# FROM FORMAL SPECIFICATION TO FULL PROOF:
# A STEPWISE METHOD

*by*

Lavinia Burski

Submitted for the degree of

Doctor of Philosophy

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF MATHEMATICAL AND COMPUTER SCIENCES

HERIOT-WATT UNIVERSITY

March 2016

# Todo list

# Acronyms

**CGa** Core Grammatical aspect.

**DRa** Document Rhetorical aspect.

**MathLang** MathLang framework for mathematics.

**TSa** Text and Symbol aspect.

**UTP** Unifying theories of programming.

**WTT** weak type theory.

**ZCGa** Z Core Grammatical aspect.

**ZDRa** Z Document Rhetorical aspect.

**ZMathLang** a toolkit for checking various degrees of correctness for Z specifications.

# Glossary

**computerisation** The process of putting a document in a computer format.

**formalisation** The process of extracting the essence of the knowledge contained in a document and providing it in a complete, correct and unambiguous format.

# Chapter 1

# Background

Formal methods are a specific type of mathematical notation which is based on the techniques of the specification, verification and development of software and hardware systems [11]. Since our thesis presents a toolkit for checking various degrees of correctness for Z specifications (ZMathLang) we go right to the beginning of the framework, to describe how mathematical notation came about. Then we describe the original MathLang framework (the framework which ZMathLang is an adaptation of) and then give the reader an idea of other formal methods and languages. In the next section we wish to describe what is the language of Z and give more details of it's syntax and semantics. We then highlight other proving techniques which have been done for maths, formal methods and Z.

## 1.1 Mathematical Notations

Computer science (and thus computer systems) have evolved from basic mathematics. We can say that formal specification writers are practising mathematicians as they write system specifications in a formal manner. Therefore we must start right at the beginning at the foundation of mathematical notation.

### 1.1.1 Right from the beginning

The relationship between mathematical reasoning and practising mathematicians started out early on during the ancient Greeks where logic was already being studied.

Reasoning in logic was used for just about anything not just mathematics such as law, medicine and farming. This very early form of mathematics made very famous discoveries such as Aristotles logic [58], Euclid's geometry [18] and Leibniz Calculus [38].

Further on in the 1800's, Frege wrote *Die Grundlagen der Arithmetik* [22] and other works where he noted that mathematics is a branch of logic. In this works, he began building a solid foundation for mathematics. This early foundation along with Cantors set theory [13] was argued to be inconstant and thus Russel found a paradox in this work.

In the late 19th century and beginning of 20th century, Russell & Whitehead [54] started to form a basis for mathematical notation. Their three volume work describes a set of rules from which all mathematical truths could be proven. In these early stages the authors try to derive all maths from logic. This ambitious project was the first stepping stone in collaborating all mathematics under one notation.

Further to Russell & Whitehead's work, Bourbanki [1] wrote a series of books beginning n the 1935's with the aim of grounding mathematics. Their main works is included in the Elements of Mathematics series [8] which does not need any special of knowledge of mathematics. It describes mathematics from the very beginning and goes through core mathematical concepts such as set theory, algebra, function etc. and gives complete proofs for these concepts.

Adding to Russell's work, Zermelo introduced an acclimatisation of set theory which was later extended by Frankel and Skolem to form ZF set theory [52]. This new theory is what we will later see the Z notation is based on and the notation this thesis checks the correctness of.

## 1.1.2   Computerisation of Maths and Proof Systems

In the 21st century, a great area of research is how use, store and support this mathematical knowledge. Since automation has become more and more used, mathematicians have looked into ways in which they can use computers to reason about

---

[1]A name given to a collective of mathematicians

and provide services to mathematics. This would include all areas of mathematics, such as logic, mechanics and software specifications. Mathematical knowedge can be represent in lots of different ways including the following:

- One can typeset mathematics into a computer using a system such as LaTeX [36] . These systems can edit and format mathematical knowledge so that it can be stored or printed. These systems provide good visual appearance and thus can be used for storing and archiving ones documents. Even Z specification, have their own package for LaTeX and thus the structure of the specification can be represented both formally and informally. However, it is difficult to represent the logical structure of mathematical formulas and the logical structure of mathematics is embedded in natural language. Therefore, there isn't a lot of support for checking the correctness of general mathematics represented in the system (Z specifications on their own can be checked but this is discussed further in section 1.5.4).

- Systems such as proof assistants (e.g. Isabelle [50], Coq [48] and ProofPower-Z [5]) and automated theorem provers (e.g. Boyer-Moore, Otter) and jointly called proof systems. Proof systems each provide a formal language for writing mathematics. Early work on proof systems was done by De Bruijn when he worked on the AutoMath [39] project. AutoMath (automating mathematics) was the first attempt to digitize and formally prove mathematics which was assisted by a computer. AutoMath is described as a language for formalising mathematical texts and for automating the validation of mathematics. The AutoMath project is what brought uniform notations and automated proof together.

  Further to this work, there has been many other proof systems implemented to implement and check mathematics for total correctness. It is possible to access the semantics of mathematics in these systems. However, with these proof systems, a user must choose a specific proof system and one of these have their advantages and disadvantages. Also, each of these proof systems also take quite some time to learn. There is much documentation on some of

these systems (e.g. Isabelle) and some are very well supported. But this in turn can be a downfall, as there is so much documentation, it is difficult to know how much one must learn and where to start. The best way of learning one of these system is from someone who already is an expert in the chosen proof system. A lot of the proof systems use proof tactics to constructs proofs and make them smaller, however to prove certain properties in a proof system, one can use various tactics to get to the same goal and may sometimes be difficult to find which tactic is best to use.

With these disadvantages many academic and industrial mathematicians do not generally use the mathematics written in the language of the proof system and usually are not willing to spend the time to check the correctness of their own work in this system.

- There also exists semantical oriented document representations like OpenMath [1] and OMDoc [30]. These systems are better than the typesetting systems such as LaTeX to produce readable and printable version of the mathematical knowledge written. Some aspects of the semantics of the mathematics can be represented in these type systems. However, when using these systems it is difficult to control the presentation and therefore a typesetting system is more likely to be used. Like the typesetting system, systems like OMDoc also have difficult reading the logical structure of mathematics embedded in the natural language of the text. Systems like OMDoc can associate symbolic formulas with chunks of natural language text, however these chunks can not be seen by the computer and thus can not be checked if it is correct.

  Another disadvantage is that although there is support for the semantics of the mathematics, these systems can not support the semantics in terms of logical foundations for mathematics (unlike proof systems).

### 1.1.3 Conclusion

In summary the MathLang framework for mathematics (MathLang) framework has been developed to be used as a bridge between the categories mentioned above as

a way to represent and automatically check mathematical knowledge. Since the Z notation has stemmed from the origins of mathematics and industry is starting to use formal methods in there system development we have chosen to adapt the MathLang framework to accommodate Z specification and have developed a set of tools to do so.

## 1.2    MathLang for mathematics

MathLang originally started in 2000. It's original goals was to allow gradual computerisation and formalisation of mathematical texts.

MathLang is not a system for proof verification but a framework to computerise and translate information (such as mathematical text) into a form on which proof checkers can operate.

The MathLang framework provides extra features supporting more rigour to translation of the common mathematical language. One can define further levels of translations into more semantically and logically complete versions. This gradual computerisation method should be more accessible than direct formalisation, because a number of first levels do nor require any particular expertise in formalisation.

So far Mathlang has given alternative and complete paths which transform mathematical texts into new computerised and formalised versions. Dividing the formalisation of mathematical texts into a number of different stages was first proposed by N.G. de Bruijn to relate common mathematical language to his Mathematical Vernacular [17] and his proof checking system Automath.

### 1.2.1    Overview and Goals

The MathLang Framework instructs the computerisation process to be broken up into a number of levels called **aspects**. Each aspect can be worked out independently, simultaneously or sequentially without prior knowledge of another aspect. The current MathLang Framework contains three well-developed aspects, the Core

Grammatical aspect (CGa), the Text and Symbol aspect (TSa) and the Document Rhetorical aspect (DRa), and has further aspects such as the Formal Proof Sketch.
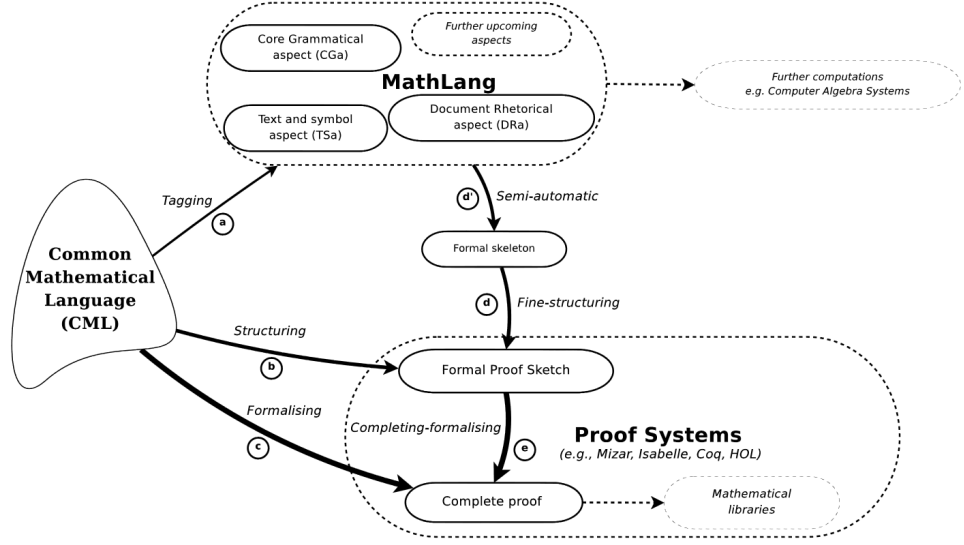


Figure 1.1: The MathLang approach to computerisation/formalisation [24]

Figure 1.1 shows the overall situation of work in the current MathLang Framework. The labelled arrows show the computerisation paths from the common mathematical language to any proof system. The width of the arrow representing each path segment increases according to the expertise required. The level of expertise needed to computerise a CML text straight into a complete proof is very high, however the level of expertise is much smaller by using the MathLang framework to help form a formal skeleton and then into a complete proof. The dashed arrows illustrate further computerisation that one can envision.

The initial design of MathLang was based on ideas from weak type theory (WTT) [26] which then inspired de Bruijns Mathematical Vernacular [17]. WTT defines terminology to describe a *book* which is a sequence of *lines*, each of which is a pair of a *sentence* (a *statement* or a *definition*) and a *context* of facts (*declarations* or *statements*) assumed in the *sentence*. A *definition* can introduce a name whose scope is associated with a meaning throughout the rest of the book. A *declaration* in the *context* can introduce a name whose scope is the current line. The *preface* gives names which can be reference throughout the entire document. Within *declarations*, *definitions* and *statements* are *phrases* which can be built from the basic elements;

*terms*, *sets*, *adjectives* and *nouns*. WTT uses a weak type system using the following types `noun`, `set`, `adjective`, `statement`, `definition`, `context` and `book` to check basic well-formedness conditions. The idea's of WTT are useful in some instances however it does not deal with proofs or logical correctness of the documents. There is no way to group statements together and indentify their roles within the document such as what is a theorem, what is a section, what is a chapter etc. It also doesn't provide a way to use some concepts used in a document within another.

We use the following standard metasymbols for the categories mentioned above:

| level | category | symbol | representative |
|---|---|---|---|
| atomic level | *variables* | V | $x$ |
| | *constants* | C | $c$ |
| | *binders* | B | $b$ |
| phrase level | *terms* | t | $t$ |
| | *sets* | $\sigma$ | $s$ |
| | *nouns* | $\mathcal{V}$ | $n$ |
| | *adjectives* | $\alpha$ | $a$ |
| sentence level | *statements* | $\mathcal{S}$ | $S$ |
| | *definitions* | $\mathcal{D}$ | $D$ |
| discourse level | *contexts* | $\boldsymbol{\Gamma}$ | $\Gamma$ |
| | *lines* | l | $l$ |
| | *books* | $\mathbf{B}$ | $B$ |

| category | symbol | representative |
|---|---|---|
| *expressions* | $\mathcal{E}$ | E |
| *parameters* | $\mathcal{P}$ | P |
| *typings* | $\mathcal{T}$ | T |
| *declarations* | $\mathcal{Z}$ | Z |

*Variables* can range of terms, sets or statements.

Parameters $\mathcal{P}$ are either terms, sets or statements. A *constant* is always followed by a parameter list and constants can be either for terms, nouns, statements, sets

or adjectives. For example:

The constant '+' takes 2 terms as parameters and gives back a term, therefore we can have:

$$+(3, 2)$$

meaning the constant + takes the parameters 3 and 2 and in this case would give back 5.

*Binders* can be giving terms, adjectives, nouns, sets or statements. For example a binder giving a set would be the symbol '$\bigcup$' or a binder giving a statement would be the symbol '$\forall$'.

A *definition* introduces new constants. We can have phrase definitions and statement definitions.

A *context* is a list of declarations and statements. A declaration in a context introduces variables which are then used in expressions. A context can be empty or non empty.

*Lines* contain either a statement or a definition which is relative to a context. A *book* is a list of lines. A simple book could be empty and not contain any lines or it can contain 1 or more lines.

A detailed explanation on each of the categories along with examples can be found in [26].

## 1.2.2   CGa

The current CGa in MathLang uses a finite set of grammatical *categories* to identify the structure and common concepts used in mathematical texts. The aims of the CGa is to make explicit the grammatical role played by the elements of mathematical texts and to allow the validation of the grammatical and reasoning structure within the CGa encoding in a mathematical text. The CGa checks for grammatical correctness and finds errors like an identifier being used without and prior introduction or the wrong number of arguments being given to a function [43].

The CGa for mathematical documents is a formal language originally created by

ideas by WTT and Mathematical Vernicular [17] and then adapted to suit mathematical texts by various PhD, Master and Dissertation students.

The smallest constructs of CGa are *step* and *expression*. Tasks handled by books, prefaces, lines, declarations, defintions, and statements are represented as steps in CGa. A step can be a *block* which is a sequence of steps. A step can also be a definition, declaration or expression.

For example:

'Given that $\mathfrak{M}$ is a set, $y$ and $x$ are natural numbers, and $x$ belongs to $\mathfrak{M}$, it holds

that $x + y = y + x$'.

This statement encoded in CGa would be the following:

$$\{M : set;\ y : naturalnumber;\ x : naturalnumber;\ \in (x, M)\}$$
$$\rhd = (+(x, y), +(y, x))$$

CGa uses grammatical *categories* (also called *types*) to make explicit the grammatical role played by the elements of a mathematical text in a document. All these described categories and the rules for the grammar typing can be found in [26]. CGa also uses colour coding for these categories. They are:

$\boxed{\textit{term}}$ , $\boxed{\textit{set}}$ , $\boxed{\textit{noun}}$ , $\boxed{\textit{adjective}}$ , $\boxed{\textit{expression}}$ , $\boxed{\textit{definition}}$ , $\boxed{\textit{declaration}}$ ,

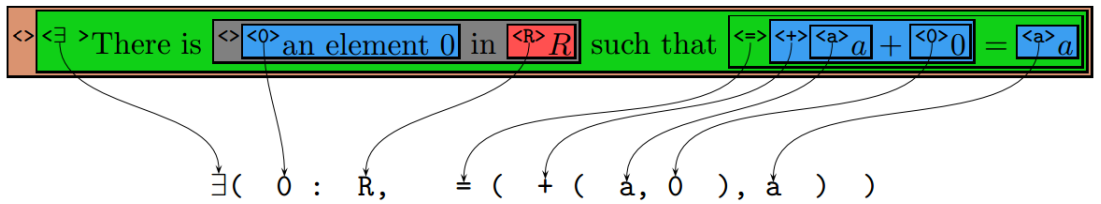$\boxed{\textit{step}}$ and $\boxed{\textit{context}}$



Figure 1.2: Example of CGa encoding of CML text, taken from [27].

CGa types are powerful does not allow higher order types, to work around this weakness CGa annotation often gives entitied the type term instead of a more precide type. The goal of the CGa type is system is not to endure full correctness but to check whether the reasoning parts of the document are coherent and built sensibly.

The design of the CGa is credited to Kamareddine, Maarek and Wells [25]. The implementation of CGa is due to Maarek [37].

### 1.2.3 DRa

The Document Rhetorical aspects checks that the correctness of the reasoning in the mathematical document is correct and that there are no loops. The DRa mark-up system is simple and more concentrated on the narrative structure of the mathematical documents whereas other previous systems (such as DocBook [2], Text Encoding Initiative [3], OMDoc [4]) were more concentrated on the subtleties of the documents. It is used to describe and annotate chunks of texts according to their narrative role played within the document [43]. Using the DRa annotation system we can capture the role that a chunk of text imposes on the rest of the document or on another chunk of text. This leads to generating dependency graphs which play an important role on mathematical knowledge representation. With these graphs, the reader can easily find their own way while reading the original text without the need to understand all of its subtleties. Processing DRa annotations can flag problems such as circular reasoning and poorly-supported theorems.

The annotation process for the DRa can be done before, after or with the annotations for CGa and TSa. Table 1.1 shows the roles and relations which can be used in annotating a document with DRa.

| Roles | Relations |
|---|---|
| lemma | justifies |
| proof | uses |
| corollary | subpartOf |
| definition | |
| claim | |
| case | |

Table 1.1: A table showing DRa roles and relations.

---

[2]http://www.docbook.org
[3]http://www.tei-c.org/index.xml
[4]http://www.omdoc.org

Annotations in DRa follow three steps:

1. The user wraps chunks of text with boxed. Each box is assigned a unique name, this avoids any confusion when stating relations between the boxes.

2. To each unique box, the user then assigns a structural or mathematical rhetorical role which this box may play. He uses RDF triplets to assign a role to each box.

3. The user then describes the relations between each of the chunks of text he annotated before.

RDF triples [12] are used to represents the relationships between the annotated boxes. The triples are represented as *subject-predicate-object* triple, where *predicate* denotes the relationship. The order of *subject* and *object* is important as it denotes how the dependency graph is formed. An example of RDF triples for the Z Document Rhetorical aspect (ZDRa) is:

| Assigned Rhetorical Roles | Relations |
|---|---|
| (A, hasMathematicalRole, lemma) | (B, justifies, A) |

The DRa performs three basic checks:

1. Check if each of the mathematical and rhetorical roles and relations which are used have been defined.

2. Check that each name of a node is unique.

3. Check that each relation source and target are valid DRa names.
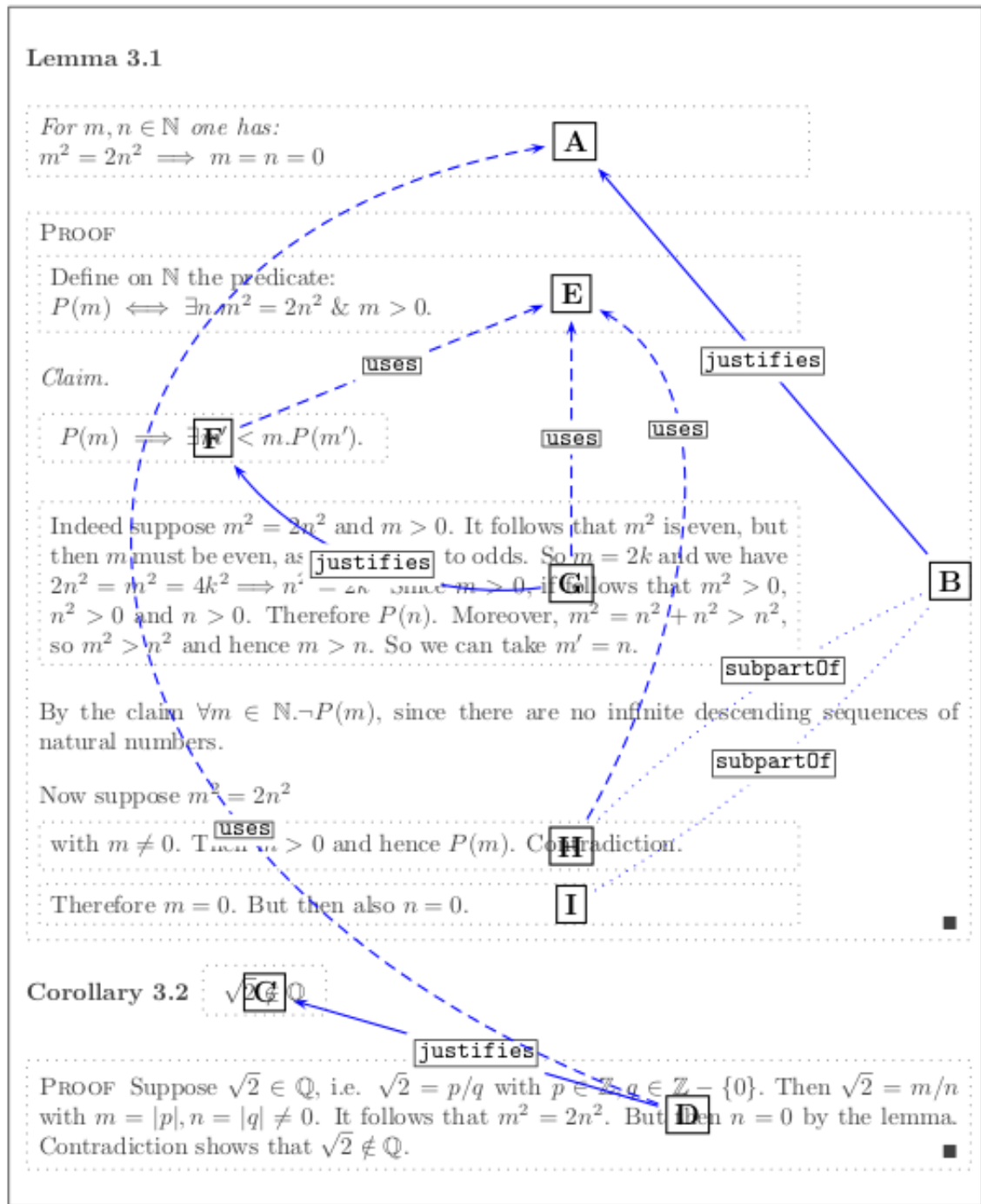
The DRa also automatically produces a dependency graph.

**Lemma 3.1**

*For $m, n \in \mathbb{N}$ one has:*
$m^2 = 2n^2 \implies m = n = 0$

PROOF

Define on $\mathbb{N}$ the predicate:
$P(m) \iff \exists n. m^2 = 2n^2 \ \& \ m > 0.$

*Claim.*

$P(m) \implies \exists m' < m. P(m').$

Indeed suppose $m^2 = 2n^2$ and $m > 0$. It follows that $m^2$ is even, but then $m$ must be even, as [justifies] to odds. So $m = 2k$ and we have $2n^2 = m^2 = 4k^2 \implies n^2 = 2k^2$. Since $m > 0$, it follows that $m^2 > 0$, $n^2 > 0$ and $n > 0$. Therefore $P(n)$. Moreover, $m^2 = n^2 + n^2 > n^2$, so $m^2 > n^2$ and hence $m > n$. So we can take $m' = n$.

By the claim $\forall m \in \mathbb{N}. \neg P(m)$, since there are no infinite descending sequences of natural numbers.

Now suppose $m^2 = 2n^2$

with $m \neq 0$. Then $m > 0$ and hence $P(m)$. Contradiction.

Therefore $m = 0$. But then also $n = 0$.

**Corollary 3.2** $\sqrt{2} \notin \mathbb{Q}$

PROOF Suppose $\sqrt{2} \in \mathbb{Q}$, i.e. $\sqrt{2} = p/q$ with $p \in \mathbb{Z}, q \in \mathbb{Z} - \{0\}$. Then $\sqrt{2} = m/n$ with $m = |p|, n = |q| \neq 0$. It follows that $m^2 = 2n^2$. But then $n = 0$ by the lemma. Contradiction shows that $\sqrt{2} \notin \mathbb{Q}$.

Figure 1.3: Wrapping/naming chunks of text and marking relationships in DRa [59].

#### 1.2.3.1 Dependency Graph

The dependency graph is a directed label graph which contains all annotated DRa nodes of a mathematical text and all the relations. The names of the instances become the nodes of the graph and the relations become the named directed vertices

of the graph [59].

Since each DRa annotation itself is a step, DRa annotations can be nested.

An example of a dependency graph is shown in figure 1.4:
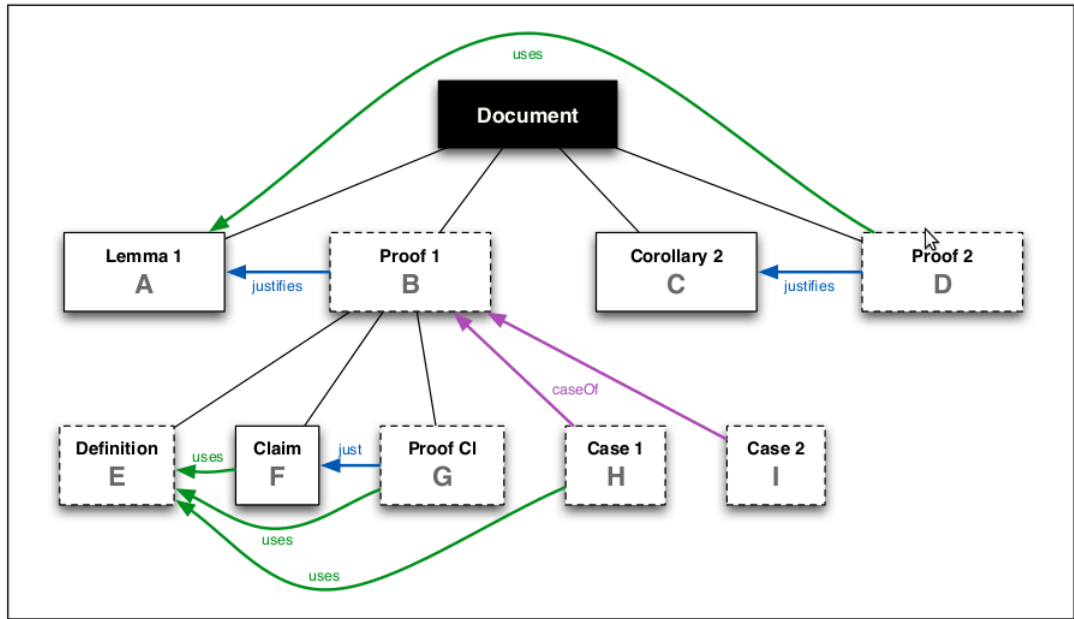


Figure 1.4: The Dependency Graph for the Pythagoras example, taken from [59].

#### 1.2.3.2    GoTo Graph

The textual order between two nodes within a tree can be used to examine the structure of the DRa. The textual order is a modification of the logical precedence presented in the dependency graph. The textual order expresses the dependencies between parts of the text and is necessary for the generation of a proof skeleton. For example if a node A uses node B than logically B should be written before A. The GoTo graph is the same as the dependency graph but rearranged to follow the correct textual order of the nodes and vertices. The GoTo graph is produced for the following reasons:

1. The automatic checking of the GoTo can reveal errors in the document such as loops in structure of the document.

2. The GoTo is used to produce a proof skeleton for a certain prover.
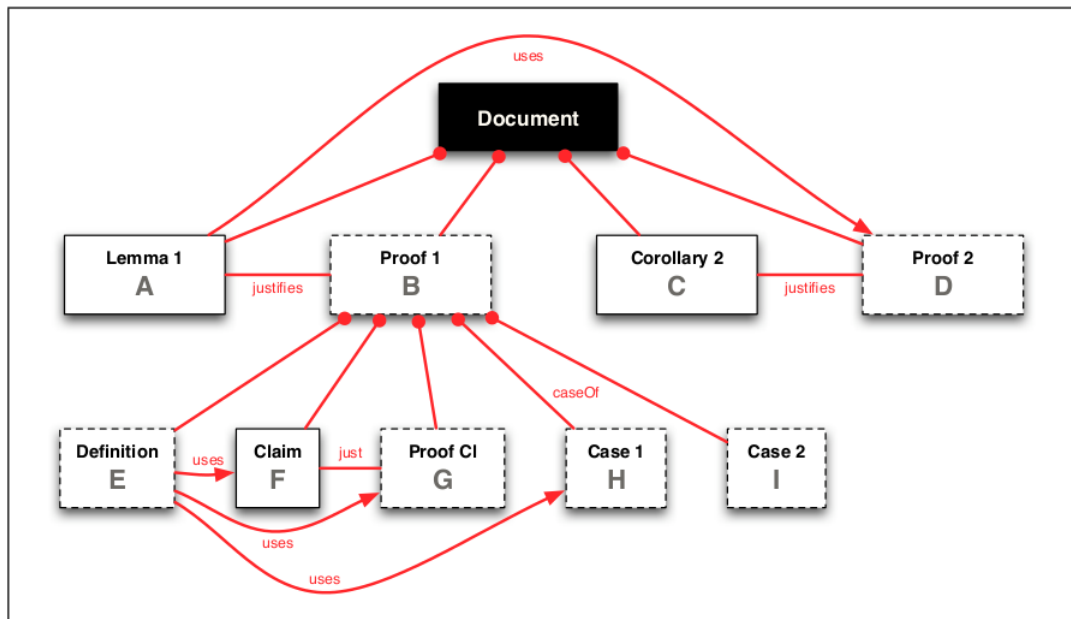
Figure 1.5: Example of the GoTO of a document, taken from [59].

### 1.2.4 Mathamatical skeletons

There has been some work on computerising mathematical texts without fully formalising or proof checking them. The computerisations are no detailed to check for full total correctness but are used as *skeletons* in the full formalisation [55]. However, in order to produce these skeletons the user still needs to be an expert in the theorem proving language they wish to check their mathematical document with. Once a document has been annotated with MathLang, the proof skeleton for a theorem prover could be generated from the annotations. 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 however they should only be justification errors.

Retel [43] describes a Mizar document skeleton which has been generated from a MathLang document. The skeleton uses the annotations from DRa and inputs the document in Mizar syntax along the names used in DRa. The original context of the mathematical document has not yet been input into the Mizar formal proof skeleton.

Figure 1.6: The Mizar formam proof document skeleton transformed from the Math-Lang computerised document [43].

The next part is to fill in the skeleton using all information and context of the original mathematical document, this may include the names of the theorems, statements, definitions etc. which were in the original mathematical text.



Figure 1.7: The Mizar FPS environment built for the example from Figure 1.6. The Mizar system responds with only one error stating that the theorem is not sufficiently proved, taken from [43].

Figure 1.7 still shows that there is still some work to do as the '*inference is not accepted*'. The user would then need to finish off the proof manually (shown in figure 1.8), however a big chunk of the work has already been done within the theorem prover.



Figure 1.8: A filled in Mizar proof built from Figure 1.7 taken from [43].

### 1.2.5 TSa

The TSa builds the bridge between a mathematical text and its grammatical interpretation. The TSa is a way of rewriting parts of the text so they have the same meaning. For example some mathematicians may prefer to write "a=b and b=c and c=d", others may prefer "a=b, b=c, c=d" and some others may prefer "a=b=c=d".

As you can see all these methods of writing have the same meaning however some symbols are different. The TSa annotates each expression in the text with a string of words or symbols which aim to act as the mathematical representation of which this expression is. This allows everything in the text to be uniform.

We do not describe the TSa in detail since we do not use the TSa in this thesis but it is important to know that it exists.

## 1.2.6 A fully worked example in Mathlang

This section describes a fully worked example of translating a mathamtical text into the theorem prover Isabelle using the MathLang framework, taken from [31].

### 1.2.6.1 The original text

This text was taken from an undergraduate textbook [21], chapter 12, entitled 'Introduction to Rings'.

**Definition 1.2.1.** *(Ring). A ring R is a non-empty set with two binary operations, addition (denoted by a + b) and multiplication(denoted by ab), suchthat 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. content...*

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

**Theorem 1.2.1.** *(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. □

### 1.2.6.2    CGa annotated text
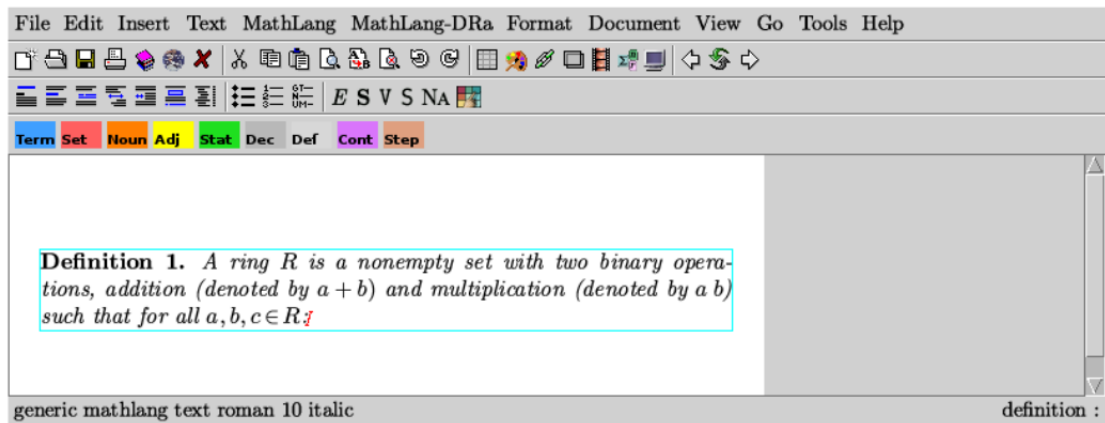


Figure 1.9: Typing the beginning of a document into TeXMacs.

The user then annotate this text using '*TeXMacs* shown in figure 1.9. These annotates would be made at sentence level and it is common for single words and individual variables to be annotated.

Figure 1.10: Completed CGa annotation of ring theory text.

### 1.2.6.3 DRa annotated text

The text is annotated with DRa relationships and instances which can find errors such as loops within the reasoning and poorly-supported theorems. Figure x shows the original ring theory text (left) and the ring theory text annotated with DRa (right).

**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. $a(bc)=(ab)c$.

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 1.11: Comparison of original common mathematical language text with a dependency graph.

The dependency graphs and goto graphs are automatically generated from the DRa annotated document shown in figure 1.11.
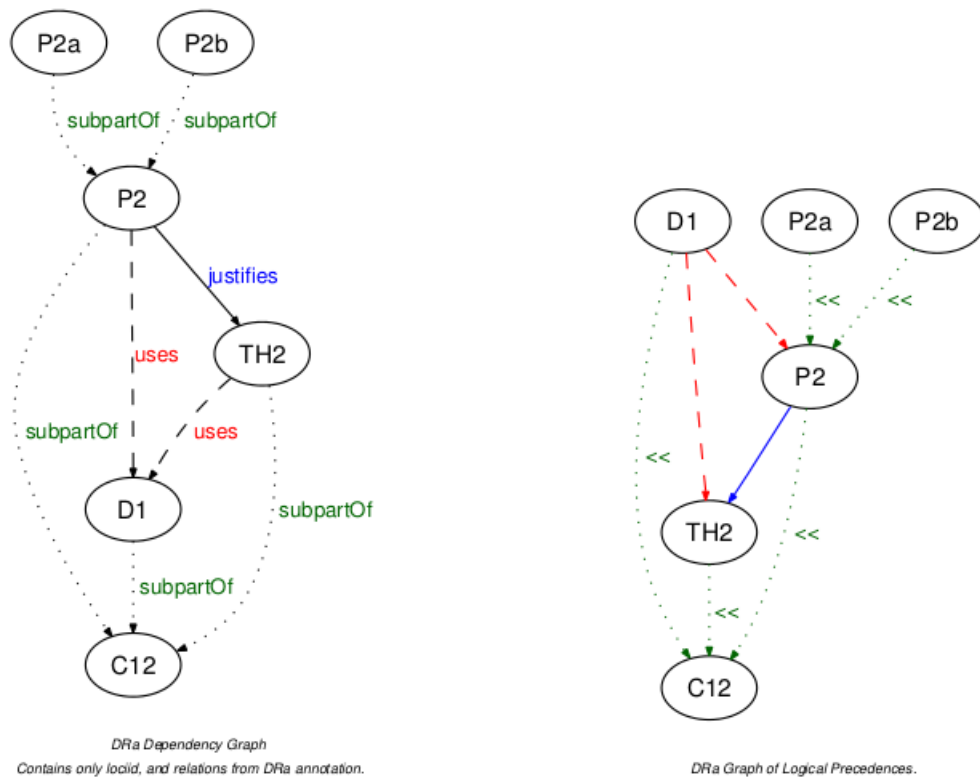
Figure 1.12: Two graphs automatically generated from DRa annotations: The dependency graph (DG) and the graph of logical precedence (GoTo).

#### 1.2.6.4  Translating CGa and DRa into Isabelle syntax

This section shows the CGa and DRa annotated document translated into Isabelle syntax. In this thesis we refer to this part as the filled in Isabelle skeleton or halfbaked proof. This document is now legal Isabelle syntax, but is not formally correct. There still may be gaps in the reasoning or particular proofs which need completing. However, this file may now be worked on by an Isabelle expert to finish off the final proof. The Isar command sorry tells Isabelle to skip a proof-in-progress and treat the goal under consideration to be proved.

Note: Translating just the DRa annotations gives the Isabelle Proof Skeleton and translating the CGa **and** DRa annotations gives the halfbaked proof.

```
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"
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
```

Figure 1.13: Automatically generated Isabelle skeleton using zglscga and DRa annotations.

### 1.2.6.5 Finishing of the halfbaked proof

The halfbaked proof still needs some extra bits of information to finish the final proof. This section removes the '`sorry`' commands and completes the proofs through the document so that it is fully parseable by Isabelle. This final step is done entirely by hand and there is no automation to help the user finish this last step.

```
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"
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)
```

```
      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"

      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)"
```

```
            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
6 218 qed
```

Figure 1.14: Manually finished Isabelle proof.

## 1.2.7 Conclusion

This section provides detailed information on the framework this thesis will be based on. We have described a brief history of where the original MathLang framework stemmed from and an overview of the methodology. We describe the CGa, DRa in detail and give a brief view of the TSa which has been used to formalise the common mathematical text which is what mathematical documents are usually written in. We have shown examples of the documents produced in each step whether they were produced automatically or manually. A full worked example was shown from the original text into a fully parseable and checked Isabelle proof.

We will use the same steps to check the correctness of a software specification albeit each step in the MathLang framework will be adapted, changed and implemented to suit the software specification syntax.

## 1.3   Formal Methods and Languages

Formal methods are usually used to assist in formalising in a variety of texts including systems, software and even language itself.

**Definition 1.3.1** (Formal Language). *A language designed for use in situations in which natural language is unsuitable, as for example in mathematics, logic, or computer programming.*[5]

**Definition 1.3.2** (Formal Specification). *The specification of a program's properties in a language defined by a mathematical logic.*[6]

**Definition 1.3.3** (Formal methods). *Mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems.*[7]

Formal methods are mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems [19]. Specifications are a collection of statements describing how a proposed system should act and function. Formal specifications use notations with defined mathematical meanings to describe systems with precision and no ambiguity. The properties of these specifications can then be worked out with more confidence and can be described to the customers and other stakeholders. This can uncover bugs in the stated requirements which may not have found in a natural language specification. With this, a more complete requirements validation can take place earlier in the development life-cycle and thus save costs and time of the overall project. The rigour using formal methods eliminates design errors earlier and results in substantially reduced time [23].

### 1.3.1   A brief history of formal methods

The first known formal language is thought to be used by Frege in his Begriffsschift (1879), Begriffschift meaning 'concept of writing' described as 'formal language of

---

[5]www.dictionary.com/browse/formal-language
[6]www.wiki.c2.com/?FormalSpecification
[7]www.fmeurope.org/?page_id=2

pure thought'. Frege formalised propositional logic as an axiomatic system.

Formal methods then grew in the following:

- 1940's, Alan Turing annotated the properties of program states to simplify the logical analysis of sequential programs

- 1960's Floyd, Hoare and Naur recommended using axiomatic techniques to prove programs meet their specification.

- 1970's Dijkstra used formal calculus to aid development of non-deterministic programs

Formal methods are used today are just as important as when they were used before. Formal methods have a large presence in academia and have also made their way into various industries to prevent design flaws in high integrity systems. Previous design errors have been found in systems such as the Therac-25 machine, which was used for radiation therapy produced by Atomic Energy of Canada Limited in 1982. It was involved in multiple incidents in which patients were give massive overdoses of radiation [6]. Another major fault which led to disastrous results was NASAs Checkout Launch and Control System (CLCS) cancelled 9/2002 [8].

## 1.3.2 Types of formal methods

Today there are many different types of formal methods used both in industry and academia. Specification languages are expressive languages with general proof methods, such as VDM, Z, B. Another type of formal method could be program correctness proofs which associates logical inference rules with programming syntax, e.g. Hoare triples and Gries Methodology.There may also be model based approaches to formal methods, which are domain specific languages with precise algorithms for correctness proofs, e.g. Temporal logic [51], Fuzzy logic.

Formal methods are used to precisely communicate specifications and the function of programs. They are also used to ensure the correctness of systems particularly safety critical systems.

---

[8]www.spaceref.com/news/viewnews.html?id=475

Formal methods have been a success in a variety of projects. For example, in the Sholis project [29], using a formal specification was most effective for fault finding, therefore if the specifications are correct, then the program implemented should then in turn contain less errors if it follows the correct specification. King, Hammond, Chapman and Pryor's paper [29] was based on the SHOLIS defence system. It highlighted the importance of having a formal specification on a system to check for errors. It was found that the Z proof was the most cost effective for fault finding. The Z specification found 75% of the total faults for the system. Since Z specifications are important for finding faults in SIL4 systems (based on the Sholis project), then checking the correctness of the Z specification is itself very important. Note that the specifications found 75% of errors and not 100%. As human error can still occur in formal specifications, using the ZMathLang approach may increase the percentage of errors found.

Another case study where formal methods have been used in industry is the NASAs Mars Science Laboratory Mission (MSL) [7]. This system relied on various different mechanisms to command the spacecraft from earth and to understand the behaviour of the rover and spacecraft itself. The paper suggested that "*test engineers cannot eyeball the hundreds of thousands of events generated in even short tests of such a complex system*". Therefore runtime verification using formal specification offered a solution to this problem.

A paper reflecting on industry experience with proving properties in SPARK **??**, describes a programming language and verification system that will offer sound verification for programs. It states that SPARK and the use of proof tools remain a challenge (published in 2014) as the 'adoption hurdle' is perceived too high. Customers and regulators have taken a variety of stances on static analysis and theorem provers. Where some places in industry have adopted the idea others remain sceptical. Hopefully this thesis will present an idea on how formal analysis could be simplified and broken up into smaller more understandable steps and thus would allow more users to take on the idea.

Despite these advantages some managers sometimes argue the cost of producing

a system using formal methods do not cover the costs. However the rigour using formal methods eliminates design errors earlier and results in substantially reduced time. Investing more effort in specifying, verifying and testing will benefit software projects by reducing maintenance costs, higher software reliability and more user-responsive software [14].

So far we have seen what are the different types of formal methods, where formal methods have been used, and why some people are still reluctant to use them. So what needs to be done to make formal methods industrial strength? Nirmal Pandey [42] suggests the following:

- Bridge gap between real world and mathematics

- Mapping from formal specifications to code (preferably automated)

- Patterns identified

- Level of abstraction should be supported

- Tools needed to hide complexity of formalism

- Provide visualization of specifications

- Certain activities not yet formulizable methods

- No one model has been identified which should be used for software)

### 1.3.3    Conclusion

In this section we have identified the differences between formal methods, formal languages and formal specification. We have seen how formal methods originated from mathematics and how it grew over time to become what it is today. We see a connection with Frege's work on mathematical notation and identified it as the first formal notation found in history. We identified there are a variety of formal methods and new methods are even being developed today to comply with the systems in questions. For example MSL had it's own specification language developed for the system in order for the system to be verified. Despite all the advantages, some

managers are still reluctant to use formal methods in their system development and thus as Pandey suggested there still needs to be some work done on making formal methods industrial strength. One of his suggestions was that tools are needed to hide complexity of formalisms and to provide visualization of specifications, which this thesis addresses.

## 1.4 Z Syntax and semantics

In this section we give a brief overview on the core fundamentals of the Z formal language. We outline a basis of propositional and predicate logic and describe how Z syntax is formed to specify a system.

### 1.4.1 Introduction to Z

Z is based on predicate Calculus, Zermelo-Frankel set theory as we introduced at the beginning of this chapter in section 1.1.1.

It is a particular formal method which was developed to specify the new Customer Information Control System (CICS) functionality [49]. The set theory includes standard set operators, set comprehensions, Cartesian products and power sets. Z also has other aspects such as schemas which are used to group mathematical objects and their properties. The schema language can be used to describe the state of a system and ways in which that state may change [57].

### 1.4.2 Propositional and predicate logic

Z specifications are built using predicate and propositional logic.

#### 1.4.2.1 Propositional logic

Propositional logic works with statements which must be either true or false but can not be both. The following are propositional statements:

- A tree is green

- A tree has leaves

- All plants have flowers

Propositions can be connected in a variety of ways. Figure 1.2 shows a table of logical connectors in order of operator precedence.

| ¬ | negation | **not** |
|---|---|---|
| ∧ | conjunction | **and** |
| ∨ | disjunction | **or** |
| ⇒ | implication | **implies** |
| ⇔ | equivalence | **if and only if** |

Table 1.2: Logical connectors giving the symbol, its name and pronunciation.

We can now build compound propositions, which are propositional statements joined together using these connectors, e.g.

- the glass is full ∧ the glass is clear

- the phone isn't working ∨ the phone battery has died

- the sun is shining ⇒ I don't need a raincoat

### 1.4.2.2 Predicate logic

Predicate logic allow us to make statements which describe properties that must be satisfied by every, some or no objects in some universe of discourse. Examples are:

- Every plane can fall out of the sky.

- At least one cloud has a silver lining.

- Jake knows all rugby players in the Scottish team.

To formalise such statements we require a language of predicate calculus. A predicate is a statement with a place for an object. A predicate can turn into a proposition once we put an object in it, therefore we can not say whether it is true or false once the predicate has filled in mission information. For example we can say '$x > 5$' is a predicate but not a proposition until we know what '$x$' is. We can make

a proposition out of '$x > 5$' by putting a *quantifier* with it. So we can say, 'there is an $x$ which is larger than 5'. This is written formally as '$\exists\, x > 5$'. This statement can now be written in our Z syntax.

We can now formalise one of our previous predicate statements:

$\exists\, c : CLOUD \bullet c$ *has silver lining* This statement says that there exists $c$ which is a cloud and $c$ has a silver lining.

### 1.4.3 Sets and Types

A *set* is a collection of distinct objects called *elements*. For example the set of all people. Every expression in a Z specification belongs to a set called its *type* and whenever we introduce a variable we must declare its type [16]. For example:

$[HUMAN]$ *the set of all humans*

$\mathbb{N}$ *the set of all natural numbers*

Another way of introducing types is using a *free type definition*, where once enumerates the names of the elements in that type, for example:

$SHAPES ::= square \mid circle \mid triangle$

$PLANTS ::= tree \mid shrub \mid herb$

In Z specification, we introduce variables before they are used in the expressions (predicates) by a means of writing a *declaration*. A declaration can introduce either one or multiple variables. For example:

$ralph : HUMAN$

$a, b, c : \mathbb{N}$

### 1.4.4 Structure of a Z specification

In the Z notation there are two languages: the mathematical language and the schema language. The mathematical language is used to describe various aspects

of a design: object, and the relationships between them. The schema language is used to structure and compose descriptions: collecting pieces of information and naming them for re-use. A schema consists of two parts: a *declaration part* and a *predicate part*. The *declaration part* consists of declared variables and the *predicate part* describes the variable values. We can write a schema either horizontally (figure 1.15) or vertically (figure 1.16).

$$Schema\ Name \mathrel{\widehat{=}} [declarations \mid predicates]$$

$$\boxed{\begin{array}{l} \underline{Schema\ Name} \\ declarations \\ \hline predicates \end{array}}$$
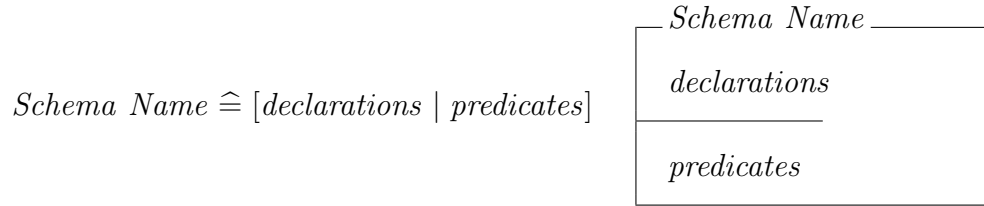
Figure 1.15: An example of a schema written horizontally.

Figure 1.16: An example of a schema written vertically.

If we wanted a property of some system which consists of two variables $x$ and $y$ and state that $x$ must be smaller than $y$ then we can write:

$$\begin{array}{|l} \underline{xLessThany} \\ x : \mathbb{N} \\ y : \mathbb{N} \\ \hline x < y \end{array}$$

We can also introduce global variables by means of an *axiomatic description* in Z. These global variables may be referred to throughout the specification. An example this Z construct is as follows:

$$\begin{array}{|l} k : \mathbb{N} \\ \hline k = 100 \end{array}$$
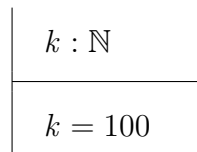
With this axiomatic description in place within the Z specification means that

whenever the global variable $k$ is referred to, it will always represent the natural number 100.

The full language of Z can be explored in [44], [16] and [57].

### 1.4.5   A full example in Z

Here is a small schema which tells a story about a child called Ralph and his mum. We declare the types *OBJECT* which is a type containing all objects. We also declare the type *EMOTION* which consists of 3 emotions: *happy*, *sad* and *angry*

$$[OBJECT]$$
$$EMOTION ::= happy \mid sad \mid angry$$

In this first schema we declare the state variables which is ralph and mum who are of type human, colourIn which maps a human to an object (representing that particular human is colouring in that object) and feeling which maps a human to an emotion (representing that human is feeling that emotion).

In the state we say that ralph is always feeling happy

$$
\begin{array}{|l}
\hline
\text{ColouringIn} \\
\hline
ralph, mum : HUMAN \\
colourIn : (HUMAN \nrightarrow OBJECT) \\
feeling : (HUMAN \nrightarrow EMOTION) \\
\hline
feeling(ralph \mapsto happy) \\
\hline
\end{array}
$$

In this schema we say that when ralph is colouring in a picture, mum is feeling happy. We use a $\Delta$ to signify this schema changes the current state.

```
┌─ ColourPicture ─────────────────────────────────
│  ΔColouringIn
│  picture : OBJECT
│ ─────────────────────────────────────
│  colourIn(ralph ↦ picture)
│  feeling(mum ↦ happy)
└──────────────────────────────────────
```

However in this schema, we say we say when ralph is colouring the wall, then mum is angry.

```
┌─ ColourWall ─────────────────────────────────
│  ΔColouringInwall : OBJECT
│ ─────────────────────────────────────
│  colour(ralph ↦ picture)
│  feeling(mum ↦ angry)
└──────────────────────────────────────
```

### 1.4.6  Conclusion

In this section we gave a brief introduction to Z what it is made up of. We give a short overview of propositional and predicate logic as well as defining the structure of a Z specification. We give a full example of a Z specification and explain each part and what it means. In the next section we identify various proving systems which currently exist for mathematics, other formal methods and Z.

## 1.5  Proving systems

Before the day's of mechanical computers the only way to check a mathematical documents logic was correct was using hand written proofs. Then one can ask another mathematician such as a peer or an expert in the field to check over ones mathematics to prove it was correct. However even with more than one mathematician checking over the mathematical logic, one could not be 100/

Once mechanical computers were developed, work started on proof assistants

(or proof checkers) and automated theorem provers. Checking the correctness using automated theorem provers adds an extra level it also adds a level of certainty that the mathematics written is correct, thus it is a good idea to check the correctness of systems which are or high integrity.

### 1.5.1 Levels of Rigour

Depending on the integrity of the system being developed there are various formal techniques which could be used to mathematically verify the system design and implementation satisfy the functional and safety properties. These specifications and verifications may be done using various degrees of rigour such as:

- Level 1 represents the use of mathematical logic to specify the system.

- Level 2 uses pencil-and-paper proofs.

- Level 3 consists of formal proofs being checked by a mechanical theorem prover.

Level 1 can be done using various levels of abstract to describe the system. One abstract level would be to describe the abstract properties of the system and another abstract level could be to describe the implementation in an algorithmic way.

Level 2 goes beyond level 1 in that the pencil-and-paper proofs are used to describe that the lower level properties imply the abstract properties in level 1.

Level 3 is the highest level of rigour one can do to check the proofs of a specification. It consists of using an autonomic theorem prover to confirm that all proofs are valid. The level 3 process of convincing a mechanical prover is really a process of developing an argument for an ultimate sceptic who must be shown every detail [28].
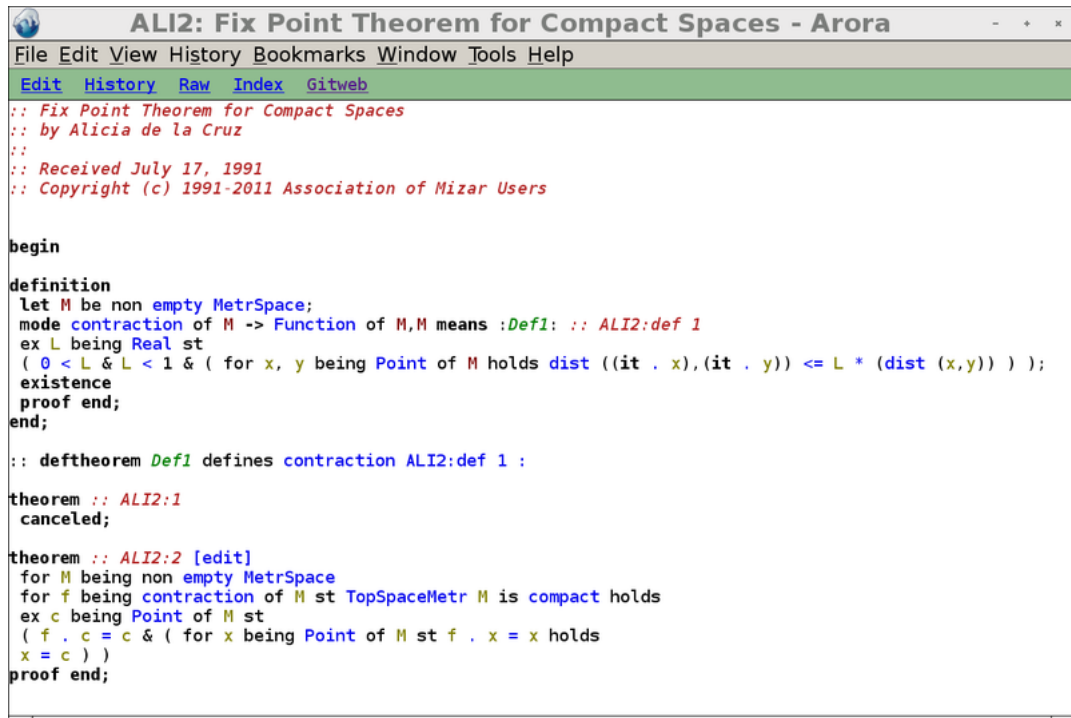
### 1.5.2 Proving systems for maths

In section 1.1.1 we saw how mathematics has evolved and become the complex system it is today. Since the rise of computers the ability to check some of these old and new theories automatically has been possible.

Mizar [4], Isabelle [50] and Coq [48] are types of **Proof assistants** (or proof Checkers). Whilst PVS and Vampire are types of **theorem provers**. Both proof assistants and theorem provers can be classed as 'proof systems'. Each proof system provides it's own formal language for writing mathematics based on some foundation of logic.

The first project which used a 'computer' for automated verification dates back to 1967. N.G. de Bruijn started the AutoMath Project (described in section 1.1.1). A few years Later Trybulec created his project Mizar [4], where he aimed to develop a computerised assistant to edit mathematical papers.

From 1989 the works of the Mizar project worked on improving the Mizar system and developing the database for mathematics. The Mizar Language is a formal language which is derived from the mathematical vernacular. The author wanted to design a language readable for mathematics and rigorous enough to enable processing and verifying by computer software.



Figure 1.17: Screenshot of the Mizar theorem prover [4].

Isabelle [50] is a generic proof assistant. It accepts a variety of mathematics into it's syntax and also has it's own formal language and provides tools for those formulas

in a logical calculus. The main application if the formalization of mathematical proofs and formal verification which includes proving the correctness for computer hardware or software.
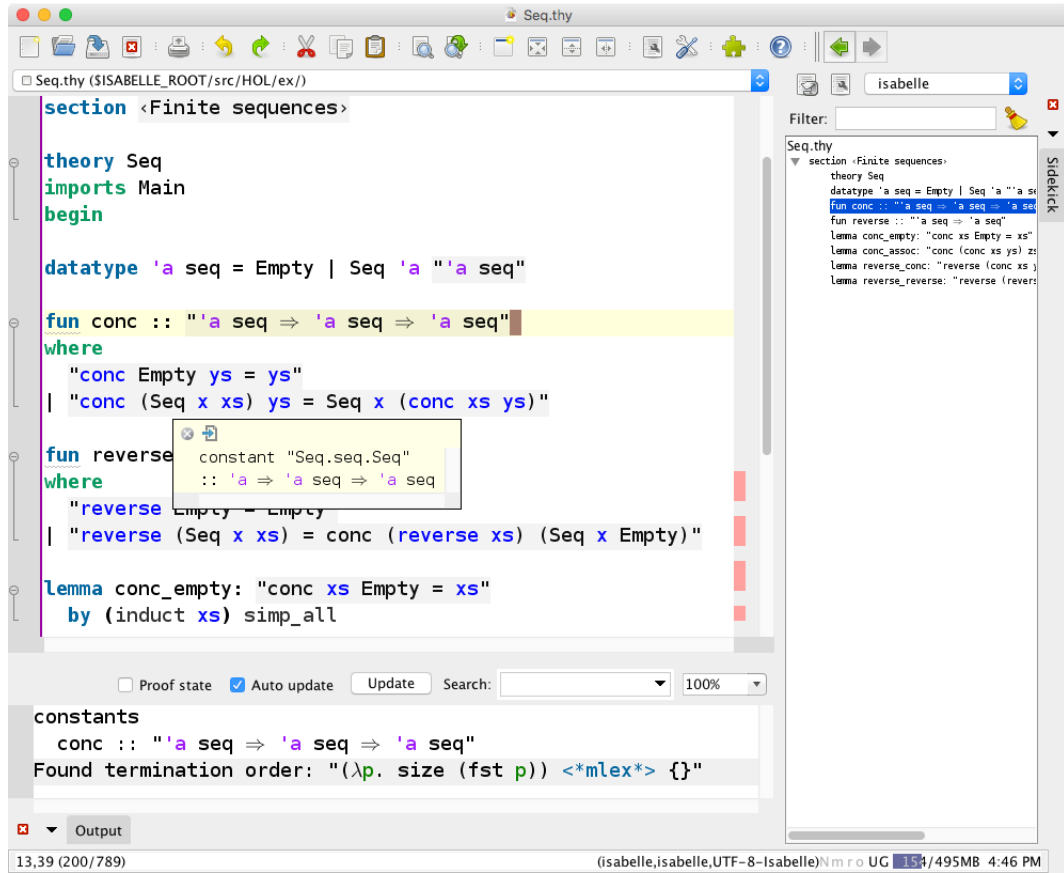


Figure 1.18: Screenshot of the Isabelle theorem prover [50].

A third proof assistant is Coq [48], which also provides it's own formal language to write mathematical definitions, executable algorithms and theorems for semi-interactive development of machine checked proofs. Although it names itself a 'formal proof management system', like Isabelle and Mizar it is used to verify and check properties of software and hardware systems and the formalization of mathematics.

Figure 1.19: Screenshot of the Coq theorem prover [48].

There are many other proof assistants for general mathematics (Nuprl, Otter etc.) and generally they all support checking for correctness of mathematical theories. However, a disadvantage of using these systems is the enormous expense of formalisation in any proof system.

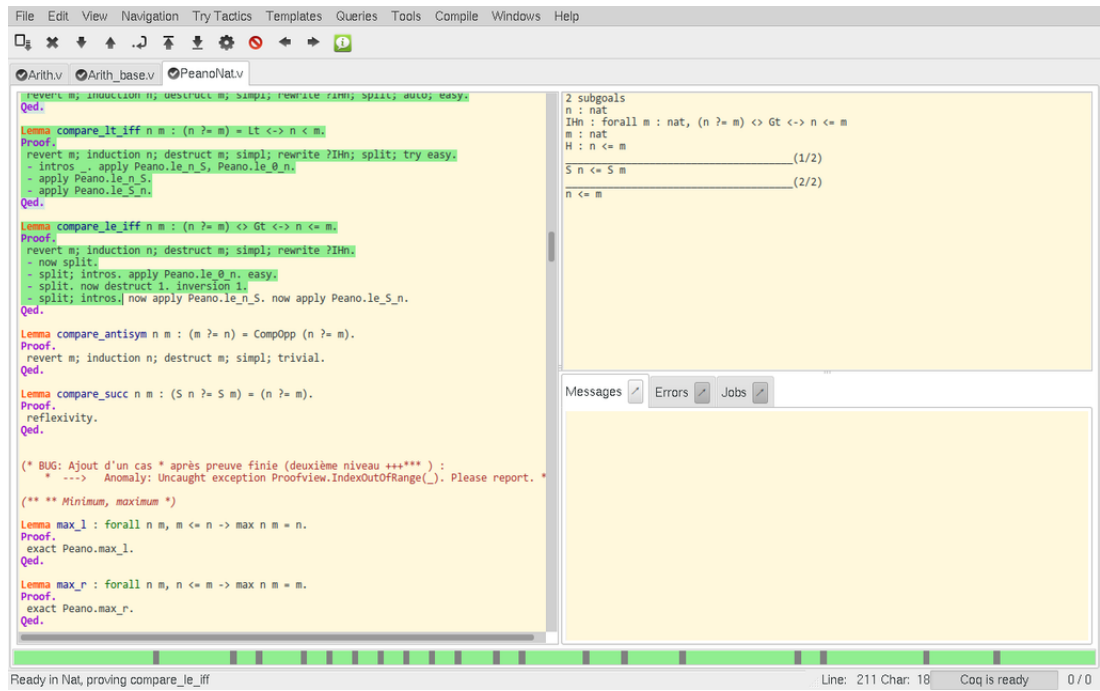A mathematician who wants to verify the logic of a document has to decide on a proof system to use. All proof systems have their own advantages and disadvantage as well as the mathematician then needs to learn and create a document with the syntax of his chosen proof system.

Also, not all proof system have meaningful support for the common mathematical language or a generic formal specification language. There may be proof systems built for specific formal specification language but not one which supports all. The rigid structure and rules for formal documents created in proof systems are more closer to a computer programming language then to natural language. An exception of this is Mizar, however even natural language written in Mizar has to be structural and rigid.

In summary, computerising a any form of mathematical document using a proof system is similar to writing a computer program. Where the formal language of

the proof system corresponds a programming language. Therefore, mathematicians require to have some form of computer programming knowledge. A proof system can check the logical correctness of a document if it detailed enough for the software to parse, therefore the computerisation of the mathematical document becomes labour intensive and tedious for the author to write.

### 1.5.3 Proving systems for specific formal method

Dafny [35] features modular verification, so that the separate verification of each part of the program implies the correctness of the whole program. This is similar to ZMathLang, where Dafny checks different parts of the text and thus confirms correctness of the full text, ZMathLang checks the correctness of the text through different levels of rigour to imply and fully correct specification.

Dafny was able to do a proof for the Schorr-Waited Algorithm, however the writer states that the loop invariants are complicated because they are concrete. A refinement approach such as Jean-Raymond Abrials [2] may be preferred in this case.

Another attempt at checking for correctness was written by Rex L.Page in Engineering Software Correctness [41]. A general theme within this paper, is that design and quality are important in engineering education. When teaching students how to create programs, it is not enough just to teach them how to develop software but to how develop good quality software. This paper describes experiments with the use of ACL2 (a subset of lisp), which is embedded on mechanical logic to focus on design and correctness. Using ACL2 emphasises the importance of software design and correctness.

ALC2 is coded therefore users must know how to code software to formulate proofs. The intention of ZMathLang is to allow many people in the development team to be able to formulate proofs such as project managers, designers, engineers etc. Therefore no coding is necessary and no new programming language is needed.

PVS (Prototype Verification System) [40] is an environment for constructing specifications and developing proofs which have been mechanically verified. PVS

has it's own specification language, which engineers would need to learn as well as using the environment for proofs. Type checking is undecideable for the PVS type system. The PVS also provides a language for defining high-level inference strategies.

### 1.5.4 Proving Systems specific for Z

A tool which analysis the Z notation is Hol-Z [10], which is also a proof environment for Z. Hol-Z is embedded in Isabelle/HOL therefore it provides a Z type checker, documentation facilities and refinement proofs with a theorem prover. The Z specification is implemented in LaTeX then typed checked using an external plug in Zeta, it is then transformed into SML files to be added into the Hol-Z theorem prover environment. The user will need to have some good expertise in using the Hol-Z proof environment in order to fully prove the specification.

Fuzz [45] is another example of a type-checker for the Z language. It includes style files for LaTeX and checks for the logical correctness and Z type correctness of Z specifications. This is different to the Z Core Grammatical aspect (ZCGa) type checker as the weak types in ZCGa check for the grammatical correctness and not the full logical correctness of Z. Therefore the grammatical correctness of partially formal specification can also be checked. The ZMathLang framework presented in this thesis uses the 'zed' LaTeX style package to typeset the Z specifications in the documents.

There are many other tools for Z which can be found on the Z Notation Wikia page [9]. For this thesis we will concentrate on translating Z specifications into theorem prover Isabelle [50]. Isabelle is a generic proof assistant which allows mathematical formulas to be expressed in a formal language and includes numerous contributions worldwide. It contains a large mathematical tool-kit (majority of Z can be represented in Isabelle) and has a lot of support in forms of documentation and online. It is distributed for free, easy to find, download and install and is regularly updated. The original MathLang has translated mathematical texts into Isabelle already ([32]) therefore we can use parts of that research to aid the research in this

thesis.

## 1.5.5 Other properties to prove

**Definition 1.5.1.** *A logical formula associated to a correctness claim for a given verification property. The formula is valid if and only if the property holds. The correctness of the property under verification is delegated to proving the correctness of the new formula [34].*

Therefore a proof obligation is a logical formula which the specifier must show to be a consequence of the specification so that a specification can be taken to be acceptable. In a more pragmatic sense proof obligations may be viewed as what the developer of a specification is obliged to prove in order to confirm that development is consistent.

Woodcock and Cavalcanti [56] use the alphabetised relational calculus to give denotational sematics to different constructs from programming patterns. This paper describes '*healthiness conditions*' which identifies properties that characterise theories. Each one of these healthiness conditions represents a fact about the computational model for the programs being studied.

**Example 1.5.1.** *The variable 'clock' is an observation of the current time which is always moving onwards. The following predicate 'B' specifies the clock variable:*

$$B \mathrel{\widehat{=}} clock \leq clock'$$

The healthiness condition described in this paper are specific proof obligations for the concept of Unifying theories of programming (UTP). The semantic model of UTP is presented a Z specification. Therefore the healthiness conditions for the specification would in one way be checking for the correctness of the UTP model. In a similar way, it is important to add healthiness conditions or as we call them in this thesis, 'safety properties' in order to check each individual specification for various types of correctness.

Stepney describes two proof project written in Z in her paper a tale of two proofs

[46]. She explains how the proof process is deeply affected by **why** something is being proved, **what** is being proved and **how** the finished proof is to be presented. Stepney suggests that the proofs for specifications themselves do not have to be deep but the workings of what to proves can add to the labour. It is also important to keep in mind how deep the customer wants the proof and what level of assurance they need. She highlights 5 different points to prove:

1. **Consistency checks**: Prove that your specification is consistent and that it has a model.

2. **Sanity Checks**: In a 'state and operations' style specification, prove that the state invariants are satisfied throughout and that the precondition of each operation is not *false.*

3. **Emergent properties of a single specification**: Make explicit as a theorem some desired or suspected property of the specification, then prove it holds.

4. **Required properties of a single specification**: If some property is required to hold of the specification, and the specification has captured it implicitly, it needs to be made explicit and shown to hold.

5. **Properties across specifications**: Prove that a certain relationship holds between two specification such as the refinement relationship.

Figure 1.20: Different points to prove in a specification [46].

Some of the points in figure 1.20 would be fairly easy to automate such as 'consistency checks' and 'sanity checks' however the other 3 points would be difficult to automate as they would all depend on the specification in question and would perhaps need some *extra information* to decipher these properties. For example if one wish to automate proof obligations from point 5 the user would need to implement another specification (such as a refinement specification) and then prove the relationship between the original specification and the new one. In this thesis, we will concentrate on properties described in point 1 which have been automated.

Stepney also explains that if the development process is incremental, it would be worthwhile getting a good structure for a proof up front which you can add more details in later. ZMathLang can automatically generate this first structure which could be added to if needed. It can produce a specification in Isabelle syntax where other properties and details could be added to by the user at a later stage.

Most recently Mark Adams [3] describes that even formalisation itself can be prone to error and therefore even if we do get a fully proven specification, the proof will also need to be checked. He outlines the flyspeck project which assists with the checking of formalised proofs. Adam calls this '*proof auditing*', which adds another step of rigour to the theorem. ZMathLang assists with translating the specification into a theorem prover along with consistency lemmas to prove. The user can then choose to prove these lemmas and the proof audit their theorem. However, even if proof auditing adds another level of rigour it is important to keep in mind who the user is doing the proofs for. As Stepney pointed out in [46], some clients wouldn't need that amount of rigour for their projects and only the proved safety properties may be enough.

Woodcock et al also describes that a specification can be developed in such a way that can lead to a suitable implementation called refinement [57].

To refine a formal specification, more data must be added e.g. how certain calculations should be carried out. He states an abstract specification may leave design choices unresolved and its up to the refinement specification to resolve some of these choices. An example of this is shown in the following:

**Example 1.5.2.**

$$\begin{array}{|l}
\hline
\textit{ResourceManager} \\
\hline
\textit{free} : \mathbb{F}\mathbb{N} \\
\hline
\end{array}$$

*Any resource that is free can be allocated*

```
 ___ Allocate _____
|
|  ΔResourceManager
|
|  r! : ℕ
|  _____
|
|  r! ∈ free ∧ free' = free \ {r!}
|_____
```

If there is more than one resource free in example 1.5.2 then this will class the specification as non deterministic.

Example 1.5.2 can be refined in that if there is more than one resource free, the resource with the lowest number should be allocated. This is shown in the following example:

**Example 1.5.3.**

```
 ___ Allocate _____
|
|  ΔResourceManager
|
|  r! : ℕ
|  _____
|
|  r! ∈ min free ∧ free' = free \ {r!}
|_____
```

Refining an abstract specification which is described in [57] and in [44] is exactly what Stepney points out in point 5 from figure 1.20. To show that this property holds the user would need to produce another specification which refines their original one and then include properties of how they relate to each other. Since each individual specification is different then refinement specifications would be different to. Thus we wouldn't be able to automate this point. ZMathLang aims to assist the user in translating and proving their specification, the proof effort will be focused on properties which can be automated e.g. item 1 in figure 1.20. Items 3, 4 and 5 would be to difficult to automate and therefore out of the scope of this thesis.

Fraser and Banach [20] state that model based formal methods usually come in

the form of many incompatible tools. Therefore they devised a system to combine different techniques called the frog toolkit.

Many verification tools today tend to utilize a single technique and are unable to interact easily with other tools in other kits. A more dynamic approach such as RODIN [15] and Overture [33] are perusing a more flexible approach. ZMathLang will need to follow in the footstep of these two tool-kits in order to be successful and not commit to a particular system until needed. This is why the steps 1-3 of the translation path (figure ??) are generic and can be used to translate the specification in any theorem prover. It is only at step 4 which is where we start committing to Isabelle.

The main aim of the frog tool-kit was to support retrenchment, which is a formal technique used along side refinement. The new tool-kit was created which was able to use a variety of techniques in a single working environment. The frog tool-kit can prove the relationships between multiple specifications (which we see as point 5 in figure 1.20 from Stepney [46]). To do this the user should have at least 2 specifications implemented to prove the relations (usually one is a refinement specification of the first one). These specifications where then written in frog-ccc, a meta language to use within the frog tool-kit. ZMathLang's aim is to prove the properties described as point 1 in figure 1.20 therefore another specification to refine the abstract one is not needed.

Since formal specifications are not executable it is difficult to verify the consistency of the specification. Wen, Miao and Zeng [53] present and approach to generate proof obligations to check for consistency of object Z specifications. Checking for consistency of specifications is described in point 2 in figure 1.20. In their paper, the authors explain two types of proof obligations which check that a object Z specification is non conflictive. They are:

- Existence of initial state

- Feasibility of an operation

The initial state is a state within the state space of the specification which should

exist and satisfy the stateInvariants. In all specifications their should exist a state which initialised the beginning state of the specification.

An operation can transform one state to another. Therefore if the operation is feasible, the pre state (state before the state change) and post state (state after the state change) should always satisfy the state invariant.

We will use the definitions shown in [53] to automatically generate the proof obligations to check for the correctness of our Z specifications as the specifications we use in this thesis are all state based.

In summary there are many different approaches to finding properties to prove about formal specifications. Some which can be easily automated, some which need some more information and some which will need to be written manually. Since we do not expect the users of ZMathLang to be experts in the fields of theorem proving we want to attempt to automate proof obligations which can be easily understood by the user. Then once the user gains some knowledge of their theorem prover, they can add other proof obligations manually. As explained in this section, it is important to note who the user is doing the proofs for, as the customer may not need or want complex proofs. Therefore we shall generate the proof obligations which can be automated and do not require any extra information to be automated.

### 1.5.6 Conclusion

Write conclusion to proving systems

## 1.6 Background Conclusion

### 1.6.1 MathLang for Z

Write conclusion to background chapter

- [47] states what ro do to make formal methods industrial strength

- [31] stating in future work mathlang should be developed to cope with more mathematics (formal spec is a type of mathematics)

- diagram of math text to theorem prover using mathlang + diagram of specification to theorem prover using mathlang

ZMathLang covers items 1, 3, 5, 6, 7 from section 1.3.

# Bibliography

[1] J. Abbott, A. Díaz, and R. S. Sutor. A report on openmath: A protocol for the exchange of mathematical information. *SIGSAM Bull.*, 30(1):21–24, Mar. 1996.

[2] J.-R. Abrial. Event Based Sequential Program Development: Application to Constructing a Pointer Program. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.

[3] M. Adams. Proof auditing formalised mathematics. *Journal of Formalized Reasoning*, 9(1):3–32, 2016.

[4] Andrzej Trybulec. Mizar. `http://www.mizar.org`, December 2016.

[5] R. Arthan. Proof Power. `http://www.lemma-one.com/ProofPower/index/`, February 2011.

[6] S. Baase. *A Gift of Fire: Social, Legal, and Ethical Issues for Computers and the Internet*. An Alan R. Apt book. Pearson Education, 2003.

[7] H. Barringer, A. Groce, K. Havelund, and M. H. Smith. An entry point for formal methods: Specification and analysis of event logs. In *Proceedings FM-09 Workshop on Formal Methods for Aerospace, FMA 2009, Eindhoven, The Netherlands, 3rd November 2009.*, pages 16–21, 2009.

[8] N. Bourbaki. *General topology. Chapters 1-4*. Elements of mathematics. Springer-Verlag, Berlin, Heidelberg, Paris, 1989. Trad. de : Topologie gnrale chapitres 1-4.

[9] J. Bowen. Formal Methods Wiki, Z notation. `http://formalmethods.wikia.com/wiki/Z_notation`, July 2014.

[10] A. D. Brucker, H. Hiss, and B. Wolff. HOL-Z 2.0: A Proof Environment for Z-Specifications. *Journal of Universal Computer Science*, 9(2):152–172, feb 2003.

[11] R. Butler, G. Hagen, J. Maddalon, C. Muñoz, A. Narkawicz, and G. Dowek. How formal methods impels discovery: A short history of an air traffic management project. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 34–46, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[12] K. S. Candan, H. Liu, and R. Suvarna. Resource description framework: Metadata and its applications. *SIGKDD Explor. Newsl.*, 3(1):6–19, July 2001.

[13] G. Cantor. Ueber eine Eigenschaft des Inbegriffs aller reellen algebraischen Zahlen. *Journal fr die reine und angewandte Mathematik*, 1874(77):258–262, 1847.

[14] W. Chantatub. *The Integration of Software Specification Verification and Testing Techniques with Software Requirements and Design Processes*. PhD thesis, University of Sheffield, 1995.

[15] J. Coleman, C. Jones, I. Oliver, A. Romanovsky, and E. Troubitsyna. RODIN (rigorous open development environment for complex systems). In *EDCC-5, Budapest, Supplementary Volume*, pages 23–26, Apr. 2005.

[16] E. Currie. *The Essence of Z*. Prentice-Hall Essence of Computing Series. Prentice Hall Europe, 1999.

[17] N. de Bruijn. The mathematical vernacular, a language for mathematics with typed set. *In Workshop on Programming Logic*, 1987.

[18] R. Fitzpatrick and J. Heiberg. *Euclid's Elements*. Richard Fitzpatrick, 2007.

[19] Formal Methods Europe, L-H Eriksson. Formal methods europe. `http://www.fmeurope.org/?page_id=2`, May 2016.

[20] S. Fraser and R. Banach. Configurable Proof Obligations in the Frog Toolkit. In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10-14 September 2007, London, England, UK*, pages 361–370. IEEE Computer Society, 2007.

[21] J. A. Gallian. *Contemporary Abstract Algebra*. Houghton Mifflin, 2002.

[22] F. Gottlob. Die grundlagen der arithmetik. *Eine logisch mathematische Untersuchung ber den Begriff der Zahl, Bres*, 292, 1884.

[23] J. Groote, A. Osaiweran, and Wesselius2. Benefits of Applying Formal Methods to Industrial Control Software. Technical report, Eindhoven University of Technology, 2011.

[24] F. Kamareddine, M. Maarek, K. Retel, and J. B. Wells. Gradual computerisation/formalisation of mathematical texts into Mizar. In *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, pages 81–95. Springer-Verlag, 2007.

[25] F. Kamareddine, M. Maarek, and J. B. Wells. Toward an Object-Oriented Structure for Mathematical Text. In M. Kohlhase, editor, *MKM*, volume 3863 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2005.

[26] F. Kamareddine and R. Nederpelt. A refinement of de Bruijn's formal language of mathematics. *Logic, Language and Information*, 13(3):287–340, 2004.

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

[28] M. Khosrow-Pour. *Encyclopedia of Information Science and Technology*. Information Science Publishing, 2009.

[29] S. King, J. Hammond, R. Chapman, and A. Pryor. Is Proof More Cost-Effective Than Testing? *IEEE Trans. Software Eng.*, 26(8):675–686, 2000.

[30] M. Kohlhase. *OMDoc – An Open Markup Format for Mathematical Documents [Version 1.2]: Foreword by Alan Bundy (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[31] R. Lamar. *A Partial Translation Path from MathLang to Isabelle.* PhD thesis, Heriot-Watt University, 2011.

[32] R. Lamar, F. Kamareddine, and J. B. Wells. MathLang Translation to Isabelle Syntax. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Calculemus/MKM*, volume 5625 of *Lecture Notes in Computer Science*, pages 373–388. Springer, 2009.

[33] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The overture initiative integrating tools for vdm. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, Jan. 2010.

[34] I. Lee, J. Y.-T. Leung, and S. H. Son. *Handbook of Real-Time and Embedded Systems.* Chapman & Hall/CRC, 1 edition, 2007.

[35] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In E. M. Clarke and A. Voronkov, editors, *LPAR (Dakar)*, Lecture Notes in Computer Science, pages 348–370. Springer, 2010.

[36] Leslie Lamport et al. The latex project. `http://www.latex-project.org/`, September 2016.

[37] M. Maarek. *Mathematical documents faithfully computerised: the grammatical and test & symbol aspects of the MathLang Framework.* PhD thesis, Heriot-Watt University, 2007.

[38] L. Mastin. 17th century mathematics - Leibniz. `http://www.storyofmathematics.com/17th_leibniz.html`, 2010.

[39] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected papers on Automath*, volume 133. Elsevier, 1994.

[40] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[41] R. L. Page. Engineering Software Correctness. *J. Funct. Program.*, 17(6):675–686, 2007.

[42] N. Pandey. Formal methods slides. Slides, 2011.

[43] K. Retel. *Gradual Computerisation and Verification of Mathematics: Math-Lang's Path into Mizar*. PhD thesis, Heriot-Watt University, 2009.

[44] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[45] M. Spivey. The fuzz manual. *Computing Science Consultancy*, 34, 1992.

[46] S. Stepney. A tale of two proofs. In *BCS-FACS third Northern formal methods workshop, Ilkley*, 1998.

[47] C. V. Stringfellow. Formal methods presentation. Formal methods universty presentstion, September 2016.

[48] Thierry Coquand and Grard Huet. Coq. `http://coq.inria.fr/`, September 2016.

[49] I. UK. *Customer Information Control System (CICS) Application Programmer's Reference Manual*. White Plains, New York.

[50] University of Cambridge and Technische Universitat Munchen. Isabelle. `http://www.isabelle.in.tum.de`, May 2015.

[51] T. university of western Australia. Software testing and quality assurance: Lecture 1. `http://teaching.csse.uwa.edu.au/units/CITS5501/Lectures/L17.pdf`.

[52] E. W. Weisstein. Zermelo-fraenkel set theory. `http://mathworld.wolfram.com/Zermelo-FraenkelSetTheory.html`.

[53] Z. Wen, H. Miao, and H. Zeng. Generating Proof Obligation to Verify Object-Z Specification. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006), October 28 - November 2, 2006, Papeete, Tahiti, French Polynesia*, page 38. IEEE Computer Society, 2006.

[54] A. Whitehead and B. Russell. *Principia Mathematica*. Number v. 2 in Principia Mathematica. University Press, 1912.

[55] F. Wiedijk. Formal proof sketches. In *In Proceedings of TYPES03, volume 3085 of LNCS*, pages 378–393. Springer, 2002.

[56] J. Woodcock and A. Cavalcanti. A tutorial introduction to designs in unifying theories of programming. In *Integrated Formal Methods*, pages 40–66. Springer, 2004.

[57] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[58] J. Woods. *Aristotle's Earlier Logic (2nd Edition)*. College Publications, 2014.

[59] C. Zengler. MathLang- Towards a Better Usability and Building the Path into Coq, First Year Report. Technical report, Heriot-Watt University, November 2008.