"
        and plusassoc:  "plus (plus a b) c = plus a (plus b c)"
12  fixes                zero :: "'r"
    assumes zeroinR:    "zero : R"
14      and zeroisid_r: "a:R ==> plus a zero = a"
    fixes                negative :: "'r => 'r"
16  assumes negcancels: "a:R ==> plus a (negative a) = zero"
    assumes timescomm:  "[| a:R; b:R; c:R |] ==>
18                          times a (times b c) =
                              times (times a b) c"
20      and distleft:   "[| a:R; b:R; c:R |] ==>
                          times a (plus b c) =
22                          plus (times a b) (times a c)"
        and distright:  "[| a:R; b:R; c:R |] ==>
24                          times (plus b c) a =
                              plus (times b a) (times c a)"
26  assumes plusclosed: "[| a:R; b:R |] ==> plus a b : R"
       and timesclosed: "[| a:R; b:R |] ==> times a b : R"
28      and negclosed:  "a:R ==> negative a : R"

30 theorem (in gallianring)
   assumes "x : R" and "bar : R"
32 shows "plus x bar = plus bar x"
   proof -
34   from prems pluscomm [of x bar]
     show ?thesis by simp
36 qed

38 lemma sub_2arg_pred: "a=b ==> P a c = P b c" by blast

40 lemma (in gallianring) add_on_both_sides:
   assumes "a=b" shows "plus a c = plus b c"
42 proof -
     from prems sub_2arg_pred [of a b plus c]
44   show ?thesis by simp
   qed
46
   lemma (in gallianring) zeroisid_l:
48 assumes "a:R"
   shows "plus zero a = a"
50 proof -
     from zeroinR prems pluscomm [of zero a]
52   have "plus zero a = plus a zero" by simp
     also from prems this zeroisid_r [of a]
```

```
54        have "... = a" by auto
      finally show ?thesis .
56  qed

58  lemma (in gallianring) plus_cancels:
    assumes "a:R" and "b:R" and "c:R" and "plus a c = plus b c"
60  shows "a = b"
    proof -
62    from prems
          add_on_both_sides [of "plus a c" "plus b c" "negative c"]
64     have "plus (plus a c) (negative c) = plus (plus b c) (negative c)"
         by auto
66    from this plusassoc [of a c "negative c"]
          plusassoc [of b c "negative c"]
68     have "plus a (plus c (negative c)) = plus b (plus c (negative c))"
         by auto
70    from prems this negcancels [of c] zeroisid_r [of a]
          zeroisid_r [of b]
72      show "a = b" by auto
    qed
74
    theorem (in gallianring)
76  assumes "a:R" and "b:R"
    shows
78    "times a zero = times zero a \<and> times zero a = zero
       \<and>
80     times a (negative b) = times (negative a) b
         \<and> times (negative a) b = negative (times a b)"
82  proof -
      have B [simp]: "times zero a = zero"
84    proof -
        from prems zeroinR timesclosed [of zero a]
86          zeroisid_r [of "times zero a"]
          have "times zero a = plus (times zero a) zero" by auto
88      also from prems this zeroinR timesclosed [of zero a]
            negcancels [of "times zero a"]
90        have "... = plus (times zero a)
            (plus (times zero a) (negative (times zero a)))"
92        by auto
        also from this plusassoc [of "times zero a" "times zero a"
94          "negative (times zero a)"]
          have "... = plus (plus (times zero a) (times zero a))
96          (negative (times zero a))" by auto
        also from prems zeroinR distright [of a zero zero]
98        have "... = plus (times (plus zero zero) a)
                                            (negative (times zero a))"
100       by simp
        also from zeroinR zeroisid_l [of zero]
102       have "... = plus (times zero a) (negative (times zero a))"
          by simp
```

```
104     also from prems zeroinR timesclosed [of zero a]
            negcancels [of "times zero a"]
106       have "... = zero" by simp
        finally show ?thesis .
108   qed
      have A: "times a zero = times zero a"
110   proof -
        have "plus zero (times a zero)
112                               = plus (times a zero) (times a zero)"
        proof -
114       from prems zeroinR timesclosed [of a zero]
              zeroisid_l [of "times a zero"]
116         have "plus zero (times a zero) = times a zero" by auto
          also from zeroinR zeroisid_l [of zero]
118         have "... = times a (plus zero zero)" by auto
          also from prems zeroinR distleft [of a zero zero]
120         have "... = plus (times a zero) (times a zero)" by auto
          finally show ?thesis .
122     qed
        from prems zeroinR timesclosed [of a zero]
124         this plus_cancels [of zero "times a zero" "times a zero"]
          have "zero = times a zero" by auto
126     from this B show "times a zero = times zero a" by auto
      qed
128   have D: "times (negative a) b = negative (times a b)"
      proof -
130     have "plus (times (negative a) b) (times a b) = zero"
        proof -
132       from prems negclosed [of a] distright [of b "negative a" a]
          have "plus (times (negative a) b) (times a b)
134                                   = times (plus (negative a) a) b"
          by auto
136       also from prems negclosed [of a] pluscomm [of "negative a" a]
            negcancels [of a]
138         have "... = times zero b" by auto
          also have "... = zero"
140       proof -   -- "This is slightly modified from the proof for B"
            from prems zeroinR timesclosed [of zero b]
142             zeroisid_r [of "times zero b"]
            have "times zero b = plus (times zero b) zero" by auto
144         also from prems this zeroinR timesclosed [of zero b]
                negcancels [of "times zero b"]
146           have "... = plus (times zero b)
              (plus (times zero b) (negative (times zero b)))"
148           by auto
            also from this plusassoc [of "times zero b" "times zero b"
150               "negative (times zero b)"]
              have "... = plus (plus (times zero b) (times zero b))
152               (negative (times zero b))" by auto
            also from prems zeroinR distright [of b zero zero]
```

```
154           have "... = plus (times (plus zero zero) b)
                                          (negative (times zero b))"
156           by simp
            also from zeroinR zeroisid_l [of zero]
158           have "... = plus (times zero b) (negative (times zero b))"
              by simp
160         also from prems zeroinR timesclosed [of zero b]
                negcancels [of "times zero b"]
162           have "... = zero" by simp
            finally show ?thesis .
164       qed
          finally show ?thesis .
166     qed
        from this
168       have "plus (plus (times (negative a) b) (times a b))
                                                  (negative (times a b))
170                     = plus zero (negative (times a b))"
          by auto
172     from this plusassoc
          have "plus (times (negative a) b) (plus (times a b)
174                                           (negative (times a b)))
                        = plus zero (negative (times a b))"
176       by auto
        from prems this timesclosed [of a b]
178         negcancels [of "(times a b)"] negclosed [of a]
            zeroisid_r [of "times (negative a) b"]
180         timesclosed [of "negative a" b]
          have "times (negative a) b = plus zero (negative (times a b))"
182       by auto
        from prems this timesclosed [of a b] negclosed [of "times a b"]
184         zeroisid_l [of "negative (times a b)"]
          have "times (negative a) b = negative (times a b)" by auto
186     from prems this show ?thesis by auto
      qed
188   have C: "times a (negative b) = times (negative a) b"
      proof −
190     have "plus (times a (negative b)) (times a b) = zero"
        proof −
192       from prems negclosed [of b] distleft [of a "negative b" b]
            have "plus (times a (negative b)) (times a b)
194                                       = times a (plus (negative b) b)"
            by auto
196       also from prems this negclosed [of b] negcancels [of b]
            pluscomm [of "negative b" b]
198         have "... = times a zero" by auto
          also from this A B
200         have "... = zero" by auto
          finally show ?thesis by auto
202     qed
        from this
```

```
204       have "plus (plus (times a (negative b)) (times a b))
                                                  (negative (times a b))
206                           = plus zero (negative (times a b))"
          by auto
208     from prems this plusassoc timesclosed [of a b]
          negcancels [of "times a b"]
210       have "plus (times a (negative b)) zero
                                  = plus zero (negative (times a b))"
212       by auto
        from this zeroisid_l [of "negative (times a b)"]
214         zeroisid_r [of "times a (negative b)"] prems
          timesclosed [of a b] negclosed [of "times a b"]
216         negclosed [of b] timesclosed [of a "negative b"]
          have "times a (negative b) = negative (times a b)" by auto
218     from this D
          have "times a (negative b) = times (negative a) b" by auto
220     from this show ?thesis by auto
    qed
222   from A B C D show ?thesis by auto
  qed
224
  end
```

Appendix B

Number Theory Example

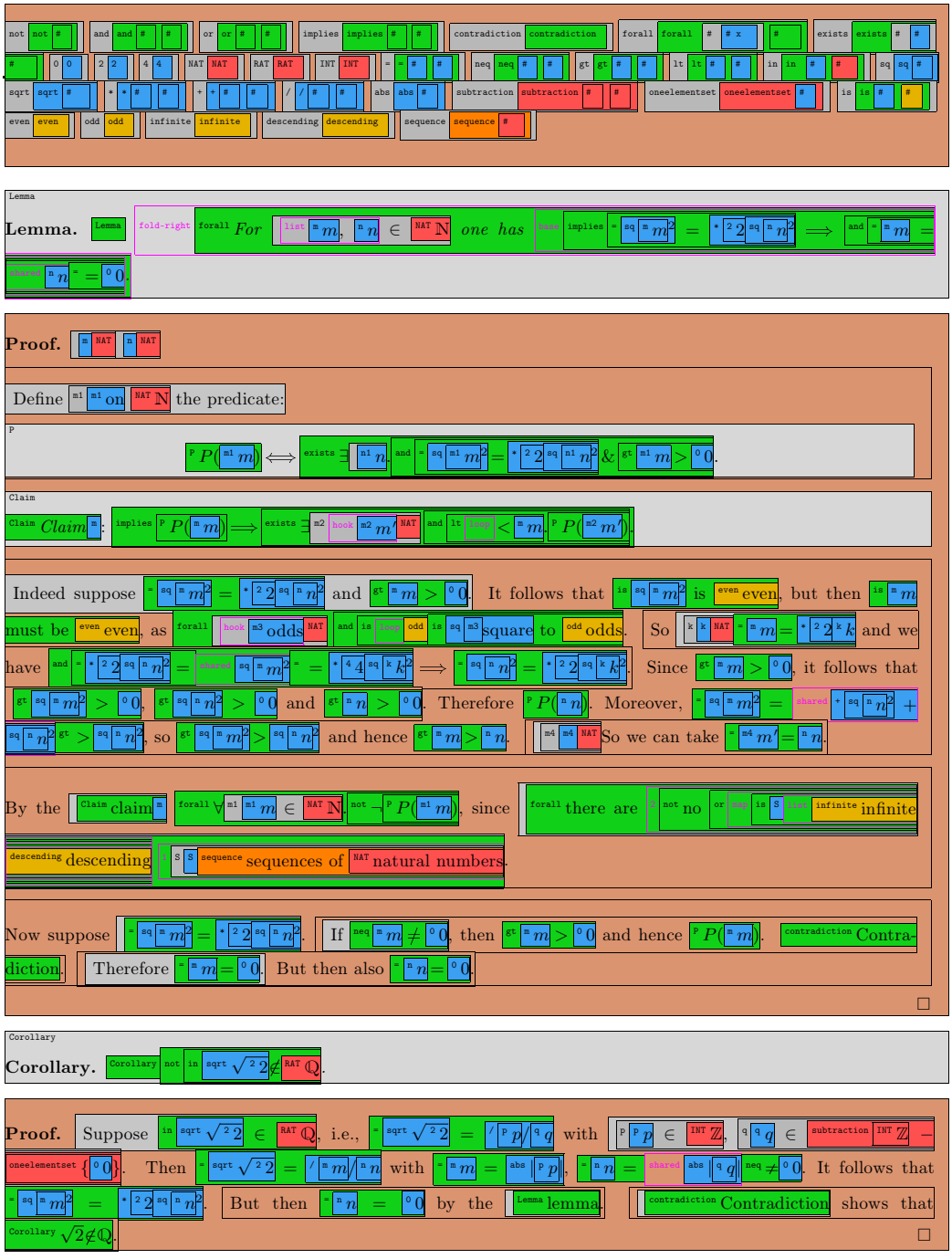


Figure B.1 Proof of $\sqrt{2} \notin \mathbb{Q}$ after CGa annotation

```
theory IrrTwo
imports Main
begin

theorem Lemma:
shows "ALL m:NAT, ALL n:NAT, m^2 = 2*n^2 ==> (m=n & n=0)"

have "is (m^2) even"
have "is m even"
have "ALL m3:NAT, (is m3 odd) & (is (m3^2) odd)"
have "m = 2*k"
have "(2*n^2 = m^2) & (m^2 = 4*k^2)"
have "n^2 = 2*k^2"
have "m > 0"
have "m^2 > 0"
have "n^2 > 0"
have "n > 0"
have "P n"
have "m^2 = n^2 + n^2"
have "n^2 + n^2 > n^2"
have "m^2 > n^2"
have "m > n"
have "m4 = n"
have "ALL m1:NAT, ~ (P n)"
have "m > 0"
have "P m"
have "contradiction"
have "n = 0"

theorem Corollary:
shows "~(sqrt 2 : RAT)"

have "sqrt 2 = p / q"
have "sqrt 2 = m / n"
have "m = abs p"
have "n = abs q"
have "abs q ~= 0"
have "m^2 = 2*n^2"
have "n = 0"
have "Corollary"

end
```

Figure B.2 Isabelle code created using rules from Section 8.1 on annotations in Figure B.1.

Bibliography

[1] A. Asperti, Luca Padovani, Claudio Sacerdoti Coen, and I. Schena. HELM and the semantic math-web. In *Theorem Proving in Higher Order Logics: 14th Int'l Conf., Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2001.

[2] 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.

[3] Serge Autexier, Christoph Benzmüller, Armin Fiedler, and Henri 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.

[4] 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 [38], pages 67–81.

[5] Clemens Ballarin. Locales and locale expressions in isabelle/isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.

[6] Gertrud Bauer. *The Hahn-Banach Theorem for Real Vector Spaces*. Technische Universit at Munchen, 2000.

[7] Chad E. Brown. Verifying and invalidating textbook proofs using Scunak. In MKM '06 [38], pages 110–123.

[8] B. Buchberger, A. Crăciun, T. Jebelean, L. Kovács, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Wind-

steiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, pages 470–504, 2006.

[9] S. Buswell, O. Caprotti, D. P. Carlisle, M. C. Dewar, M. Gaëtano, and Michael Kohlhase. The OpenMath standard, version 2.0. Technical report, The OpenMath Society, 2004.

[10] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*. W3C (World Wide Web Consortium), `http://www.w3.org/TR/xpath/`, 1999.

[11] N. G. de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In *Workshop on Programming Logic* [40]. Reprinted in [40, F.3].

[12] Joseph A. Gallian. *Contemporary Abstract Algebra*. Houghton Mifflin Company, 5th edition, 2002.

[13] Klaus Grue. The layers of logiweb. In MKM '07 [39], pages 250–264.

[14] Godfrey Harold Hardy and Edward Maitland Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 5th edition, April 1980.

[15] Thomas L. Heath. *The 13 Books of Euclid's Elements*. Dover, 1956. In 3 volumes. Sir Thomas Heath originally published this in 1908.

[16] Muhammad Humayoun. Software specifications and mathematical proofs in natural languages. Poster in Journées scientifiques du cluster, ISLE Rhône-Alpes, Domaine universitaire, Grenoble, France, 2008.

[17] Muhammad Humayoun and Christophe Raffalli. MathNat – mathematical text in a controlled natural language. *Journal on Research in Computing Science*, 46, 2010. Special issue: Natural Language Processing and its Applications.

[18] Fairouz Kamareddine, Robert Lamar, Manuel Maarek, and J. B. Wells. Restoring natural language as a computerised mathematics input method. In MKM '07 [39], pages 280–295.

[19] Fairouz Kamareddine, Manuel Maarek, Krzysztof Retel, and J. B. Wells. Gradual computerisation/formalisation of mathematical texts into Mizar. In Roman Matuszewski and Anna Zalewska, editors,

From Insight to Proof: Festschrift in Honour of Andrzej Trybulec, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 95–120. University of Białystok, 2007. Under the auspices of the Polish Association for Logic and Philosophy of Science.

[20] Fairouz Kamareddine, Manuel Maarek, Krzysztof Retel, and J. B. Wells. Narrative structure of mathematical texts. In MKM '07 [39], pages 296–311.

[21] Fairouz Kamareddine, Manuel Maarek, and J. B. Wells. Flexible encoding of mathematics on the computer. In MKM '04 [37], pages 160–174.

[22] Fairouz Kamareddine, Manuel Maarek, and J. B. Wells. Toward an object-oriented structure for mathematical text. In *Mathematical Knowledge Management, 4th Int'l Conf., Proceedings*, volume 3863 of *Lecture Notes in Artificial Intelligence*, pages 217–233. Springer, 2006.

[23] Fairouz Kamareddine and Rob Nederpelt. A refinement of de Bruijn's formal language of mathematics. *J. Logic Lang. Inform.*, 13(3):287–340, 2004.

[24] Fairouz Kamareddine and J. B. Wells. Computerizing mathematical text with MathLang. In Mauricio Ayala-Rincon and Heusler, editors, *Proc. Second Workshop on Logical and Semantic Frameworks, with Applications*, pages 5–30, Ouro Preto, Minas Gerais, Brazil, 2008. Elsevier. The LSFA '07 (post-event) proceedings is published as vol. 205 (2008-04-06) of *Elec. Notes in Theoret. Comp. Sci.*

[25] Manfred Kerber and Martin Pollet. A tough nut for mathematical knowledge management. In MKM '06 [38], pages 81–95.

[26] Donald Erwin Knuth and Silvio Levy. *The CWEB System of Structured Documentation: Version 3.0*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[27] Andrea Kohlhase, Michael Kohlhase, and Christoph Lange. SᴛEX – a system for flexible formalization of linked data. In Nicola Henze, Adrian Paschke, Andreas Blumauer, Richard Cyganiak, and Tassilo Pellegrini, editors, *Proceedings of I-Semantics 2010*. ACM, 2010.

[28] Michael Kohlhase. *An Open Markup Format for Mathematical Documents, OMDoc (Version 1.2)*, volume 4180 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2006.

[29] Michael Kohlhase. Using LaTeX as a semantic markup format. *Mathematics in Computer Science*, 2(2):279–304, December 2008.

[30] Robert Lamar. Rings in automated proof. Senior research paper, Stetson University, April 2006.

[31] Robert Lamar, Fairouz Kamareddine, and J. B. Wells. MathLang translation to Isabelle syntax. In *Intelligent Computer Mathematics (Calculemus 2009 and MKM 2009 joint proceedings)*, volume 5625 of *Lecture Notes in Artificial Intelligence*, pages 373–388. Springer, 2009.

[32] Edmund Landau. *Grundlagen der Analysis*. Chelsea, 1930.

[33] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) model and syntax specification. W3C Recommendation, `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[34] LogiCal Project, INRIA, Rocquencourt, France. *The Coq Proof Assistant Reference Manual – Version 8.0*, June 2004. Available at `ftp://ftp.inria.fr/INRIA/coq/V8.0/doc/`.

[35] Manuel Maarek. *Mathematical Documents Faithfully Computerised: the Grammatical and Text & Symbol Aspects of the MathLang Framework*. PhD thesis, Heriot-Watt University, Edinburgh, Scotland, June 2007.

[36] Lionel Elie Mamane and Herman Geuvers. A document-oriented Coq plugin for TeXmacs. In *Mathematical User-Interfaces Workshop 2006 [Workshop]*, Wokingham, UK, 2006.

[37] *Mathematical Knowledge Management, 3rd Int'l Conf., Proceedings*, volume 3119 of *Lecture Notes in Computer Science*. Springer, 2004.

[38] *Mathematical Knowledge Management, 5th Int'l Conf., Proceedings*, volume 4108 of *Lecture Notes in Computer Science*. Springer, 2006.

[39] *Towards Mechanized Mathematical Assistants (Calculemus 2007 and MKM 2007 Joint Proceedings)*, volume 4573 of *Lecture Notes in Artificial Intelligence*. Springer, 2007.

[40] 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.

[41] Tobias Nipkow. Structured proofs in Isar/HOL. In *Types for Proofs and Programs*, Lecture Notes in Computer Science 2277. Springer, December 2002.

[42] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[43] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[44] Luca Padovani and Stefano Zacchiroli. From notation to semantics: There and back again. In MKM '06 [38], pages 194–207.

[45] Martin Pollet, Volker Sorge, and Manfred Kerber. Intuitive and formal representations: The case of matrices. In MKM '04 [37].

[46] Aarne Ranta. Grammatical framework: A type-theoretical grammar formalism. *J. Funct. Programming*, 14(2):145–189, 2004.

[47] Krzysztof Retel. *Gradual Computerisation and Verification of Mathematics: MathLang's Path into Mizar*. PhD thesis, Heriot-Watt University, Edinburgh, Scotland, April 2009.

[48] P. Rudnicki. An overview of the Mizar project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[49] Andrzej Trybulec. The mizar logic information language. *Studies in Logic*, 1, 1980. Bialystok.

[50] Lambert S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system*. PhD thesis, Eindhoven, 1977.

[51] W3C. Mathematical markup language (mathml) version 2.0. W3C Recommendation, February 2001.

[52] Marc Wagner. *A Change-Oriented Architecture for Mathematical Authoring Assistance*. PhD thesis, Universität des Saarlandes, 2011.

[53] Markus 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.

[54] Markus Wenzel and Freek Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. *J. Automated Reasoning*, 29(3–4):389–411, 2002.

[55] F. Wiedijk, editor. *The Seventeen Provers of the World, foreword by Dana S. Scott*, volume 3600 of *LNCS*. Springer Berlin, Heidelberg, 2006.

[56] Freek Wiedijk. Mizar: An impression. `http://www.cs.ru.nl/~freek/notes/`.

[57] Freek Wiedijk. Formal proof sketches. In *Proceedings of TYPES'03*, volume 3085 of *LNCS*, pages 378–393. Springer-Verlag, December 2004.

Index of Subjects

Authors of Referenced Work