# Translating formal proofs into English

Kapelonis Konstantinos 302440
MSc in Computing and Software Technology
University of Wales Swansea

September 20, 2004

**Abstract**

This MSc project deals with the translation process of machine-generated proofs to human readable English text. The implementation is based on the theorem prover Minlog. Minlog is an interactive proof checker written in Scheme. We describe the development of a Minlog submodule which processes the internal representation of the finished proof and outputs a human readable description of it, in English sentences. Since the final goal is the direct inclusion of this output into scientific documents, the suggested target format is LaTeX. A prototype for this submodule already exists, it works however, only for previous versions of Minlog which are now obsolete, and the target format is generic TeX. Our aim is to express the proof in an acceptable manner and at the same time we explore several strategies which make the resulting text more natural for the human reader. Possible extensions to the project include the output to other data representation formats such as the emerging XML standard.

# Contents


**I Motivation and Background** **6**

**1 Introduction** **7**
- 1.1 The importance of formal proofs . . . 7
- 1.2 The Minlog system . . . 9
- 1.3 Motivation . . . 9
- 1.4 Conceptual background . . . 10
- 1.5 Project plan . . . 15
- 1.6 Related work . . . 16

**2 On Scheme** **17**
- 2.1 History of Scheme . . . 18
- 2.2 The flexibility of Scheme . . . 19
  - 2.2.1 Procedural programming style . . . 19
  - 2.2.2 Object oriented programming style . . . 21
- 2.3 Scheme today . . . 24

**3 Overview of the Minlog system** **25**
- 3.1 Introduction to Minlog Logic . . . 26
- 3.2 A sample Minlog session . . . 28
  - 3.2.1 Automatic proof search . . . 29
- 3.3 Storage of proofs . . . 30

**II Specification** **32**

**4 Project Objectives** **33**
- 4.1 Abstraction and integration . . . 34
- 4.2 Following the abstract data types paradigm . . . 35
- 4.3 Building a Minlog module . . . 37

**5 Analysis and Design** **40**
5.1 The initial approach . . . . . 40
5.2 The two-step transformation process . . . . . 41
5.3 Interface definition . . . . . 43
5.4 Design enhancements . . . . . 45

**III Development** **48**

**6 Implementation Process** **49**
6.1 Representation of formulas . . . . . 49
6.2 Proof verbalisation (top-down) . . . . . 52
6.3 Proof analysis . . . . . 54
6.4 Proof verbalisation (mixed reasoning) . . . . . 56

**7 Expression Enrichment** **61**
7.1 Numbering assumptions . . . . . 61
7.2 Grouping multiple assumptions . . . . . 62
7.3 Grouping multiple all eliminations . . . . . 64
7.4 Layout considerations . . . . . 65
7.4.1 Subproofs . . . . . 65
7.4.2 Selective formula listings . . . . . 66
7.5 Induction . . . . . 67

**IV Conclusions** **69**

**8 Evaluation** **70**
8.1 Contributions . . . . . 70
8.2 Correctness . . . . . 71
8.3 Efficiency . . . . . 72
8.4 Extensibility . . . . . 73
8.5 Project review . . . . . 76

**9 Future Considerations** **79**
9.1 Regarding XML . . . . . 79
9.2 Possible extensions . . . . . 81

**V Appendix and References** **83**

**A Example sessions** **84**

A.1 General example . . . . . 84
A.2 The main example of the text . . . . . 86
A.3 Distribution Law . . . . . 87
A.4 Multiple all elimination . . . . . 89
A.5 Nested subproofs . . . . . 91
A.6 The Peirce formula . . . . . 92
A.7 Simple induction on natural numbers. . . . . 95
A.8 Induction on lists . . . . . 97

# List of Figures


1.1 System overview . . . . . 11
1.2 The Object Oriented Programming approach . . . . . 13
1.3 The functional approach . . . . . 14
1.4 Proofs as programs . . . . . 14

4.1 Expressing the proof structure . . . . . 34
4.2 The data abstraction principle . . . . . 36
4.3 A set of abstract data types . . . . . 37
4.4 The module approach . . . . . 38

5.1 The obvious solution . . . . . 41
5.2 Two-step transformation process. . . . . 42

6.1 Processing and presenting Minlog formulas . . . . . 50
6.2 Converting a formula to/from a string representation . . . . . 51
6.3 Forward reasoning (top-down) . . . . . 54
6.4 Mixed reasoning style . . . . . 57

7.1 Merging multiple assumptions . . . . . 63

# List of Tables

2.1 Standards of various programming languages . . . . . 18

3.1 Minlog evolution . . . . . 26
3.2 Minimal logic for Minlog . . . . . 26
3.3 Natural deduction and $\lambda$-calculus . . . . . 31

5.1 Object programming in the module code . . . . . 45

6.1 Formulas supported (connectives and elements) . . . . . 51
6.2 Verbalisation functions . . . . . 60

8.1 Module usage . . . . . 71

# Part I

# Motivation and Background

# Chapter 1

# Introduction

This document is submitted as a partial fulfilment of the Master of Science in Computing and Software Technology 2003/2004 at the University of Wales Swansea. It contains the MSc project dissertation which counts as the second part of the MSc course. The first part consists of the taught modules which took place during the academic year 2003/2004.

In summary the project described is the development of a submodule, of the Minlog interactive proof checker, which is responsible for transforming the mathematical proof from the Minlog specific syntax to a human readable form in the LaTeX typesetting system.

## 1.1 The importance of formal proofs

A subset of all the sentences that surround us, are statements which can be either true or false. Examples:

1. John has a red hat.
2. $1 + 1 = 2$.
3. If I have time, I will go to the cinema.
4. Mary bought today 4 apples and also sent 2 letters by post.
5. If for all natural numbers $n$ we compute $p(n) = n^2 + n + 41$, then $p(n$ is a prime number[MN02].
6. Every map can be coloured with 4 colours so that adjacent regions have different colours. (the four colour theorem).

These statements are called propositions. Some propositions are easy to verify. We can ask John if he has a red hat or not. Even if he has a hat, but with green colour the proposition will be false. The proposition will be true only if John actually has a red hat[1].

Some propositions like number 2 are not true on their own, but rather we have agreed on their truth. We have agreed that $1 + 1$ is 2 in order to have common mathematical foundations. Everyone who uses these mathematical symbols with the same (usual) meaning will also find this proposition true. But there is no reason why someone could not use the same symbols with different meanings and declare that $1+1 = 3$. It is all a matter of interpretation and the way we choose to understand what we see.

Some propositions like number 3 and 4 are composite. They consist themselves of simple propositions. The truth of the whole sentence depends on whether the individual propositions are true or false.

We are more interested in propositions like number 5 and 6. These are propositions which need some additional thought before one can claim that they are true or false. These are the kind of propositions which need a *proof*.

A proof is a formal process which can convince someone that a proposition is true (or false). A proof can be long or short, easy or difficult, but the key point is that it has to be clear, precise and correct[MN02]. Correctness is the primary goal and sometimes it is easy to make mistakes. For example if we use sampling we could claim that proposition 5 is true since we tried it for the first 10 natural numbers and $p(n)$ is indeed prime. The proposition is actually false because for $n = 40$ the result is not a prime number.

The safe approach is to formalise the way propositions are expressed and decide using mathematical arguments on the correctness of a proof. By formalising proofs we can avoid logical gaps or inaccuracies ("everybody knows that", "it is obvious that") that present common traps for humans. With the invention of computers we can actually shift this responsibility to the machine. A program running on a machine can check all steps of a proof easily without getting tired or bored. Repetition is a particular skill where computers really have the advantage over humans.

Using computers as problem solvers has revolutionised the way we deal with mathematical proofs. Researchers have started using computers for proofs that no human would actually try to validate. Unfortunately the resulting proofs are very complicated and can only be verified (again) by computers. Humans have to trust the machines about the correctness of these proofs. The main problems are:

- For a computer-generated proof to be correct, the computer must also

[1]He can also have other hats with different colours.

function correctly.

- For a computer-generated proof to be useful to humans, it must be readable by them.

There is a large debate about these topics. For example a proof of the four colour theorem was so complicated that a large part of it could only be checked (and understood) by computers. And even then a small part of it still had to be checked manually by humans which was a very tedious procedure. Humans are interested in a proof that they can understand themselves [RSST96].

A good solution is to leave the computation process to the machine, but provide some kind of facilities which convert the raw mechanical form of the proof into a legible expression of it (English text). This facility is what this project is all about. The proof checker (machine representation of proofs) is the Minlog system, and the program we develop is a special software module which outputs a formal Minlog proof into English text.

The next two sections focus on the Minlog system. For a discussion of why mathematical proofs are important to Computer Science please see section 1.4.

## 1.2 The Minlog system

Minlog is an interactive proof system[MINLOG]. It was originally developed by Helmut Schwichtenberg in the University of Munich but many other researchers from the logic group of the department[LOGIK] have contributed either code, or documentation on the usage of the system.

Apart from the full manual, there is also a tutorial[CRO4] and a command reference of all the Minlog commands[MREF]. Minlog is written in the functional language Scheme. It can run on any implementation of Scheme which follows the revised report 5 [RRS5], but the preferred Scheme environment is the Petite Scheme from Cadence Research systems[CHEZ6]. Since Petite is cross-platform (Digital Unix/HP-UX/Linux/Windows/AIX/Solaris/IRIX) Minlog can run equally well in the respective platforms.

More information on Minlog can be found in section 3.

## 1.3 Motivation

There are three versions of Minlog at the moment. The latest stable one is Minlog3 and most documentation refers to this. There is the development

version of this named as Minlog3i, and finally there is the latest version of Minlog named as Minlog4 which is not completely backwards compatible with the previous versions.

The last stable version of Minlog (3) included a *texoutput* module which can be used to export the finished proof of a problem into a TEX file. This Tex file along with a small TEX macro file specifically designed for Minlog could be processed by common TEX tools and utilities to produce a human readable description of the proof (in English) in various output formats (HTML/ps/pdf) that TEX supports. Figure 1.1 shows an overview of the full system.

This module is also present in the development version (3i) and is even regarded as a submodule of Minlog rather than an accompanying utility.

Many things changed though, when Minlog 4 appeared. Minlog 4 is not just the next version of Minlog but large parts of the system have been completely rewritten. Unfortunately this means the the old texoutput module is now obsolete and needs to be ported to this new version.

So the goals of the MSc project are:

- to port the texoutput module to Minlog 4.
- to extend the module with additional functionality.
- to support the more user friendly LATEX output instead of the plain TEX.

The next section explains why we need a proof checker program in the first place, and how formal mathematical proofs can be used in real world problems.

## 1.4 Conceptual background

Traditionally the technology press is dominated by commercial programming languages and operating systems news. The hot topic of our days are the so called "Web Services" along with the lingua franca of the Internet XML. The spotlights fall on the two major competing technologies which aim to make Internet programming easier and more robust. The well established Java technology (from Sun Microsystems) is the first option, while the .NET framework (from Microsoft Corporation) represents the latest approach for web applications that run on the Internet.

These systems however are destined to run on non-critical situations where an incorrect implementation is not fatal. Any errors in such applications can at best prevent the users from accessing the (commercial) service,

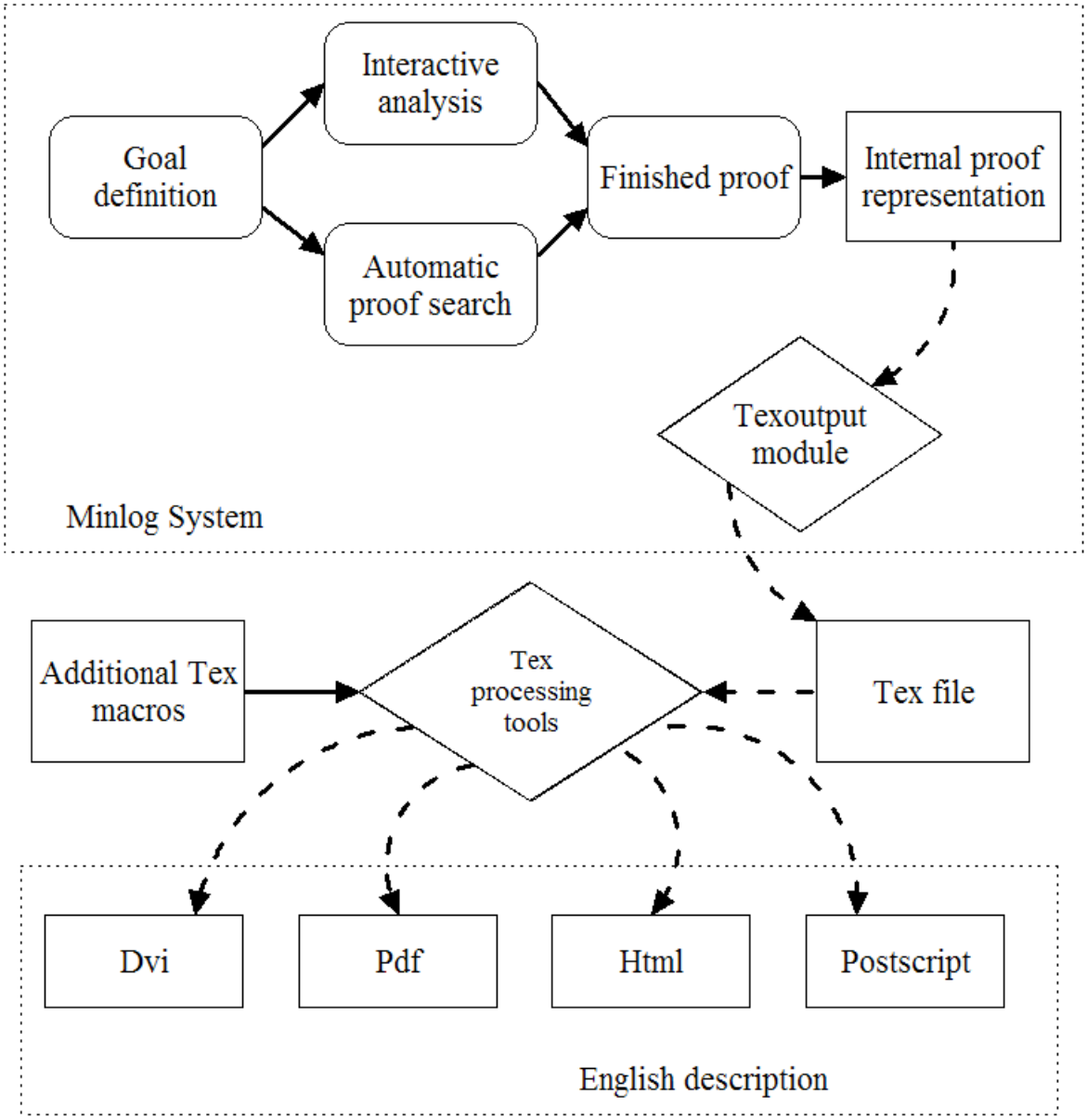


Figure 1.1: System overview

and in the worst case they can compromise the security of the system itself. During the development of such systems the design goals are usually ease of use, painless deployment, efficient integration and even backwards compatibility. Correctness is assumed after extensive testing sessions which of course can reveal mistakes but cannot prove the complete absence of them, not to mention the fact that it is hard to develop test cases for all possible aspects of the system[BU01]. If an error does appear after the deployment of the system, it is fixed in the next version of the system and after the re-deployment of the code the system is assumed to be "correct" again.

There are other systems however, (mostly unknown to the public) where correctness is the ultimate goal and any miscalculations are simply unacceptable. Due to the nature of these systems (mission critical), correctness is the essential feature of the controlling programs. Such systems are installed in nuclear reactors, avionics electronics, hospitals, military installations and even cars. Obviously, errors in these areas can result in the loss of human life. The literature is abundant with real accidents that have been attributed to software errors.

There are even systems (usually running in embedded microprocessors) which are hard real-time, meaning that their response under certain situations must be always correct *and* on time. This means that additional timing restrictions apply to the function of the system. For example the computer logic behind the electronic ABS subsystem of a car must be correct and at the same time able to provide prompt results according to the input parameters.

Testing a system thoroughly does not make it correct. A great deal of software errors are disguised during the testing sessions and become visible only after the system has been installed in the working environment. Most commercial solutions advertise the correctness of the respective programs using the Object Oriented Programming tag. Object oriented languages which constitute a major breakthrough in software development are based on the idea of objects. A program is essentially a set of objects and the control flow takes the form of message passing between the interconnected objects. Correctness of programs is based on the fact that a program which relies on well-tested smaller subcomponents is presumably free of errors itself. See figure 1.2.

This approach is inherently flawed since well tested components are not guaranteed to be correct in all the possible situations that a system can be deployed. Also as more components are added to a system, the increasing entropy will make management and maintenance more difficult, allowing for more errors on the part of the programmers who perform the integration. Finally, imperative (procedural and object oriented languages) have notoriously complicated syntax, making programs even more prone to software

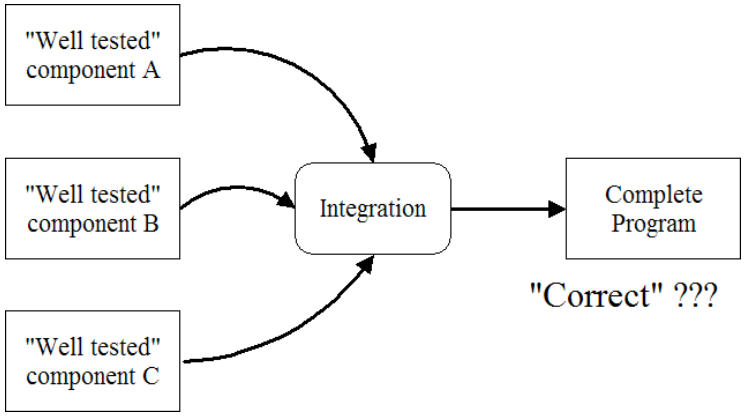


Figure 1.2: The Object Oriented Programming approach

errors by humans. The shortcomings of imperative languages and their effects on programming have been criticised already by researchers for some time now [BAC78]. Object oriented languages have achieved a lot, but computer scientists should always look for the next silver bullet[BRO87]. And perhaps the next silver bullet can be found in formal program verification and automatic programming.

Functional languages take a different approach to correctness. Functional programs are meant to be deterministic, producing always the same output for the same input without relying on the executing environment. Functionality is achieved through expression evaluation instead of side effects as is the usual case with imperative languages. Functional programming allows essentially the programmer to argue that his/her program is *mathematically* correct instead of relying on extensive testing. A common design pattern used in functional programming for example is mathematical induction. The functional program includes code only for the base case and the induction step and it will automatically work correctly for every possible case which can be proved by the induction hypothesis. Functional languages are inherently based on mathematical concepts, since they are a high level abstraction of the $\lambda - calculus$ introduced by Alonzo Church[CHU41]. Figure 1.3 outlines the idea.

It is possible however to extend this approach. Instead of constructing a program and *then* trying to use mathematical methods to prove its correctness, we could do the exact opposite. Extract the program itself from the mathematical proof in the first place! This is the foundation of the proofs-as-programs approach. Using this idea for software development results in programs which are correct by definition. Figure 1.4 shows the process.

Minlog is a system developed to support interactive proofs, and it is

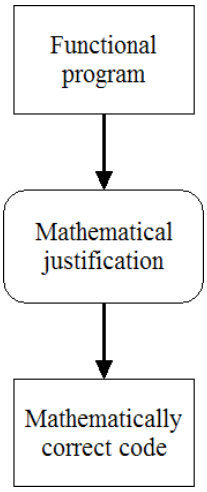


Figure 1.3: The functional approach

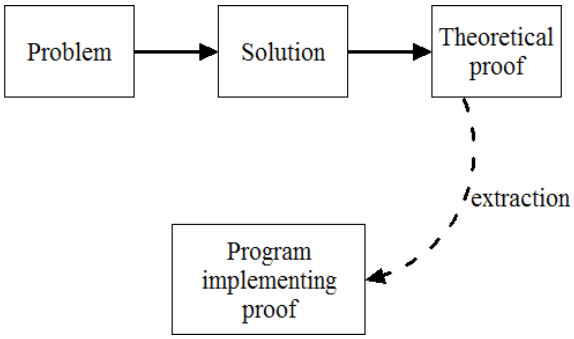


Figure 1.4: Proofs as programs

centered around the idea of extracting automatically programs (algorithms) from the resulting proofs. Currently the development and usage of Minlog is aimed at researchers and scientists mainly. It is for this reason that a TEX output module which outputs proof in a format easily embeddable in scientific documents is absolutely essential. The TEX typesetting system is the de-facto system for producing professional looking scientific papers and it is the natural choice for the target format of the Minlog output module which is described in this document (in its friendliest version of LATEX).

It is worth mentioning on a side note, that even in the commercial world some companies have realised that imperative languages are not always the perfect solution for a problem. The most promising initiative is the launch of the F# language[FSHARP] from Microsoft. F# is a functional language, but it is hosted in the .NET framework along with C++, C# and Visual Basic .NET all of which are traditional imperative languages.[2]

## 1.5 Project plan

The project involves three different technologies/areas which are blended together for the final program.

- The Minlog system.
- The Scheme programming language which hosts Minlog.
- The LATEX syntax rules.

To tackle the problem the following steps were taken:

1. Acquired a working knowledge of Scheme.
2. Acquired a working knowledge of the logical calculus that powers the Minlog system.
3. Studied the internal structure of the Minlog system.
4. Studied the original texoutput module for Minlog.
5. Developed a similar module from scratch for Minlog 4.
6. Integrated several ideas from the original texoutput module.

[2]Additionally, some of the "new features" of both virtual machines, like garbage collection and lack of pointers are native characteristics of functional languages...

7. Refined the output of the module to LaTeX instead of plain TeX.

8. Implemented several extensions to the module and restructured it for easy future maintenance.

9. Assessed other output formats used for document processing such as XML files, which will allow the immediate usage of all available XML processing tools.

## 1.6 Related work

Minlog is not the only proof assistant. Other well known similar systems include:

- Isabelle[ISA] at University of Cambridge.
- Coq[COQ] at I.N.R.I.A.
- Nu/PRL[NUPRL] at Cornell University.
- Agda[AGDA] at Chalmers University of Technology.

Since there is already a prototype texoutput module *for Minlog* the best place to start is by extending this code, instead of dealing with similar solutions for other systems. Additionally Minlog is rather small compared to other interactive proof checkers, which have more generic goals instead of concentrating on the proof-as-programs concept. The major difference however, is that the internal proof representation in Minlog is just a set of $\lambda - terms$ which can be "unrolled" and provide a complete history on the state of proof and the transformations that have been applied so far. This concept (commonly known as the Curry-Howard correspondence) is described in section 3.3.

# Chapter 2

# On Scheme

Scheme is one of the many dialects of Lisp. Lisp[MAEHL62] is a very old language developed around 1960 which had a great impact in Computer Science and became very popular for Artificial Intelligence applications among the research community. The name stands for "List processing" and it was the first language which acted as a high level abstraction of the $\lambda - calculus$. It was not purely functional however, since many imperative constructs were added as the language evolved.

Lisp spawned numerous dialects and of course affected the creation of other functional languages as well. What is so special about Scheme (apart from the fact that it has survived for the last 30 years) is the clean and elegant style of the language. Instead of defining an extensive wealth of core functions, Scheme follows a more minimalistic approach. It just defines a powerful and flexible yet small subset of required functionality, allowing the programmer to extend the language as needed for each individual case.

Table 2.1 taken from[SF89] deserves some attention. It lists some programming languages along with the pages of the respective defining standard.

Of course some of the programming languages included are now obsolete. It is worth comparing however the numbers for Scheme and the still dominant C language. Common Lisp is at the top of the table with an impressive number, implying very complex semantics. Naturally these numbers are not absolute and under no circumstances the size of the standard of a programming language shows the quality/expressiveness/performance of the language.

It shows however, that Scheme and Common Lisp have taken opposite directions with the former promoting clean syntax and the latter providing a rich programming tool set for the user. Even now (2004) the latest revised report for Scheme[RRS5] accounts to 51 pages.

| Language | Pages of Standard |
|---|---|
| Common LISP | 1000 or more |
| COBOL | 810 |
| ATLAS | 790 |
| Fortran 77 | 430 |
| PL/I | 420 |
| BASIC | 360 |
| ADA | 340 |
| Fortran 8x | 300 |
| C | 220 |
| Pascal | 120 |
| DIBOL | about 90 |
| Scheme | about 50 |

Table 2.1: Standards of various programming languages

## 2.1 History of Scheme

Scheme started out as an experiment around 1975. Guy Steele and Gerald Jay Sussman created a lexically scoped dialect of Lisp in order to study what would later be called as "tail-recursion". They considered their creation a "toy implementation"[SG93] and they called it *Schemer* in the tradition of other Artificial Intelligence languages with similar names (Planner, Conniver). The operating system that they were using (ITS) had a limit of 6 characters for file names and so the name was truncated to "Scheme".

With Scheme they discovered that it was possible to create calling functions which did not perform any processing on the results on the returned value and therefore the system did not need to keep resources for them[HAN90]. This allowed for recursive functions which occupied a fixed amount of memory in the process stack instead of filling it continuesly at every recursion step. With the results of this discovery[STE77] Steele argued that this technique permits unrestricted procedure calls which at the time were considered "expensive" compared to the imperative GOTO statement. The first report of Scheme was also published this period.

Steele became more involved with Common Lisp and the Scheme dialect was next implemented by Jonathan Rees, Norman Adams and others. They designed a Scheme dialect called "T"[AR82]. T added to the predicate functions the question mark (null?, number?) and tagged "destructive" functions with the exclamation mark (append!, set!), conventions which are present even today[SG93]. Also a compiler for T called "Orbit"[KRA86] was

developed.

Scheme became eventually an IEEE standard[SCH91]. Less informal Scheme definitions are outlined in the various "reports" with the latest one being the fifth[RRS5] published in 1998.

Various Scheme implementations have existed. Apart from those developed in major Universities (e.g. MIT) the most well-known commercial versions include PC-Scheme[BJ86] and Petite Scheme from Cadence Research Systems[CHEZ6] which is also the one that Minlog is based on.

## 2.2 The flexibility of Scheme

At this point we could start enumerating the advantages of the functional programming approach[HUG89], or even analysing some special Scheme characteristics (e.g. continuations). We prefer however, to focus more on the versatility of Scheme as a programming language and the fact that one can program easily in mixed imperative/functional style.

Having the ability to program in mixed style with Scheme means that:

- Seasoned imperative programmers can start using Scheme gradually moving from imperative to functional constructs.
- Experienced Scheme programmers can use different styles depending on the problem, having the best of both approaches at their disposal.

It is a common misconception that imperative and functional styles do not match[HUD89]. Scheme provides us essentially with the proof that this is just a myth which is commonly found among most programmers.

*Note: Unless otherwise stated all imperative code segments are in Java and all functional ones in Scheme.*

### 2.2.1 Procedural programming style

First we will briefly mention that Scheme supports the `do, case(switch), for-each` constructs which are similar to those found in imperative languages. For example a function which adds integers from zero to its argument would normally be written in imperative style:

```
public int sumvalues(int n)
{
        int i=0;
        int sum=0;
```

```
        while(i<=n)
        {
                sum=sum + i;
                i++;
        }
        return sum;
}

// sumvalues(4) => 10
```

The functional approach for the same function would normally involve recursion:

```
(define (sumvalues n)
  (if (= n 0) 0 (+ n (sumvalues (- n 1)))))

;(sumvalues 4) => 10
```

Apart from the smaller size of the code it is interesting to note that the addition is made backwards starting from n and reaching zero. However by using the `do` Scheme construct the same program could be transformed to

```
(define (sumvalues2 n)
  (do ((sum 0) (i 0 (+ i 1)))
    ((> i n) sum)
    (set! sum (+ sum i))))

;(sumvalues2 4) => 10
```

Again there is a `sum` variable which accumulates the additions and an `i` variable used as an iterator. The code also makes use of the "destructive" `set!` method which performs assignment.

The second imperative construct of Scheme that is interesting is the `begin` construct which allows for sequencing. The following code segment demonstrates its use:

```
(define (test)
  (define x 5)
  (define y 3)
  (begin (set! x 7)
         (set! y (+ y x))
         (display "The value of y is ")
```

```
          (display y)
          (newline)))

;(test) => "The value of y is 10"
```

This implements the respective imperative test function:

```
public void test()
{
        int x=5;
        int y=3;
        x=7;
        y=x+y;
        System.out.print("The value of y is ");
        System.out.println(y);
}

//test() => "The value of y is 10"
```

### 2.2.2 Object oriented programming style

The most basic approach which really just makes Scheme mimic object oriented functionality is to create multiple functions which act as mutators/accessors to a long list which holds all the private fields of an "object". For example an object representing a student could be written in Java as:

```
public class student
{
        private String name;
        private int number;

        public student(String name, int number)
        {
                this.name=name;
                this.number=number;
        }
        public String getName()
        {
                return name;
        }
        public int getNumber()
        {
```

```
                return number;
        }
}

//student me=new student("Kostis",302440); => me
//me.getName(); => "Kostis"
//me.getNumber(); => 302440
```

The mutators have been left out for brevity but their structure should be obvious. Since we have only two private fields a pair is enough for internal storage in the functional version:

```
(define (create-student name number)
  (cons name number))

(define (getname student)
  (car student))

(define (getnumber student)
  (cdr student))

;(define me (create-student "Kostis" 302440)) => me
;(getname me) => "Kostis"
;(getnumber me) => 302440
```

The mutators have not been included for brevity again. They could be easily implemented however with `set!`. In more complex cases the internal data structure can be a simple list, a circular list, a binary tree, or any other arbitrary structure that can be created by primitive Scheme functions.

The previous technique is really "pseudo object oriented". A more advanced approach would be to truly have the data along with the methods that operate on it in one single construct. The following example is taken from [AR88] without any modifications.

```
(define (make-simple-cell value)
  (lambda (selector)
    (cond ((eq? selector 'fetch)
           (lambda () value))
          ((eq? selector 'store!)
           (lambda (new-value)
             (set! value new-value)))
          ((eq? selector 'cell?)
```

```
           (lambda () #t))
          (else not-handled))))

;(define a-cell (make-simple-cell 13))
;((a-cell 'fetch)) => 13
;((a-cell 'store!) 21)
;((a-cell 'fetch)) => 21
;((a-cell 'cell?)) => true
;((a-cell 'foo)) => error
```

The `make-simple-cell` construct demonstrates the more advanced object oriented capabilities of Scheme. The same example in Java would be:

```
public class simpleCell
{
        private int value;
        public simpleCell(int value)
        {
                this.value=value;
        }
        public int fetch()
        {
                return value;
        }
        public void store(int value)
        {
                this.value=value;
        }
        public boolean isCell()
        {
                return true;
        }
}

//simpleCell acell=new simpleCell(13);
//acell.fetch(); => 13
//acell.store(21);
//acell.fetch(); => 21
//acell.isCell(); => true
```

We have barely scratched this topic (especially object oriented style) but it should be clear now that Scheme can accommodate multiple programming

styles and is not limited to only one of them.

## 2.3 Scheme today

In summary, Scheme is very versatile language. As with any programming language it is not a universal solution for all types of problems. It is still evolving however and certainly "it has served the community well"[RAM94]. Scheme has been used in the past for as diverse projects as compiler construction or even controlling robots[RD92]. Apart from the success Scheme enjoys within the research community, it is also used in "real world" projects. More recent (and more impressive) Scheme uses include the language of choice for defining mail filtering rules for the Evolution personal information management suite[XIM], and even the plug-in system for the most popular open-source image manipulation program[GIMP].

Although Minlog is at first glance a complicated system dealing with very high level logic, it is essentially just a set of Scheme source files demonstrating again the power and expressiveness of Scheme.

# Chapter 3

# Overview of the Minlog system

"What a calculator is to number Theory, Minlog is to proof theory". This is how Martin Ruckert described Minlog in the previous version of the Minlog tutorial (which is now superseded by[CRO4]).

At this point we will attempt to give a brief overview of the Minlog system trying to avoid implementation details and too formal definitions. At a very high level Minlog is an interactive prover focusing on the following goals[BBSSZ98]:

- The ability to construct proofs for given problems.
- The successful program extraction (algorithms) from the finished proofs.
- Partial or even full automation of the process. Minlog supports apart from interactive proofs, automatic search of (sub)proofs.

A more formal definition of the idea behind the proofs-as-programs concept using Minlog would be[BBSSZ98]:

1. Description of the required program $A$ and specification of the output for the given input ($A[input, output]$).
2. Construction of proof $M$ such that $\forall x \exists y A[x, y]$, meaning that *for every input* $x$ of the program there should be an appropriate output $y$ which *satisfies mathematically the* $A$ *specification.*
3. Extraction of algorithm $M$ which by definition implements $A[x, M(x)]$, meaning that it calculates the "correct" answer $M(x) = y$ for every input $x$.

Minlog is in active development since 1990. Initial programming was done by Helmut Schwichtenberg but as the popularity of the system increased more researchers started to contribute to the project. The growth of Minlog is demonstrated at table 3.1 which shows the relative size of the last versions available. (We have not included statistics for documentation and examples included in the Minlog distribution)

| Version | Lines of code | Source files | Directories |
|---|---|---|---|
| Minlog 3 | 16852 | 4 | 2 |
| Minlog 3i | 23161 | 13 | 2 |
| Minlog 4 | 42204 | 38 | 4 |

Table 3.1: Minlog evolution

## 3.1 Introduction to Minlog Logic

At the most basic level Minlog implements minimal logic. The formulas that Minlog can accept as input can include the well known $\wedge, \rightarrow, \forall, \exists^*$. Table 3.2 has the details. The third column shows the "external representation" of each symbol meaning how each symbol should be typed so that it can be successfully parsed by Minlog's `parse-formula` function or how it is printed by Minlog itself to the user. For the full formal syntax the reader is referenced to [BBSSZ98].

| Symbol | Meaning | Minlog form |
|---|---|---|
| $\top$ | true | `T` |
| $\bot$ | false | `bot` |
| $\wedge$ | Conjunction | `&` |
| $\rightarrow$ | Implication | `->` |
| $\forall$ | For all | `all` |
| $\exists^*$ | Exists | `ex` |
| $\neg A$ | Negation | `A -> bot` |

Table 3.2: Minimal logic for Minlog

For each of the logic connectives, Minlog holds an *introduction* rule which can be used to insert the respective symbol in the current stage of the proof, and an *elimination* rule which can be used under the appropriate circumstances in order to remove the symbol (and hopefully simplify the expression).[BU01]

$$\text{Conjunction Introduction} \quad \frac{P \qquad Q}{P \wedge Q}\wedge^{+} \tag{3.1}$$

$$\text{Conjunction Elimination} \quad \frac{P \wedge Q}{P}\wedge_{l}^{-} \qquad \frac{P \wedge Q}{Q}\wedge_{r}^{-} \tag{3.2}$$

There are two symmetrical conjunction elimination rules depending on whether we want to *keep* the left or right part of the connective.

$$\text{Implication Introduction} \quad \frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \to Q} \to^{+} \tag{3.3}$$

the bracketed assumption $P$ is discarded.

$$\text{Implication Elimination} \quad \frac{P \to Q \qquad P}{Q} \to^{-} \tag{3.4}$$

$$\text{For all Introduction} \quad \frac{P(x)}{\forall x P(x)}\forall^{+} \tag{3.5}$$

provided that $x$ is not free in any assumption on which the proof of $P(x)$ depends. This rule essentially dictates that if know that $P(x)$ is true for some $x$ and we haven't chosen any particular $x$ (it is completely arbitrary) then we can assume that $P(x)$ holds for all similar to $x$.

$$\text{For all Elimination} \quad \frac{\forall x P(x)}{P(t)}\forall^{-} \tag{3.6}$$

If we know that for all $x$ $P(x)$ is true, we can select one of them (named $t$) and we know that $P(t)$ is also true.

$$\text{Exists Introduction} \quad \frac{P(t)}{\exists^{*} x P(x)}\exists_{+}^{*} \tag{3.7}$$

Since we already have a $t$ which satisfies $P$ we can assume that there exists one $x$ such that $P(x)$ holds, and this $x$ is $t$ which is already known to have this attribute.

$$\text{Exists Elimination} \quad \frac{\exists^{*} x P(x) \qquad \forall x(P(x) \to Q)}{Q}\exists_{-}^{*} \tag{3.8}$$

provided $x$ is not free in $Q$. In both cases the symbol refers to the "strong" existential quantifier and not the "weak" (classical) existential quantifier.

The latter is just another way of expressing $\neg\forall x \neg P(x)$ while the former truly notifies that one can extract a $t$ term which satisfies $P(x)$.

Of course this is not an exhaustive list of Minlog's logic capabilities. For example Minlog supports some predefined rules/proofs such as the stability rule:

$$\text{Reductio-ad-absurdum:} \quad \neg\neg A \to A \tag{3.9}$$

and the negation axiom

$$\text{Ex-Falso-Quodlibet:} \quad \bot \to A \tag{3.10}$$

which dictates that we can conclude any $A$ from false. Most recent versions of Minlog have more advanced features. A new addition for example involves the ability to use mathematical induction with arbitrary datatypes (other than natural numbers).

## 3.2 A sample Minlog session

What follows is a sample interactive Minlog session which proves a trivial formula involving conjunction. The example is taken from[CRO4].We want to prove:

$$A \wedge B \to B \wedge A$$

Every line which starts with `>` is typed from the user. All the other lines are Minlog messages.

```
>(add-predconst-name "A" "B" (make-arity))
; ok, predicate constant A: (arity) added
; ok, predicate constant B: (arity) added
```

We have just defined the A and B as null-ary predicate constants, that is propositional constants. It could also be Q and R, p1 and p2 or anything else.

```
>(set-goal (pf "(A & B) -> (B & A)"))
; ?_1: A & B -> B & A
```

Here we defined what we want to prove. The `pf` function stands for "parse-formula". Minlog has printed the formula and has assigned also number 1 to it.

```
>(assume 1)
; ok, we now have the new goal
; ?_2: B & A from
;   1:A & B
```

Now we command Minlog to break the implication into the left part (assumption) and assign it the number 1. The right part of the implication now is expression number 2 and must be derived from the left part (number 1). This command is actually the implication introduction rule applied backwards.

```
>(split)
; ok, we have the new goals
; ?_4: A from
;   1:A & B

; ?_3: B from
;   1:A & B
```

Split is a special Minlog command which expects the current goal to have conjunction (as we have now). It splits the conjunction into two separate parts. It is essentially the conjunction introduction rule applied backwards. Therefore the old expression with number 2 was split now into 4 and 3 which must be derived from the unchanged 1.

```
>(use 1)
; ok, ?_3 is proved. The active goal now is
; ?_4: A from
;   1: A & B
>(use 1)
; ok, ?_4 is proved. Proof finished.
```

Since the example is very trivial the proof is very short. Both expressions (number 3 and 4) can be derived from 1 since they are just the left and right parts of the conjunction. Typing the `use` command along with the number of the expression that we wish to use accomplishes the required effect and the proof is finished.

### 3.2.1 Automatic proof search

Since the example we have selected is very basic we can also demonstrate easily the automatic proof search feature

```
>(set-goal (pf "(A & B) -> (B & A)"))
; ?_1: A & B -> B & A
>(search)

; ok, ?_1 is proved by minimal quantifier logic. Proof finished.
>(display-proof)
; ...A & B by assumption u17
; ..B by and elim right
; ...A & B by assumption u17
; ..A by and elim left
; .B & A by and intro
; A & B -> B & A by imp intro u17
```

After defining the same goal again we instruct Minlog to attempt an automatic proof search. The search is successful. With the final command we query Minlog for the final proof. It prints out (properly indented by dots) the stages of the proof. For each step it mentions what rule it has used and whether it was an elimination (elim) or an introduction (intro). The u17 string is just an automatic internal name that Minlog has assigned to the assumption.

## 3.3 Storage of proofs

The introduction of $\lambda$-calculus had an impact on several aspects of science. There is a great amount of literature revolving around $\lambda$-calculus. A lot of scientific papers have emerged examining existing scientific fields and their connection to the $\lambda$-calculus.

The previous chapter showed the effect on programming languages. We already mentioned that a functional programming language can be thought as a high level abstraction of $\lambda$-calculus. Another interesting aspect of the topic, is $\lambda$-calculus and logic theory. We will also explain how this concept applies to Minlog.

It turns out that there is an analogy between natural deduction proofs and the $\lambda$-calculus concept. This analogy is known as the Curry-Howard correspondence. Figure 3.3 shows the principle behind this theory. More details can be found in [SCHW99].

The key point in our case, is that Minlog actually uses the Curry-Howard correspondence to store internally the proof process. A simple implementation would involve a big table of the individual transformations and any associated formulas (before and after). Minlog stores instead each proof

| Derivation/Transformation | $\lambda$-calculus |
|---|---|
| Assumption | $\lambda$-term |
| Conjunction introduction | $\lambda$-pairing |
| Conjunction elimination (left) | $\lambda$-projection |
| Conjunction elimination (right) | $\lambda$-projection |
| For all introduction | $\lambda$-abstraction |
| For all elimination | $\lambda$-application |
| Implication introduction | $\lambda$-abstraction |
| Implication elimination | $\lambda$-application |

Table 3.3: Natural deduction and $\lambda$-calculus

(partial or finished) as a single $\lambda$-term which is then used for all Minlog operations.

To print out the $\lambda$-term of the proof, we can use the `display-proof-expr` function. *Note: The cons, cdr, car keywords denote $\lambda$-pairing and the two $\lambda$-projections. This is a convention followed by Minlog. These are not Scheme functions!.* For the last example this would give us:

```
> (display-proof-expr)
(lambda (u17) (cons (cdr u17) (car u17)))
```

We can easily see how this term can be unrolled in order to get the history of the proof. Starting from the left we find a $\lambda$-abstraction. This was the `(assume 1)` Minlog command and u17 is the assumption that was created $(A \to B)$. Next we have a $\lambda$-pairing (the `cons` keyword) so we know that this term is formed by two other terms. We informed Minlog about this when we used the `(split)` command. Finally we have the two terms which are connected. These are the left and right projections of term u17 (which is the assumption introduced in the first step). These map to $A$ and $B$ as extracted from the assumption. The respective Minlog commands are the last two `(use 1)` steps.

This representation of proofs will become very important during the module implementation phase. The details will be explored in the next chapters.

# Part II

# Specification

# Chapter 4

# Project Objectives

The role of the texoutput module is very simple in principle. Since Minlog is written in Scheme which is a LISP dialect (= LISt Processing), all major Minlog structures are actually long nested lists. The same is true for every possible aspect of the Minlog system. Formulas for example are also represented as lists.

So the objective of our module is the processing of the central Scheme list, which represents the proof process, into a LaTeX text file with English sentences. This resembles a bit the function of a special compiler with input the mathematical representation of a proof and output the respective English expression. In this case however, we are not developing a separate software program which would parse the input individually. Instead we access the input structure via the published Minlog functions, and the code itself is hosted in the Minlog environment rather than running stand-alone. See figure 4.1.

This central Minlog list is calculated by the `(current-proof)` function and as the name suggests it contains all possible information of a finished (or even partial) proof. To successfully form English sentences from this structure we need to process the list and remove unwanted parts or modify the existing ones to fit our needs.

There are several architectural questions that need to be answered such as the order of the processing actions or even whether only one or more parsing phases are needed. But without even starting the implementation we need to focus on the topic of extensibility.

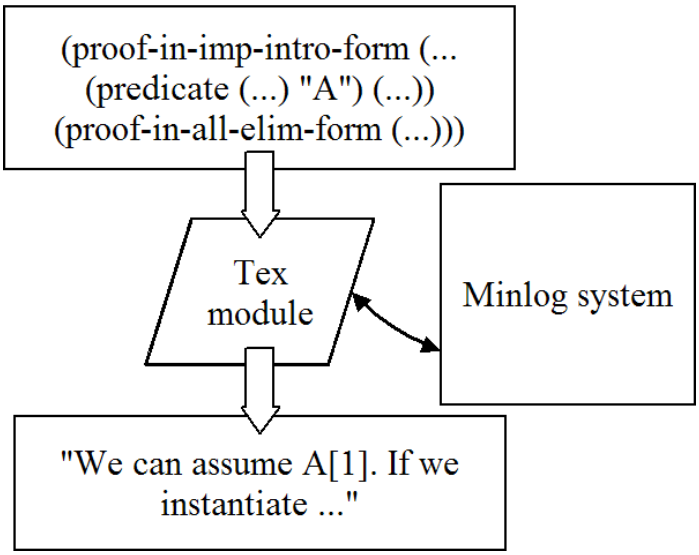


Figure 4.1: Expressing the proof structure

## 4.1 Abstraction and integration

The original texoutput module for Minlog3/3i was an impressive piece of software. It supported (among other features):

- Translation of minimal logic proofs ($\rightarrow, \wedge, \forall$ e.t.c) .
- Advanced constructs (global assumptions, axioms e.t.c).
- Induction proofs.
- Deeply nested subproofs (using indented bullet points in the output file).
- Presentation options with numerous "flags" allowing the user multiple levels of verbosity.
- Facilities for the presentation of user-defined Minlog constructs.

To achieve all this functionality the code was not an "add-on" in the strict sense. The old Minlog module was actually plugged-in so deep into the core Minlog system that it depended heavily on the Minlog internals. In this sense the module was *part* of Minlog rather that a simple extension.

Unfortunately this meant that even a slight change in the Minlog internals could break the texoutput module and make it non-functional. This was the

case with Minlog 4. This recent version was a major rewrite of the core code with new additions and features. The texoutput module became obsolete in a moment. An upgrade to the module was not simple enough. In fact we could argue that upgrading the old texoutput module to be compatible with Minlog 4 was not an easier task than creating a new module from scratch.

Therefore, it is clear that compatibility is our primary objective. We need the new module to be easy to maintain, extend and upgrade so that in the future it can follow new Minlog versions with little or (in the ideal case) no modifications. We are even prepared to sacrifice some functionality to achieve this. As we try to avoid deep integration with the main Minlog system however, we realise that we do not have direct access to Minlog internals and in effect, the capabilities are getting more limited. This is the price we have to pay for compatibility.

Of course this problem is not new in software engineering. In the presence of an existing large system (in our case, Minlog) the programmer who wishes to attach new functionality must decide on the level or integration between the new module and the system. Abstracting the low level details and using existing interfaces results in a stable implementation. Integrating deep into the core system allows for speed and direct manipulation of the system internals which results in great functionality.

There is no "correct" or "wrong" approach. Depending on the situation the programmer must choose the one that seems fit for the given project.

## 4.2 Following the abstract data types paradigm

Data abstraction is one of the concepts that allows us to build stable software. The central idea is that the data manipulation routines are hidden from the outside world. Only communication interfaces are exposed for data input and data output, but the processing functions are never directly accessible. This means that they can be improved or completely changed at any time and nothing will break as long as the communication interfaces stay the same. Figure 4.2 shows the idea.

This idea can be extended by defining that a software module is just a set of such abstract data types. Object oriented programming is also based on this concept. The software code is never just a spaghetti of interconnected functions which have universal access everywhere. Instead, we have only individual objects with exposed interfaces. Figure 4.3 resembles a full software system. The programming rules are simple:

1. Data and control flow is resolved into inter-object communication via the exposed interfaces.

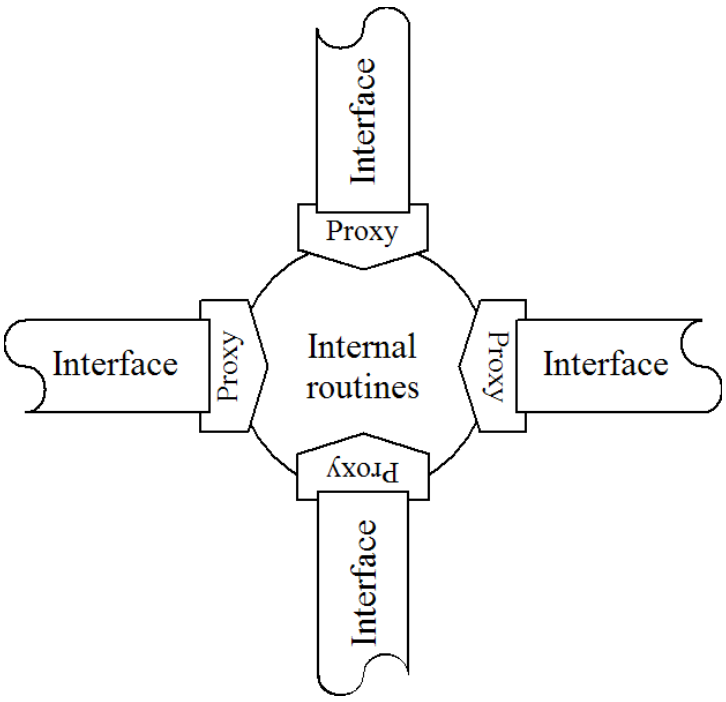


Figure 4.2: The data abstraction principle

2. An object has no knowledge of the internal implementation of any other object.

Object oriented programming goes even further with additional concepts like encapsulation, inheritance, access control e.t.c in order to achieve maximum flexibility. A full explanation of all these is outside the scope of this document. More important is the fact that each time we need to upgrade certain functionality we change only the internal implementation for one or more objects and the whole system remains unaffected. It is also possible to define additional communication interfaces which co-exist with the existing ones. Breaking the communication interfaces themselves is more difficult but certainly easier then the spaghetti code approach. The programmer just needs to track the objects that make use (have bindings) of the interfaces in question and make appropriate changes. Objects that never used them anyway, can work now as before. For a formal description of abstract data types the reader is referenced to [EA00].

Abstract data types are in no way perfect. They have their disadvantages too. The most obvious one is the "plumbing" code that we need to add in order to form all the infrastructure of the communication channels. We need to build all the proxies which will forward the outside function calls to the

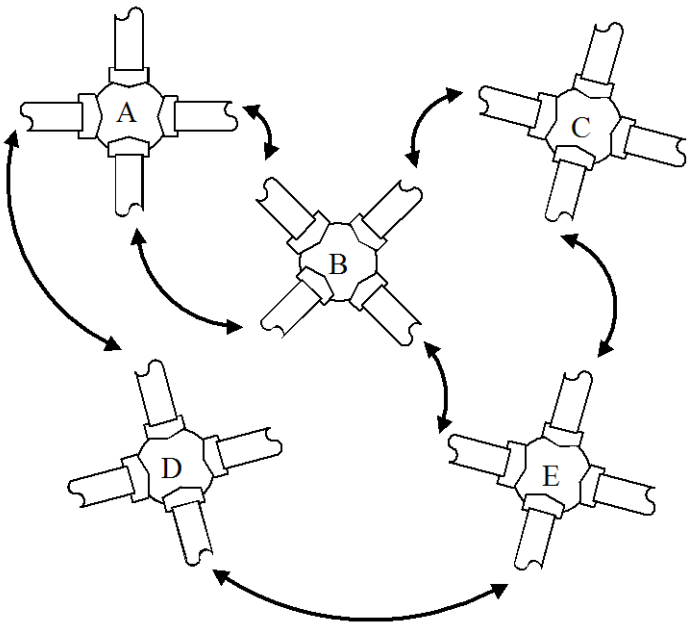


Figure 4.3: A set of abstract data types

internal structure of an object (in both directions). We must agree on the communication channels themselves because many objects might be built by external developers (third parties). We must decide on marshaling and unmarshaling of parameters as they are packaged and transfered through the system. Finally there is a small impact on performance since communication overhead is added for each function call, now that data must travel via proxies from source to destination.

Software engineering is always full of decisions regarding the design and implementation of the system. The reasons behind our choices as we progress through development should be clear now, as we have already mentioned that we want easy maintenance and extensibility.

## 4.3 Building a Minlog module

If we focus on the Minlog system we can easily visualise the ideal case. We draw an imaginary line (see Figure 4.4) which splits the main system from the module. We will only write code that belongs to the module but we will never touch the internals of the Minlog system itself. To follow the abstract data types approach we need to select a few but important Minlog functions which will be used as communication channels. These channels will be the only ones

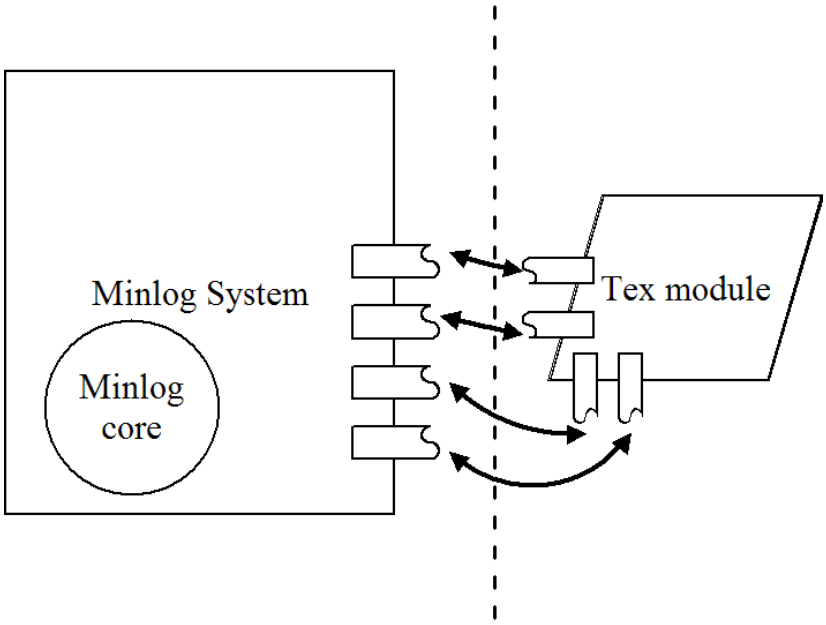


Figure 4.4: The module approach

which will transfer data between the module and the Minlog system, and that is why they are crossing the boundaries (the imaginary vertical line).

The imaginary line essentially defines that neither the module nor the Minlog system know or care about the internal implementation of each other. They only communicate via the well-known interfaces. The Minlog system can change at any time several parts of its code, and in a similar manner the module may be upgraded or extended in the future. Nothing will break however, since the interfaces will stay the same.

This design spawns several architectural questions that need to be answered. These questions have nothing to do with the module implementation (by definition) but rather with which parts will be considered public (meaning interfaces that are expected to stay in place) or private (meaning functions which might change or perish without notice).

The first question is the position of the imaginary line. We can position this line in the virtual space more on the right (making a tight module and bloating the Minlog system with interfaces) or more on the left (making the Minlog system minimal, and implementing most functionality in the module). The line does not have to be vertical too. It can make corners/curves depending on the level of integration we want. Maybe there are parts where the core Minlog system has strong foundations where we can build on, and other parts where we feel that we need to include a great deal of functionality

in the module code.

The second question has to do with the selection of the interfaces which will serve as "bridges" connecting the two sides. There can be numerous ways to cross the border, but finding the optimal solution can be difficult. We need to choose those functions that will help us build the module using current Minlog code without having to reinvent the wheel each time (i.e. solving problems which are already solved by the Minlog system itself). Having too abstract interfaces means that the module needs a lot of plumbing code. Having too specific interfaces means that the module knows a great deal about Minlog internals which is what we want to avoid in the first place.

On a side note we need to consider also the number of the interfaces. This means that apart from *which* functions are allowed to cross the border, we care about *how many* of them exist. Again a balance must be found for the optimal results. Too few interfaces will make the module very limited. Too many interfaces and we will not have a "module" anymore, but rather a new Minlog component.

This was the theoretical background regarding the software architecture of the project. The next chapter examines how all these decisions affect the actual programming code of the module.

# Chapter 5

# Analysis and Design

As already mentioned, Minlog stores all the details for a proof in a huge list called `(current-proof)`. Actually this is the name of the Minlog function that returns the list as its result, but this minor point does not affect our design decisions.

## 5.1 The initial approach

The naive (and most obvious way) to transform the contents of this list to a LaTeX file would be the creation of a single function (called `transform-proof` of something similar) which takes as argument the input proof list and creates as output a big LaTeX string that can be dumped to a file. Figure 5.1 outlines this approach.

This solution seems logical and also fast. We have only one "processing pipe" which takes from one end the raw `(current-proof)` list, then processes it and from the other end we have a big LaTeX string.

Unfortunately this is not the optimal in our case. Our goals (as defined from the previous chapter) are:

- Extensibility
- Reliability
- Easy maintenance
- Easy upgrade/improvement

These goals may seem similar but the obvious solution does not accomplish them.

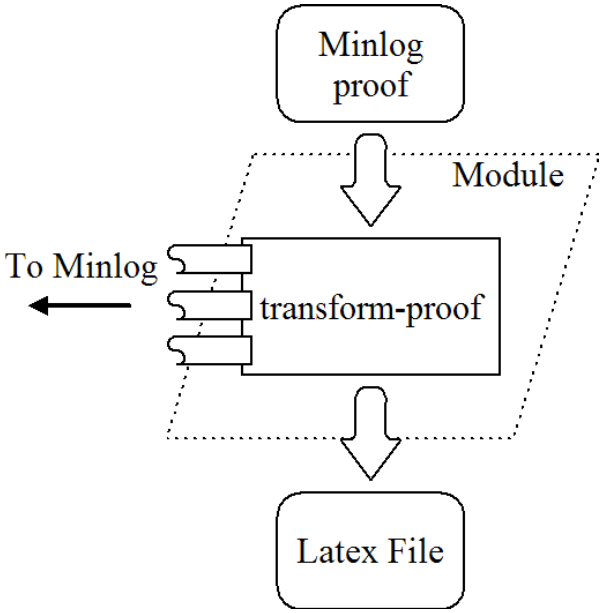


Figure 5.1: The obvious solution

The reason for this, is that a careful analysis of the situation reveals that there are actually two distinct kinds of code that have to be implemented. The first kind is Minlog-dependent since it uses Minlog functions to extract from `(current-proof)` the needed information. The second kind is Minlog-independent since it just creates LaTeX sentences and mathematical formulas.

A monolithic solution like the one mentioned, mixes these elements together. A programmer who is called to extend the code cannot easily identify the parts of the module which belong to the first or the second kind. If for example the Minlog system changes in the future, the programmer should have an easy way to focus only on the Minlog dependent parts. In a similar manner if someone wants to add functionality to the LaTeX output, he/she should care only about the LaTeX code segments.

## 5.2 The two-step transformation process

It is therefore natural to split the transformation process into two separate phases. Each phase contains the respective code functionality. Now the different role of each part is evident.

**Stage 1** In this stage the `(current-proof)` list is parsed. Several Minlog functions are used to extract only the needed information. Formulas

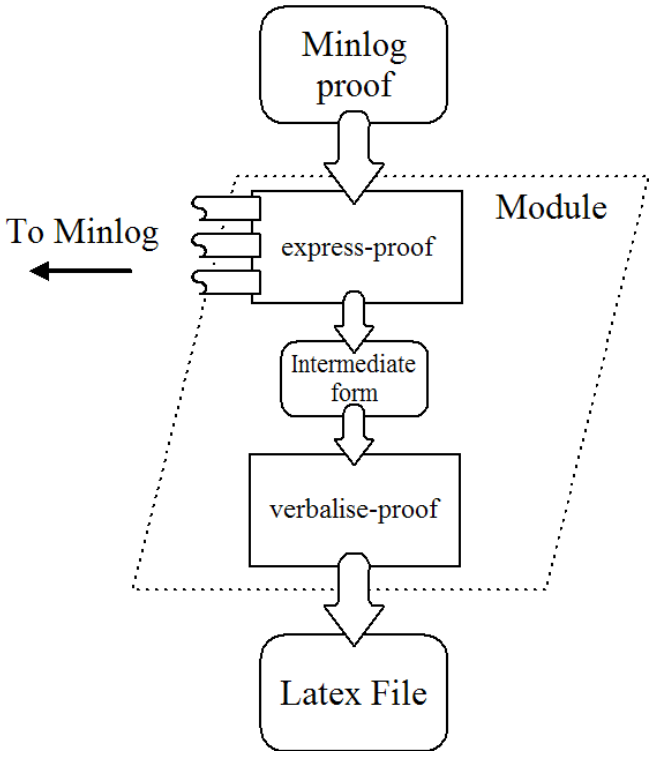


Figure 5.2: Two-step transformation process.

are also formatted. The result of stage one is an intermediate proof form independent of Minlog. All communication between Minlog and the texoutput module happens at this stage.

**Stage 2** In this stage the module transforms the intermediate result from stage one to a LaTeX file. No communication takes parts with the Minlog system. The code is completely self-contained with no external dependencies of any kind.

Figure 5.2 shows this approach. The function that performs the first step is called `express-proof`. The function responsible for step two is called `verbalise-proof`. The full LaTeX string can now be obtained (in Scheme terms) by

```
(verbalise-proof (express-proof (current-proof)))
```

The motives behind the intermediate form are well known. The same approach is found in LaTeX itself where a document is first processed into a device independent form (`.dvi` files) which is then transformed to its final

state (HTML, ps and pdf files). Implementing another output target for LaTeX means implementing only the output routines of the dvi file but not the source text file (.tex) which have already been implemented.

Compiler technology is another area of intermediate forms. A compiler is split into the front-end which is responsible for parsing the input source files and creating pseudo assembly code, and the back-end. The latter is responsible for converting the pseudo assembly code to the real machine code of a specific architecture. This means that front-end and back-end can be developed separately. Improving a compiler to take advantage of the latest in processor technology means that improvements will be only made to the back-end. In a similar manner a compiler for a completely new language can reuse the well tested back-end of an existing compiler, while the front-end will deal only the new language. And of course it is possible to *port* the compiler to a completely different processor architecture, since the amount of work that has to be done is halved. Only the back-end needs to be changed[1].

Finally the same idea is the basis of the virtual machines used today. The Java compiler from Sun Microsystems has one front-end (for the Java language syntax) and multiple back-ends for the various computer platforms (Solaris, Win32, Linux). The .NET framework from Microsoft has instead multiple front-ends (for C#, Visual Basic e.t.c) and one back-end (for the Windows platform). In both cases the compiled code first passes from a platform/language independent "byte-code" code which is then executed by the respective virtual machine environment.

In our case this two stage process means that if in the future the tex-output module is to be enriched and used for another output formal other than LaTeX, it can easily be done without starting from scratch having to implement again the Minlog specific functions. For example an XML output routine could be developed which has as input the custom proof created by the `(express-proof)` function, rather than the raw `(current-proof)` list.

## 5.3 Interface definition

One big difference between the first and second stage is that the latter is self-contained while only the former makes use of Minlog functions. We could just mark in the code (with comments) which segments belong to which stage, but this does not show which are the Minlog functions used.

The Minlog functions that we use in our code serve as the interfaces which we have focused on all the previous paragraphs. These functions are clearly

[1]This seems to be one important factor of the success of the popular GCC (GNU compiler collection)

included at the start of the source code file where their aliases are introduced. Here is a small sample:

```
; Defined in src/pproof.scm, line:201
(define input-proof num-goals-and-proof-to-proof)

; Defined in src/proof.scm, line:15
(define get-formula-from-proof proof-to-formula)

; Defined in src/proof.scm, line:169
(define is-proof-in-imp-intro-form? proof-in-imp-intro-form?)
```

At first glance these function redefinitions may seem redundant. They are however crucial to our purpose. They serve two goals at the same time.

First they show which functions of the Minlog system are used in the module. They are actually the interfaces that the module will use to communicate with the Minlog system. All data flow from/to Minlog passes from these functions. The programmer has all this information gathered in one place. No need to read the actual implementation code to track down Minlog functions. They are known in advance from this section. It is also easy to check if a Minlog upgrade will affect the module or not. Changes happening to Minlog in other functions apart from those included in this section do not concern the module at all. If on the other hand, a function defined here changes, it is easy to isolate it and see how the module should be changed too.

The different name for these functions helps also code maintenance in the hypothetical case where a Minlog function changes its name. In this case a simple renaming of the function can happen here and all the module code will continue to function since it uses the alias. For example the Minlog function `num-goals-and-proof-to-proof` is currently used in the module as `input-proof`. It is possible however that the name of this function will become obsolete, and in its place only the `current-proof` function will be used (which exists already in the Minlog system). In this case a simple renaming in the respective line will point the `input-proof` name to the `current-proof` function. The module will continue to work since all the module code contains references to `input-proof` which do not depend on the Minlog name but just need to point to the correct function.

For clarity we have also included in the comments the exact Minlog file name and line number where each function resides.

## 5.4 Design enhancements

We have already discussed how the abstract data types approach affects the module as a whole. We have shown that this will make the separation from the Minlog system more evident and at the same time help future maintenance of the code.

This does not stop us however, to think in the same way for the texoutput module itself. It would not make sense after all not to follow this principle in the core code of the module. Data abstraction can exist in small or large scale projects. In large projects it is almost imperative but it can also be applied in small projects too. Therefore we will attempt again to hide the internal layout of major data structures and provide access, only via well guarded interface functions.

In programming terms we have used the techniques introduced in the chapter about the Scheme programming language. We have specifically chosen the first (simpler) way to create Scheme "objects" for various structures inside the code. For each such object, there is a constructor function and several accessor functions. Even functions which process the object itself do not have direct access on its internals. They must go through the interfaces like everything else in the module. Table 5.1 shows the format of this.

| Function | Format | Example |
|---|---|---|
| Creating | (make-object component-list) | (make-all-elim-node formula ...) |
| Accessing | (get-component-from-object object) | (get-arg-from-all-elim-node node) |
| Processing | (action-object object) | (verbalise-all-elim node) |

Table 5.1: Object programming in the module code

A similar syntax is used all over the Minlog system code so we decided to adopt it too for the module code.

A large part of the texoutput module code revolves around the intermediate proof form which is created by `(express-proof)` and processed by `(verbalise-proof)`. The structure itself is a long Scheme list which contains smaller lists of its own with subproof. If we map this list into 2-dimensional space it is essentially a tree structure. The leaves are small subproofs (e.g. assumptions) while the nodes are proof transformations (this is further explained in the next chapter). It is therefore natural to consider the tree nodes as individual objects which can be manipulated independently.

For example the all elimination and conjunction introduction are (like most other data structures) simple lists. This is abstracted by presenting them as objects.

```
;All elimination methods
(define (make-all-elim-node formula operator argument) ...)

(define (get-result-from-all-elim-node node) ...)

(define (get-op-from-all-elim-node node) ...)

(define (get-arg-from-all-elim-node node) ...)

(define (verbalise-all-elim all-elim-node) ...)

;And Introduction methods
(define (make-and-intro-node formula left-part right-part) ...)

(define (get-left-from-and-intro-node node) ...)

(define (get-right-from-and-intro-node node) ...)

(define (verbalise-and-intro and-intro-node) ...)
```

In reality the accessor functions are really simple. Most times they just select an element from the internal list (the constructor itself is a `(list)` function). For example in order to get the operator from an all elimination node the code just selects the third element from the internal list.

```
(define (get-op-from-all-elim-node node)
  (list-ref node 2))
```

But this is the hidden implementation. If in the future the list components are rearranged then nothing will break because a single change in the accessor function will correct all the code that uses it.

This also brings us to a similar abstraction topic. Most Lisp derived languages revolve around lists and their manipulation. The primitive functions are `car` and `cdr` (and variants) which can be used to select the head or the rest of a list. Code however which uses them is "ugly". It reveals the internals of the data structures. We go a step further as to eliminate them too.

We do this in a specific way for each structure. As an example we assume we have a data structure which contains two objects called `intro-list` and `kernel` respectively. The structure is formed just by "listing" them together. We build additional code which hides this fact and makes the code implementation independent.

```
(define (get-custom-intro-list intro-list-and-kernel)
  (car intro-list-and-kernel))

(define (get-custom-kernel intro-list-and-kernel)
  (cadr intro-list-and-kernel))
```

It is hard to read code filled with primitive list functions. Having named functions (as delegates) not only serves data abstraction but also gives some meta-information about the data recovered (this first function obtains the intro-list while the second return the kernel object).

This concludes our design and format decisions for the code.

# Part III

# Development

# Chapter 6

# Implementation Process

Now that all the theoretical foundations are in place, we can dive into the implementation details of the development process and attempt to explain all our decisions during each coding phase.

## 6.1 Representation of formulas

Proofs are essentially a combination of formulas. Every Minlog session starts from a given formula which needs to be proved. It is the starting point of the whole mathematical process. Then by using Minlog functions and facilities several transformations are applied on each step until the goal is proved (the original formula). It is therefore imperative to develop a LaTeX presentation of formulas. In fact the whole development starts from formulas. A minimal tex module which just prints the given formula but nothing else is our first milestone.

Internally, a formula is just a long Scheme list holding the symbols, variables, predicates and all its other parts. This list can be thought of as a tree. Nodes of the tree are connectives, while leaves are either "atomic" parts or formulas themselves. See figure 6.1.

Because we will use Minlog accessor functions to process the list structure rather than parsing it directly, the resulting module function will clearly belong to stage 1.

At this point normally we would have to scan the Minlog sources and collect all the formula related functions. All code regarding formulas in Minlog is included in the source file *formula.scm* in the Minlog source directory. In this case however, things were more easy than we thought.

Minlog includes already for its own purposes a formula-formatting function. The function is named `formula-to-string` (which resides in for-

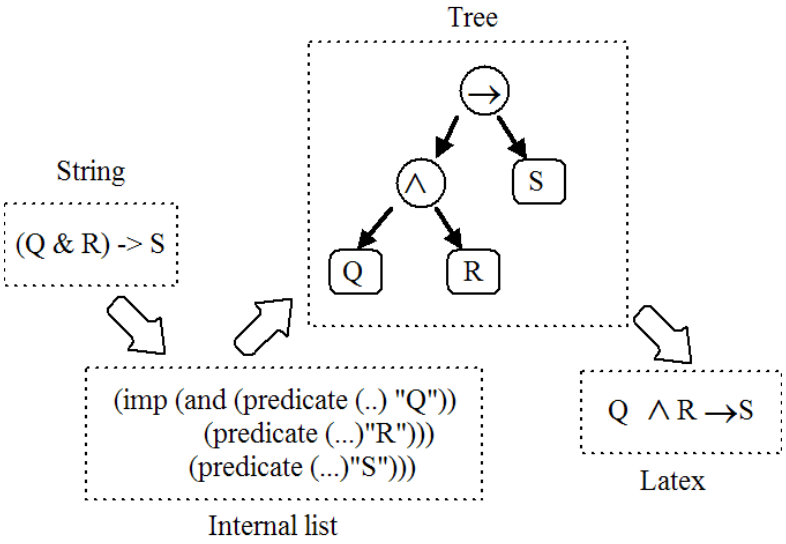


Figure 6.1: Processing and presenting Minlog formulas

mula.scm too). Minlog uses it to express formulas during the interactive proof sessions. It can be thought as the opposite of `parse-formula` (or pf for short) which was demonstrated in the section about Minlog. In effect the `parse-formula` function reads a normal string and creates a Minlog formula structure while the `formula-to-string` converts it back. Figure 6.2 shows this concept.

It only seems logical to adopt the `formula-to-string` Minlog function and modify it for our needs. The changes are limited. All the "string representation of symbols" (like ex,all) need to be changed to the LaTeX counterparts (`\exists, \forall`).

The modified function is included in the module as `convert-formula-to-latex`. It is a recursive function (as the original). Basically it takes a formula structure, formats and prints the connective symbol and then calls itself on the sub-formulas of it. We can illustrate this with an example. This is the sample code for an implication formula.

```
[...in a cond block of convert-formula-to-latex...]
((is-imp-formula? formula)
   (let* ((prem (get-premise-from-imp-formula formula))
          (concl (get-conclusion-from-imp-formula formula))
          (string1 (convert-formula-to-latex prem))
          (string2 (convert-formula-to-latex concl)))
          (string-append
     (if (or (is-quant-prime-formula? prem) (is-and-formula? prem))
```

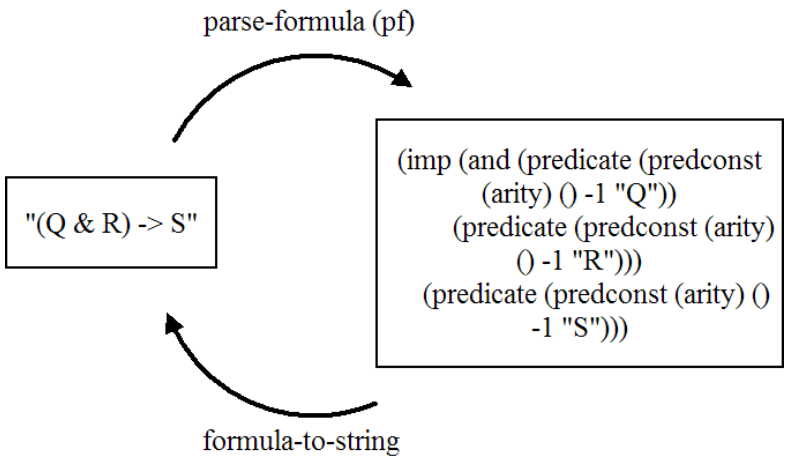


Figure 6.2: Converting a formula to/from a string representation

```
        string1
        (string-append "(" string1 ")"))
      " \\to " string2)))
[... rest of cond block...]
```

The code just inserts the `\\to` string (the LaTeX token for implication) between the components of the formula, which themselves are then formed by `convert-formula-to-latex` recursively (the enclosing function). We use two backslashes for all LaTeX elements to show Petite Scheme that we mean a literal backslash.

We should also note that all functions used to analyse formulas (e.g. `get-premise-from-imp-formula`) are not native Minlog functions, but rather aliases as described in the previous chapters. So this is where the interfaces are used. Thus the `convert-formula-to-latex` function belongs to stage 1.

The current implemented formula-formatting function supports the types of formulas shown in table 6.1 where "nc" means non-computational.

| Description | Minlog symbol | LaTeX token | Example |
|---|---|---|---|
| Implication | imp | `\to` | $A \to B$ |
| Conjunction | and | `\land` | $A \wedge B$ |
| For all | all | `\forall` | $\forall x A(x)$ |
| Exists | ex | `\exists^{*}` | $\exists^{*} x B(x)$ |
| For all(nc) | allnc | `\forall^{nc}` | $\forall^{nc} y D(y)$ |

Table 6.1: Formulas supported (connectives and elements)

The function also includes code for atomic elements and predicate variables which are the lowest level formula components (the leaves of the tree).

With the formula representation code in place, we can now tackle the main problem. The ultimate goal of the module is to express in an efficient manner the Minlog proofs.

## 6.2 Proof verbalisation (top-down)

Proof verbalisation is a huge topic on its own. It exists even out of the context of Minlog. Logic proofs consist of many small individual steps which lead to the final result. Apart from the important steps there also some details which most times are left out (they are obvious). Also the style of the representation can vary. There are even subjective reasons where a proof can be expressed in one way or another, among logic researchers themselves. It is generally hard to predict every possible case and try to provide a representation that satisfies everyone.

As an experiment we can use the proof representation support which is built-in already into the Minlog system. Minlog includes the `(diplay-proof)` function which prints out the individuals steps involved in the proof. It is completely mechanical and there is no attempt to beautify or clarify the result. It was briefly shown in section 3.

For example for the test listed in section A.2 the output would be:[1]

```
> (dp)
; ......all x.A(f x) -> B x by assumption u13
; ......z
; .....A(f z) -> B z by all elim
; ......all y A y by assumption u14
; ......f z
; .....A(f z) by all elim
; ....B z by imp elim
; ...all z B z by all intro
; ..all y A y -> all z B z by imp intro u14
; .(all x.A(f x) -> B x) -> all y A y -> all z B z by imp intro u13
; all f.(all x.A(f x) -> B x) -> all y A y -> all z B z by all intro
```

The dots show the depth of the proof tree that is being processed by the algorithm. The "uxx" strings are the assumptions introduced during the proof process. The numbers are internally generated by Minlog. Each line

[1] "dp" is a short for "display-proof"

has a very simple format. First it mentions the result of the transformation and then mentions the name of the transformation applied in this step.

The changes to beautify the algorithm are minimal. In fact we have already implemented a similar function into the module code, which is used for debugging purposes. That is, the final LaTeX file includes this output so that someone can see the "raw" proof process, since the actual output of the module involves a bit more high level code.

The main expression problem of this function (apart from the rough presentation) is the "flow of thought" that governs it. It starts from various assumptions, performs transformations and then reaches the final result. A human reading this proof would have difficulties to comprehend why the specific transformations were made and why in this order. Of course after seeing the final result their purpose is clear, but the reader is expected to feel "cheated". Meaning that for someone who knows already the final transformations (such as the machine) the beginning of the proof is also clear.

This is the whole concept of forward reasoning. The proof process starts from the top and goes straight down to the bottom of the proof tree[ACP00]. By proof tree we mean the usual proof notation which is the sum of all individual transformations:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall xA(f(x)) \to B(x) \qquad z}{A(f(z)) \to B(z)}\forall^- \qquad \text{u14:}\dfrac{\forall yA(y) \qquad f(z)}{A(f(z))}\forall^-}{B(z)}\to^-}{\forall zB(z)}\forall^+}{\forall yA(y) \to \forall zB(z)}\text{u14}\to^+}{(\forall x.A(f(x) \to B(x)) \to \forall yA(y) \to \forall zB(z)}\text{u13}\to^+}{\forall f.(\forall x.A(f(x) \to B(x)) \to \forall yA(y) \to \forall zB(z)}f\forall^+$$

Forward reasoning maps into a top to bottom analysis of the proof tree. Figure 6.3 shows this in a visual way.

A human however, would probably choose another path for the proof. Rarely a human would start with forward reasoning. After all, knowing in advance all the steps of the proof process from start to end would require heavy expertise on the field. The alternative path is to start from the end. We look at the final formula (what we want to prove) and move backwards. This feels sometimes more natural because one can easily guess the transformation step if the end result is known and then just needs to find a way to "reach" this step from the current context.

In the real life a mixture of the two ways is actually used. One starts with backward reasoning to reach an intermediate stage in the proof process

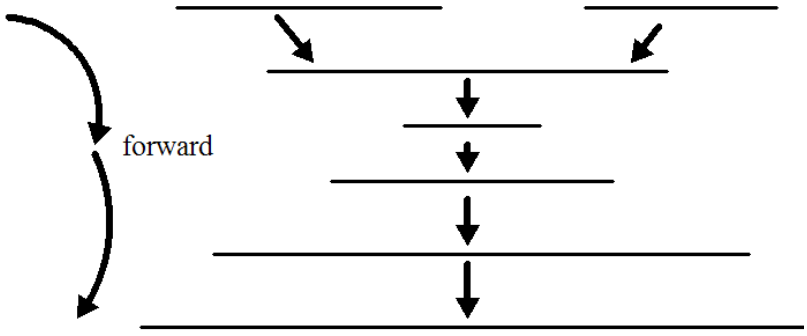


Figure 6.3: Forward reasoning (top-down)

and then realises the steps needed to get there so continues with forward reasoning. For our present example this would go like:

> (Starting with backward reasoning-from the bottom of the tree) Our goal is $\forall f.(\forall x.A(f(x) \rightarrow B(x)) \rightarrow \forall y A(y) \rightarrow \forall z B(z)$. Let f be arbitrary. Let us assume u13: $\forall x A(f(x)) \rightarrow B(x)$. Let us also assume u14:$\forall y A(y)$. Finally let z be arbitrary. We have to show now $B(z)$. (We have reached the middle of the tree).
>
> (At this point we change to forward reasoning-from the top of the tree) We use the assumption u13 ($\forall x A(f(x)) \rightarrow B(x)$) with x:=z which gives us $A(f(z)) \rightarrow B(z)$ . It remains to prove $A(f(z))$. We use the assumption u14 ($\forall y A(y)$) with y:=f(z) to obtain this. Proof finished.

A human also would use numbers for identifying assumptions rather than u13, and u14. The code implementation violated anyway the intermediate form idea since it was a direct translation of the Minlog structure to English text. Therefore it was clear that a modification of the `display-proof` function of Minlog was not going to become the final implementation of our code. For this reason we decided to start from scratch and design a solution which would:

1. Truly match the specification (intermediate form/2 stage process).
2. Allow for both backward and forward reasoning.

## 6.3 Proof analysis

With the definition and clarification of the architecture of the code, we could now focus on each stage separately. Stage 1 is the transformation of proof

into an intermediate form without considering any LaTeX matters at this point. This means that the main algorithm for stage 1 (proof analysis)

- Takes as source the Minlog internal proof.
- Has access to the Minlog system via the interfaces during processing.
- Outputs as target the intermediate form.
- Is LaTeX independent.

Since the algorithm is not bound by presentation matters the intermediate form can have any structure that we want. The important point here is that we can follow the already well chosen idea of lambda terms. Minlog uses the Curry Howard correspondence for the internal proof structure, so it is only logical to share the same idea in the module code. The details were explained in section 3.3.

Minlog provides (as we saw in section 3.3) the `display-proof-expr` function which prints out the $\lambda$-term of the proof. If we continue our example from the previous section this would give us:[2]

```
> (dpe)
(lambda (f)
  (lambda (u13)
    (lambda (u14) (lambda (z) ((u13 z) (u14 (f z)))))))
```

This is printed out in the terminal window and in mathematical text it would translate as $\lambda f, \lambda u13, \lambda u14, \lambda z, .(u13\, z)(u14\, (f\, z))$. This actually presents the proof as a series of $\lambda$-abstractions and applications allowing us to examine the "history" of this particular proof, rather than just see the finished result.

Therefore we implemented our intermediate proof form as a low level instance of the $\lambda$-terms proof structure already used in Minlog. The resulting function is the `express-proof` function in the code which outputs the intermediate form. Again the result is a long Scheme list with various components.

The next important point is the information that we need to extract from each step. The abstract $\lambda$-form of the proof shown above, is a bit too abstract for us. At the very basic level we are interested in the type of each step. For instance, we want to know that the first $\lambda$-abstraction is an all-introduction, and we want to know that the second $\lambda$-abstraction is an

[2]Again "dpe" is a short for "display-proof-expr"

implication introduction. And we are interested of course in all the formulas involved. What was the stage of the proof before *and* after each step is vital for the final presentation.

This leads us to the second set of Minlog interfaces (the first was for formulas) which are used in order to extract the various components from the Minlog proof structure. For each step the necessary information is obtained from the Minlog proof and then is stored in the intermediate form as tree node (`make-something-node` function as seen in the specification chapters).

Here is a small sample from the `express-proof-aux` function which is called by the actual `express-proof`.

```
[...in a cond block of express-proof-aux...]
   ((is-proof-in-imp-elim-form? proof)
     (let* ((op (get-op-from-imp-elim-proof proof))
    (op-formula (get-formula-from-proof op))
    (arg (get-arg-from-imp-elim-proof proof))
    (arg-formula (get-formula-from-proof arg))
    (formula (get-formula-from-proof proof)))
       (make-imp-elim-node formula (express-proof-aux op) op-formula
(express-proof-aux arg) arg-formula)))
[... rest of cond block...]
```

This describes the processing of implication elimination. We need to extract two things from the proof. Minlog names the two parts as operator and argument. These are stored in an abstract node object along with their formulas. The function is (as expected) recursive since it calls itself for the two parts in turn. Also notice that the "get-" functions are aliases for the actual Minlog functions.

## 6.4 Proof verbalisation (mixed reasoning)

Developing the code for the intermediate form export is only the half part of the process. We still need to implement the code that outputs the LaTeX file. This brings us to stage 2. The characteristics of the main stage 2 algorithm are:

- Takes as source the intermediate form produced in stage 1.
- Does *not* have access to Minlog functions.
- Outputs LaTeX source files ready to be parsed by the LaTeX tool-chain.

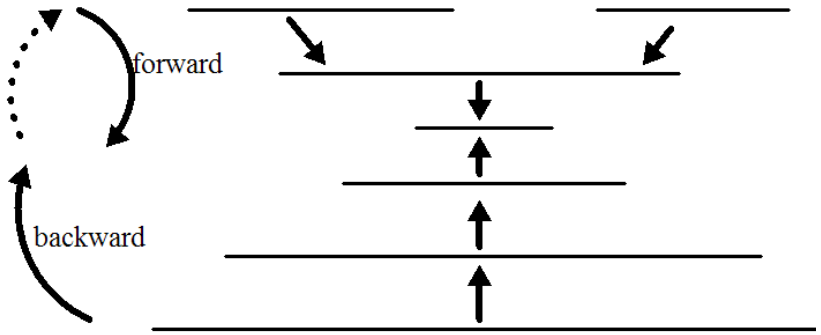


Figure 6.4: Mixed reasoning style

- Is LaTeX dependent (but Minlog independent).

The most important limitation we have for stage 2 is the inability to use Minlog functions. Stage 2 should be pure Scheme code and any communication with Minlog should have finished at this time. If we need communication with Minlog again we would actually modify the stage 1 code and store the returned information temporarily for stage 2 code to access it.

If we return to the proof verbalisation topic we still need to find a way to use mixed reasoning in the final result, and not just forward reasoning as the `display-proof` function does. As shown from the previous chapters the ideal proof verbalisation structure for our example would be:

> (Backward reasoning first) Let f be arbitrary. Let us assume u13 .... Let us also assume u14 .... Finally let z be arbitrary ....
>
> (Then forward reasoning). We use the assumption u13 .... We use the assumption u14 .... Proof finished.

If we visualise this for the proof tree of the example we get something like figure 6.4. The dotted arrow shows the change of reasoning and the "jump" from the middle of the tree to the top of it.

We have now reached another important milestone of our project. We need to find a way to express proofs in mixed style reasoning. And not just for this particular example which we have examined so far, but for all Minlog proofs. The solution has to be generic enough to accommodate all proofs, but at the same time sophisticated enough to output the expected results for all cases. We have spent a lot of time trying to find patterns in the proof that will give us some hints about the type of reasoning that seems "natural" to the reader, and when we are supposed to use each one. At first glance the only important question seems to be the point that we switch to backward

reasoning. But this is only part of the real problem since it is perfectly possible to return to forward reasoning again at some later time.

The breakthrough happened when we looked back at the $\lambda$-term representing the original proof. We include it here again for convenience.

```
> (dpe)
(lambda (f)
  (lambda (u13)
    (lambda (u14) (lambda (z) ((u13 z) (u14 (f z)))))))
```

If we compare the $\lambda$-form with the "ideal" text representation using mixed reasoning as outlined before, we can discover they they are matching! It is easy to find this if one knows where to look for. Let us break down the $\lambda$-form and see how its parts are expressed in text.

First we have a $\lambda$-abstraction. This is the first sentence of the text "Let f be arbitrary". Then we have two more $\lambda$-abstractions involving assumptions u13 and u14. The text continues as expected with "let us assume u13 ..., let us also assume u14." Finally we have one more $\lambda$-abstraction which again is the next sentence of the text stating that z is assumed to arbitrary.

Next, the $\lambda$-term changes from abstractions to applications. This is also our cue for the change of reasoning style from backward to forward. We have the $\lambda$-applications involving u13 and u14 which are the sentences in the text starting with "we use assumption...". Actually there is one more $\lambda$-application in between (an implication elimination) which is not mentioned in the text since it is considered obvious, but the general principle applies everywhere.

It should be evident now, that not only storing the proof process as a $\lambda$-term is convenient from the proof verbalisation point of view, but also we made a good decision keeping the $\lambda$-term style storage for our intermediate proof form too. The $\lambda$-term actually "guides" us in the way we express the proof in English text.

Having realised this concept, the technical part of expressing the proof does not present any major difficulties. The function responsible for stage 2 (proof verbalisation) is `verbalise-proof`, which is really just a front-end to `verbalise-proof-aux`. This functions is rather simple. It just detects the kind of nodes the intermediate form contains and calls the appropriate verbalisation function which performs the actual job of creating a LaTeX string. Here is a small sample of the code:

```
[...in a case block of verbalise-proof-aux...]
  (case (car custom-proof)
```

```
    ((all-intro)
     (verbalise-all-intro custom-proof proof-depth long?))
[... rest of case block...]
```

The `proof-depth` and `long?` arguments will be explained in the next chapter. What follows is one verbalisation function for the all-intro-node. There is one such function for each type of node that the intermediate proof form can contain.

```
(define (verbalise-all-intro all-intro-node proof-depth long?)
  (let ((english1 "Let ")
(english2 " be arbitrary")
(what (get-var-from-all-intro-node all-intro-node))
(rest (get-rest-from-all-intro-node all-intro-node))
(english3 ", and then we have to show ")
(goal (get-kernel-string-from-all-intro-node all-intro-node)))
    (string-append
          english1 what english2 english3 goal
          ". " (verbalise-proof-aux rest proof-depth #f))))
```

What is important to notice here is that the code contains no Minlog functions at all. All the accessors used are module code only. This keeps the verbalisation process (stage 2) completely Minlog independent obeying the specification explained already.

The complete result of the proof verbalisation for our example is this:

> Formula to be proved :$\forall f.\,(\forall x.\,A(fx) \to Bx) \to \forall y\,Ay \to \forall z\,Bz$
>
> Let $z$ be arbitrary, and then we have to show $(\forall A(fx) \to Bx) \to \forall yAy \to \forall zBz$. Let's assume $\forall x.A(fx) \to B(x)$[u13]. We now need to prove $\forall yAy \to \forall zBz$. Let's assume $\forall yAy[u14]$ . We now need to prove $\forall zBz$. Let z be arbitrary, and then we have to show $Bz$.
>
> We instantiate u13 with $z$ in order to obtain $A(f(z)) \to B(z)$. It remains to show $A(fz)$. We instantiate u14 with $fz$ in order to obtain $A(fz)$.

The result is far from perfect. In the next chapter we will explain some of the modifications and improvements involved in the beautification process. The basic foundations are now in place and we have the output of all derivations/transformations shown in table 6.2. Induction is a special case that will be examined later.

| Description | Module function |
|---|---|
| All introduction | `verbalise-all-intro` |
| Implication introduction | `verbalise-imp-intro` |
| Implication elimination | `verbalise-imp-elim` |
| All elimination | `verbalise-all-elim` |
| Conjunction introduction | `verbalise-and-intro` |
| Conjunction elimination (right) | `verbalise-and-keep-right` |
| Conjunction elimination (left) | `verbalise-and-keep-left` |
| Assumption variable | `verbalise-avar` |
| Axiom constant | `verbalise-aconst` |

Table 6.2: Verbalisation functions

# Chapter 7

# Expression Enrichment

At this point our implementation can deal with all kinds of proofs listed in the previous chapter in a predictable way. The presentation process (stage 2) however, can be enhanced more in order to seem more natural for human readers.

In this chapter we describe all our efforts to enrich the final English text and make it more readable, trying to avoid the mechanical output of the export algorithm. These modifications do not affect at any level the abilities of the code. They do not add new features. Instead they are verbalisation enhancements for various proof cases.

It is impossible to predict every possible case of course. Text exported automatically by a machine is doomed to look strange and cold to human readers. The effort to beautify certain cases simply exceeds our knowledge and programming expertise. This project is not about artificial intelligence or natural speech processing so we are happy to always have a working but cumbersome text output process.

## 7.1 Numbering assumptions

The first evident problem in the text output is the naming for assumptions as they are defined and used. Minlog uses small strings with a letter and a number (e.g. assumption u14). These are introduced when an assumption is created, and referenced when an assumption is used. The result is very ugly, especially since the numbers have no meaning to the human reader (they do not start from the beginning). In fact it appears that Minlog creates the numbers in a way that seems almost random to the external observer.

Therefore we needed to replace these small strings with purely numbered assumptions in order to match the common technique used by humans them-

selves. That is, convert the output so that it contains phrases like: *... we can assume $A \rightarrow B$ [1]...* and later when the assumption is used: *... using assumption [1] ....*

This proved to be one of the easiest modifications since the problem was already solved by Felix Joachimski, the author of the old texoutput module. The idea is very simple, but it works. At any moment the program uses a Scheme association list to keep track of the assumptions introduced along with their assigned number. This list is filled during stage 1 via the function `add-assumption-number`. When in either stage an assumption needs to be referenced, this list is scanned in order to obtain the matching number. The function that implements this is `look-for-assumption-number`. The program also keeps tracks of a global variable with the last assumption number used, which is accessed during the creation of a new assumption.

A minor shortcoming of the code is that it employs global storage space. The list itself is a global variable. But since it worked flawlessly and there were no performance problems in the horizon we decided to adopt the idea completely unchanged.

## 7.2 Grouping multiple assumptions

The assumption verbalisation algorithm prints out the assumption introduced and then, immediately, what needs to be done in the next step. This works pretty well when we have only one assumption. The resulting text is something like this: *We can assume X[1]. Now we still need to prove Y. ...*

Most proofs however start with multiple assumptions. In our example this would render as:

> Let $z$ be arbitrary, and then we have to show $(\forall A(fx) \rightarrow Bx) \rightarrow \forall yAy \rightarrow \forall zBz$. Let's assume $\forall x.A(fx) \rightarrow B(x)$[1]. We now need to prove $\forall yAy \rightarrow \forall zBz$. Let's assume $\forall yAy$[2] . We now need to prove $\forall zBz$. Let z be arbitrary, and then we have to show $Bz$....

This seems completely unnatural. A human reader does not want to be reminded of the rest of the proof at each step. In fact, a human reader is not interested in the intermediate steps at all but only the final one. Therefore we needed to modify the code to output something like:

> We can assume $f$, $\forall x.\, A(fx) \rightarrow Bx$[1], $\forall y\, Ay$[2], $z$. Then we have to show $Bz$.

This is the concept of assumption grouping which makes the final result much more readable. In order to create this effect however, we need to

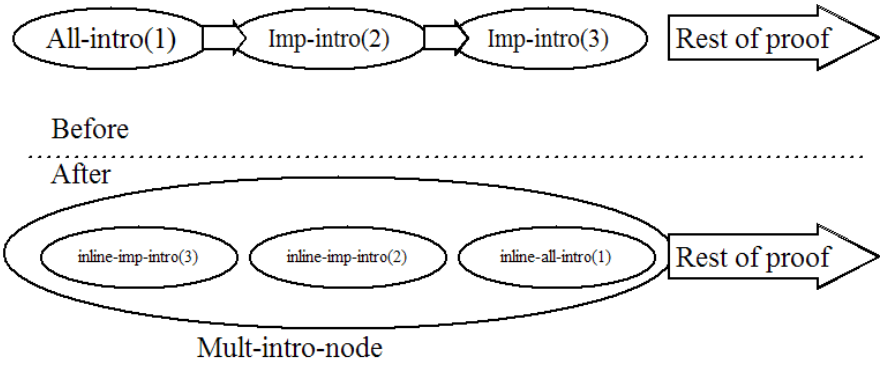


Figure 7.1: Merging multiple assumptions

override the $\lambda$-representation a bit. We create a new virtual transformation called "multiple introduction" which handles these cases.

The code essentially employs a look-ahead tactic. When it encounters an introduction transformation (either all or implication), it checks the next transformation too. If the next transformation is something else, the code just creates the respective node as normal (all-intro-node or imp-intro-node). If however, the next transformation is an introduction itself, the algorithm creates a new mult-intro-node and merges the two introductions together, keeping only the result of the second one since this will interest the reader in the end. The code is of course recursive in order to handle multiple introductions and not just two. Figure 7.1 shows the idea in a simplified way.

Notice that although the technique is depicted as being linear, this is not the truth. The *lambda*-proof structure is a tree so the "look-ahead" process actually scans in tree nodes. This process is performed by the `custom-intro-to-intro-list-and-kernel` function. Some essential modifications are also present in the `express-proof-aux` function.

The individual introduction nodes are stored in a compact form inside the host mult-intro node. They are stored inside mini nodes which are abstract node objects themselves. The respective constructors are `make-inline-imp-intro-node` and `make-inline-all-intro-node` which are bundled with associated accessor functions. The whole process takes place during stage 1. Code in stage 2 is completely unaware of this since it makes use of the mult-intro node for needed information.

It is convenient programmatically to store the inline nodes in reverse order inside the host mult-intro node. This happens because we consider important only the last transformation which will be printed in the *We still need to prove...* part. This might seem awkward when looking directly at

the intermediate form. The verbalisation functions (stage 2) are aware of this fact of course and parse the node in reverse again, therefore emitting the assumptions in the correct order as already demonstrated.

## 7.3 Grouping multiple all eliminations

A similar approach has been followed in the case of eliminations. In this case however, we decided that we should only deal with all-eliminations and not implication eliminations.

One all-elimination translates into *we instantiate $f(x)$ with $x$ in order to obtain ...*. If the subsequent transformations are all-eliminations too, the result is a bit annoying for the human reader, since the text mentions the formula after each individual transformation. Most humans would choose to group the all-elimination steps together and jump straight to the final result.

To overcome this, we modified radically the all-verbalisation code (stage 2) to take into account this fact. The code looks ahead and groups all elimination together. This time however we leave the intermediate form unchanged. That is, the $\lambda - form$ of the proof still contains the chain of all-elimination steps. Only the verbalisation stage has a different behaviour.

The modifications can be demonstrated with a simple example. We assume that we need to prove the formula: $(\forall x, y.\, Ax \wedge Bx) \rightarrow \forall x\, Ax$ . The old code would produce a result like the following:

> *Proof:* We can assume $\forall x, y.\, Ax \wedge Bx[1],\ x$. Then we have to show $Ax$. We instantiate assumption [1] with x in order to obtain $\forall y.\, Ax \wedge Bx$. We instantiate this with y in order to obtain $Ax \wedge Bx$. Keeping only the left part of this we get $Ax$.

The human reader does not want to see the assumption[1] first instantiated with $x$ and then with $y$. Instead he/she wants the final result. The modified code produces output like the following:

> *Proof:* We can assume $\forall x, y.\, Ax \wedge Bx[1],\ x$. Then we have to show $Ax$. We instantiate assumption [1] with $x$, $y$ in order to obtain $Ax \wedge Bx$. Keeping only the left part of this we get $Ax$.

The change might not seem important, but for more than two all-eliminations the improvement is noticeable. In mathematical proofs 3 or 4 all-elimination steps are not uncommon and in these cases the English text is more readable.

## 7.4 Layout considerations

Testing small proofs is easy. Sentences are small and clear. Because the resulting text is short, it is easy to follow the steps of the proof. The scalability of the module can only be observed if really big proofs are used as input. The output is also very interesting when the proof has more that one active goals. For example in a conjunction introduction step one must provide proofs for both parts of the formula before taking the conjunction as granted.

This issue was also taken into account by the the previous texoutput module. We used the same approach introduced there but of course we had to write code for our particular implementation.

Proofs are now separated in two kinds. First we have the so called simple proofs which we expect to be small in depth, and the non-simple ones which have a non-trivial $\lambda$-term (more than one step). This separation is just used for presentation purposes. Technically all proofs are of the same format. The idea is that when we find a simple proof we know in advance that their rendering in English text will be compact, while non-simple proofs can take an unknown number of sentences to explain. We use this in the verbalisation step (stage 2) inside the program to make some "smart" decisions explained in this section.

Currently we consider as simple proofs, assumption constants and assumption variables. See the definition of the `simple-proof?` function.

### 7.4.1 Subproofs

The first presentation enhancement is the use of bulleting and additional indentation to show in a clear way the goals being active. Typical examples of this are with conjunction-introduction (as already mentioned) and implication elimination. Basically every proof transformation which can introduce or depend on two smaller proofs needs this.

The program logic in these cases checks whether the smaller proofs are simple or not. If they are, we assume that the reader will not be confused since each goal will be proved in one/two sentences. If however we have non-simple proofs then we insert bullet points and indentation to signify the start of a new subproof. Of course it is also possible for a subproof to have subproofs on its own. In order to help the reader keep track the number of subproof levels at any time, we append the proof depth number (starting from zero) to the bullet point of each subproof. An actual rendering of this can be demonstrated in the example listed in section A.5.

### 7.4.2 Selective formula listings

Another small problem which can be found in the automatic proof export, is the formula where all the transformations happen. The program reminds the reader at every individual step what is the source formula. This is not wrong, but again it does not feel natural to the human reader.

Humans usually introduce important objects the first time they appear. Afterwards they refer to them as "this" and "that". Consider for example the following proof text:

> Formula to be proved :$(\forall x, y.\, Ax \wedge Bx) \rightarrow \forall x\, Ax$
>
> *Proof:*We can assume $\forall x, y.\, Ax \wedge Bx[1]$, $x$. Then we have to show $Ax$. We instantiate assumption [1] with x in order to obtain $\forall y.\, Ax \wedge Bx$. We instantiate $\forall y.\, Ax \wedge Bx$ with y in order to obtain $Ax \wedge Bx$. Keeping only the left part of $Ax \wedge Bx$ we get $Ax$.

In this simple case where we have only one flow of transformations, the reader can easily keep track of the formula at each step by himself/herself. Therefore we enhanced a bit further the export algorithm to mimic the human expectation. The modified algorithm would output the same proof as following:

> Formula to be proved :$(\forall x, y.\, Ax \wedge Bx) \rightarrow \forall x\, Ax$
>
> *Proof:*We can assume $\forall x, y.\, Ax \wedge Bx[1]$, $x$. Then we have to show $Ax$. We instantiate assumption [1] with x in order to obtain $\forall y.\, Ax \wedge Bx$. We instantiate `this` with y in order to obtain $Ax \wedge Bx$. Keeping only the left part of `this` we get $Ax$.

Notice how the last two steps of the proof refer to "this". And of course it is easy to see that "this" refers to the formula of the previous transformation step. The result seems slightly better now to the reader.

Of course we cannot simply use "this" all the time. The first sentence should always show all formulas. Also each time we start a subproof we need to show the formulas again. Basically when things get complicated (e.g. multiple transformation flows, long proofs) we need to mention the current formula. In simpler case we can get away with "this" and "that". Also it is better to mention a formula when is not actually needed rather than the opposite case. The reader will get very upset if he/she cannot understand what "this" refers to, while a redundant formula appearance will just present a minor annoyance.

To accommodate this program behaviour we curry a boolean `long` flag across transformation code. During the verbalisation of the proof (stage 2) the algorithms query this flag to decide whether or not they should mention the formula. At every recursive call of the verbalisation functions, there is also the additional decision how the flag would be transfered to the next step.

All these decisions are based also on the concept of simple and non-simple proofs explained in the previous subsection. In a similar manner we examined the previous texoutput module code and adapted the code logic for our implementation.

## 7.5 Induction

Induction proofs are very important. A lot of proofs use induction either as an intermediate or as the last step. We have not mentioned induction in the previous chapters. The reason for this, is the Minlog handles induction proofs in a special way.

Induction is not a separate derivation (like implication elimination or conjunction introduction). This was done on purpose since induction would break the concept that each each derivation is a $\lambda$-abstraction/application. That is, an induction cannot be stored (directly) as a $\lambda$-term. This means effectively that there are not Minlog functions which give us the base or step case given a derivation, because simply there is not such derivation type.

To store an induction, Minlog actually encapsulates it in the existing infrastructure. Induction in Minlog is a hard-coded axiom without any special properties. The axiom formula is:

$$A(0) \to \forall x(A(x) \to A(x+1)) \to \forall x A(x)$$

Induction is used in order to prove $\forall$-formulas. For a given formula $A(x)$ we need to prove that it holds always for every possible input. That is the last part of the axiom $\forall x A(x)$. We can do this if have a base case and the step case. If we assume that our base case is for $x = 0$ (but it can also be for any other value, then first we need to prove this. This is the $A(0)$ part of the axiom. To finish the induction we need to prove the step case too. Given $x$ and assuming $A(x)$ we must prove that the formula is also true for the successor of $x$. That is the $\forall x(A(x) \to A(x+1))$ part of the axiom. If these two cases can be proved (base and step) then the whole induction formula is also true. We can see that the cases and results are connected with implications to show this fact.

Although this implementation of induction makes Minlog less complicated and more focused on its root concepts, induction extraction for the texoutput

module becomes extremely difficult. Because an induction is hidden inside the usual derivations (in this case in implication eliminations) we need again to provide look-ahead code which pre-scans the proof structure in order to detect inductions prior to the verbalisation stage.

If for example the proof is:

```
(imp-elim (...)
          (imp elim (...)
               (aconst (...) "ind" (...))))
```

we need our code to detect that we have an induction *before* the verbalisation of the preceding implication elimination steps. The two elimination steps contain the base and step case for the induction, so all information must be gathered in order to print out the induction proof in a proper way.

The current implementation of the code handles induction proofs in this manner exactly. Stage 1 code scans ahead the proof and prepares a virtual induction tree node, while stage 2 just processes the induction node without any communication with Minlog. There are two limitations imposed on the program. The first is the detection of the induction. We need to find what patterns do induction proofs have, so that we can include the appropriate case in the look-ahead code. The second is that we must also extract correctly the base and step case from the previous derivations leading to the induction at every time.

If an induction is undetected by the program it will be printed as a normal axiom. The `display-proof` function does that. Since it does not include any special logic for induction it just prints out sentences like "...we use the axiom ind to prove ..." and then a very long formula which is the induction axiom as already explained.

Basically, induction is a tricky situation which is completely separated from the rest of the code, and thus needs special attention on our behalf.

# Part IV

# Conclusions

# Chapter 8

# Evaluation

Like any other project, several problems appeared during the code development. Several compromises were made until the final version of the code. There are parts which are really stable and other parts where only partial support is provided. This chapter examines several aspects of the present implementation of the code. We also discuss what has changed since the initial design, what still needs to be implemented, and what improvements we would like to see in the Minlog system.

## 8.1 Contributions

The finished software module is a single Scheme source file. It can be loaded into Minlog like any other Scheme code segment. Once the file is loaded, a notice is shown mentioning the revision number for it.

The module functions which are expected to be run by the user are two.

1. The `proof-to-latex` function which triggers the export process.

2. The `set-latex-filename!` function which allows the user to change the default name of the LaTeX file.

The `proof-to-latex` function has a variable number of arguments. If no argument is given the proof contained in `(current-proof)` will be exported to a LaTeX file with the default name. If one argument is given, it is assumed to be a proof structure, which will be exported again to the default LaTeX file. If two arguments are provided, the first one is assumed to represent a proof as before but the second one shows the name of the LaTeX file.

The `set-latex-filename!` just sets the default name of the exported file which will be used when not defined explicitly as described above. The initial

default filename is “default.tex”. In all cases the name of the output file is printed before the transformation process so the user knows where he/she can find the resulting file. It is also possible to use a file path along with the filename as an argument to `set-latex-filename!`. In this case both kind of slashes[1] are accepted regardless of the underlying platform.Table 8.1 outlines the user interface of the module.

| Command | Description |
|---|---|
| (load ”texoutput4.scm”) | Loads the module into Minlog |
| (proof-to-latex) | Exports current-proof to default file |
| (proof-to-latex prf) | Exports proof named ”prf” to default file |
| (proof-to-latex prf ”out.tex”) | Exports proof named ”prf” to ”out.tex” |
| (set-latex-file! ”result.tex”) | Changes default filename to ”result.tex” |

Table 8.1: Module usage

## 8.2 Correctness

Unlike other projects, it is hard to define correctness in our case. There is no “correct” way to express a proof. Different people will have different opinions of how a proof should be exported. There is no doubt about that. Additionally it is very difficult (if not impossible) to create a module capable of expressing proofs that seem to be written by a human. Anyone who reads a proof exported by the module can easily identify the hard-coded, straightforward thinking of a machine. Especially for big proofs which have non-trivial output the mechanical face of the English text is evident even by non-technical people.

We consider the proof exported by the module to be correct when:

- The English text has correct syntax/grammar/spelling.
- The mathematical content is correct.

Whether the text looks natural to humans or not is a big topic on its own. Therefore “correct” results are expected to contain text that is clearly automatically generated by the machine.

Aside from this fact, the module is capable of expressing the most common proofs that Minlog supports. Several advanced features of Minlog were not implemented due to excessive complexity or time limitations.

[1] Dos-style or Unix-style

In the unlike event where the module cannot handle the input given, the appropriate function will exit gracefully mentioning the kind of input it expected, and the actual input it received. This makes every problem easy to detect and can greatly assist in the isolation and correction of the problem.

Finally it is clear that testing done during and after development can only uncover the most obvious shortcomings. Rigorous testing can only be achieved by the real users of the systems. Casual Minlog users would really fall into the imperfections of the implementation. And only advanced Minlog users would actually push the module to its limits when the attempt to express really complex proofs. Therefore we cannot at this stage guarantee a bug-free implementation.

One important point that we should emphasise is the fact that the module is independent of the Minlog system. That is, it only gets information from Minlog without changing anything. If for some unknown reason the tex module crashes during the export process, the Minlog system will be unaffected. The user does not lose anything. The module creates its own copy of the proof structure and so the original one is untampered. This is another advantage of having built the module with robustness and stability in mind.

## 8.3 Efficiency

Minlog is a relative small system (in comparison with the other proof checkers). So far performance does not seem to suffer regardless of platform. The module is also very compact and the footprint of the code is minimal. After all, Minlog does not have a graphical user interface. Expressing a proof is almost instant and the processing time is not visible for the user.

It is worth noticing however, that unlike all other Minlog operations the proof export process is expected to take double time to traverse the proof tree. This is because a proof tree is parsed twice. The first time is during stage 1 when the intermediate form is created and the second in stage 2 when the the final LaTeX file is created from the intermediate form. Inspite of being more compact, the intermediate form has exactly the same depth and structure as the original proof structure. Also most major function calls are *not* tail recursive so with really long proofs there is always the danger of stack overflow. With modern systems however, this should not be a concern.

Another interesting issue is the look-ahead code. Currently pre-scanning of the proof tree is performed in multiple introduction detection and in induction extraction.

The multiple introduction algorithm really looks at the next step only. Additionally this happens only during stage 1. In stage 2 multiple intro-

duction is just another normal node in the proof tree. So it is not worth examining the performance impact of this.

The induction algorithm is more aggressive since it scans 4 steps ahead in the expression process. The number however is always the same (constant), so the actual overhead is $O(1)$ which is absorbed by the complexity needed in order to traverse the proof tree itself. And as the multiple introduction algorithm, induction detection happens only during stage 1 (that is the first pass).

In conclusion the computational impact of the module is trivial compared with the full Minlog system. But on modern workstations Minlog is already fast enough, so that bothering at this stage with module efficiency becomes unimportant. As stated in the specification, stability and robustness are the primary goals.

## 8.4 Extensibility

This is an aspect where we would like to think that the module really shines. We spent a great deal of effort to address all the minor details that lead to extensible and portable code. As Minlog evolves so must the module. We ought to make life easier for the next programmer who wishes to upgrade the module next time.

This is accomplished by:

- The clear separation between stage 1 code (Minlog proof to intermediate code) and stage 2 (intermediate code to LaTeX document.
- The object oriented structure of the module code.

We have already explained the theory behind data abstraction. We can now demonstrate how it works in practice. We will start from the intermediate form. For example the (simplified) code for and elimination (left) is

```
(define (make-and-keep-left-node formula kernel)
  (list
    'and-keep-left
    (convert-formula-to-latex-string formula)
    kernel))

(define (get-result-from-and-keep-left-node node)
  (list-ref node 1))
```

```
(define (get-body-from-and-keep-left-node node)
  (list-ref node 2))
```

Now we assume that the programmer wishes to change the tag symbol to "conjunction-elim-left" and also reverse the internal components. The code will become:

```
(define (make-and-keep-left-node formula kernel)
  (list
    'conjunction-elim-left
    kernel
    (convert-formula-to-latex-string formula)))


(define (get-result-from-and-keep-left-node node)
  (list-ref node 2))

(define (get-body-from-and-keep-left-node node)
  (list-ref node 1))
```

Next he/she wishes to add some preprocessing function for the kernel part and some post-processing for the formula part. After the changes, the code might look like:

```
(define (make-and-keep-left-node formula kernel)
  (list
    'conjunction-elim-left
    (preprocess kernel)
    (convert-formula-to-latex-string formula)))


(define (get-result-from-and-keep-left-node node)
  (postprocess (list-ref node 2)))

(define (get-body-from-and-keep-left-node node)
  (list-ref node 1))
```

Finally he/she decides to get rid of the simple list structure and use a highly optimised custom data structure named "blackbox" (which behaves like a hashtable).The final code would be:

```
(define (make-and-keep-left-node formula kernel)
  (begin
    (blackbox-store (preprocess kernel) 'kernel)
    (blackbox-store (convert-formula-to-latex-string formula) 'formula)))


(define (get-result-from-and-keep-left-node node)
  (postprocess (blackbox-get 'formula)))

(define (get-body-from-and-keep-left-node node)
  (blackbox-get 'kernel))
```

Even after all these modifications the rest of the module code will remain unaffected as long as the name of the functions stay the same (as in our case) and the accessor functions return the results expected by the rest of the module. Also notice, that changes happen only locally. There is no need to track others segments of the code, because they all use the accessor functions and never the actual internal structure. The blackbox structure can have on its own a complicated implementation which is simply of no concern to rest of the code, since it never deals with the blackbox structure directly. Everything happens via the accessor functions. In this simple case, all this might seem an overkill, but we used it to demonstrate how easily we can achieve scalability using multiple layers of software abstraction.

In a similar manner one can modify the backend which outputs LaTeX files and leave unchanged the front-end which obtains the proof structure from Minlog. The code which outputs LaTeX markup for all introduction follows:

```
(define (verbalise-all-intro all-intro-node proof-depth long?)
  (let ((english1 "Let ")
(english2 " be arbitrary")
(what (get-var-from-all-intro-node all-intro-node))
(rest (get-rest-from-all-intro-node all-intro-node))
(english3 ", and then we have to show ")
(goal (get-kernel-string-from-all-intro-node all-intro-node)))
    (string-append english1 what english2 english3 goal ". "
   (verbalise-proof-aux rest proof-depth #f))))
```

If in the future we need to add XML export (the next section explains why the module does not support XML at present) this can be easily modified to:

```
(define (verbalise-all-intro all-intro-node proof-depth long?)
  (let ((root-start "<all-intro>")
```

```
(root-end "</all-intro>")
(what (get-var-from-all-intro-node all-intro-node))
(rest (get-rest-from-all-intro-node all-intro-node))
(goal (get-kernel-string-from-all-intro-node all-intro-node)))
    (string-append root-start "<var>" what "</var>" "<goal>" goal "</goal>"
     "<rest>"
     (verbalise-proof-aux rest proof-depth #f) "</rest>"
     root-end)))
```

It is our belief that once someone understands the structure of the code, it is easy to make any kind of change (radical or trivial) with minimum effort, since there is a great deal of infrastructure code for modularisation and data abstraction.

## 8.5 Project review

The LaTeX module can easily handle most basic proofs. However most advanced constructs of Minlog (with the only exception being induction at the moment) are not supported. These include:

- Full termrewriting support.
- Ex-elimination.
- Ex-introduction.

The main reason behind this fact, is that Minlog simply does not have the necessary facilities for their expression. That is, Minlog does not export enough interfaces which will allow an external entity (in our case the texoutput module) to retrieve all information needed for verbalisation. At the moment Minlog is centered around the interactive proof process, rather than the extraction of the steps applied *after* the proof is finished. This is easily identified from the simplicity of the `display-proof` function which ignores constructs like induction and ex-elimination/introduction and just prints them as axioms along with the raw associated formulas. Currently Minlog holds enough information on the *correctness* of proofs but not on *why* they are correct.[HBC99]

On one hand the proof structure holds information which is irrelevant to the expression of a proof, and that is why it needs preprocessing before verbalisation (the intermediate form idea), but on the other hand it lacks information which *is* relevant to the texoutput module. For example in

the case of term-rewriting it would be convenient for the proof to hold the formula before *and* after normalisation. Minlog holds only the result and it assumes that one can derive the original formula from the goal that needs to be proved. This however is not as simple as it sounds, especially for a program. A human can look at $0 + 0 = 0$ and know immediately that this maps to TRUE. A program however sees this as another formula no "simpler" than the others.

This issue is already known for other proof systems. Quoting from [HBC99] where the Nu/PRL system is examined.

> "However, as we have noted, there is information that is needed for communication that is not stored in the proof tree, either because it is part of the background mathematics knowledge assumed in the system or because it was determined by the proof agent in a manner that did not produce a suitable proof object. This is the information that we must determine by appealing to the theorem prover."

The suggestion is that there should be a two-way communication between the theorem prover and the expression module, so that queries can be made from both sides. We already mentioned that Minlog stores induction/ex-elim/ex-intro as special axioms and not as individual transformations. This might be convenient technically from Minlog's point of view, but it is a big obstacle for the modules which are trying to export needed information. The solution on which induction is based on (pattern matching) is not the optimal one. Minlog should take this into account and export additional information which is needed for an external module. Actually this is very common for large software systems. Databases, Web servers, firewalls always include some kind of framework which provides information about the state of the system to the "external" world in a formatted and predetermined way. Minlog is currently a "closed" system in this sense.

Minlog's architecture is essentially flat. Every function is available from everywhere and every subroutine can modify any internal structure of the program. There is no notion of package or namespace inside Minlog. What we really need is a Minlog system which is *platform* for proof checking rather than a single proof checker program.

One could claim that instead of waiting, we should just define our own interfaces and get all information we need ourselves. This would make the module completely dependent on the Minlog system. Messing with the Minlog internals in a deep level will beat the concept of a software module. We think this topic should be clear from the specification section.

The decision to follow the software module approach is not always a restricting factor. For example with the pattern matching code for induction we are able to express induction on lists (apart from natural numbers) since Minlog handles induction in a more abstract form. This in effect allows the module to handle examples like this shown in section A.8, a capability which was not present in the previous version of the texoutput module.

# Chapter 9

# Future Considerations

This final chapter examines possible extensions to the module and explains the lack of XML support in the current version.

## 9.1 Regarding XML

The module does not include any options for XML output. The only output format at present is LaTeX. This does not mean that we did not think that XML was important. On the contrary, we thought that XML is so important that it should either be implemented in the correct way or not at all.

Timing constraints played a limited role in the decision not to include XML as a target format. The primary reason however, was that the "optimal" way to incorporate XML in the project required effort that exceeded a single MSc project and in addition should be attempted only after close co-operation with the other proof Systems that exist in the field. We will elaborate on this.

Following a brief research on XML and associated solutions and tools, it became clear that the integration choices were actually two.

1. Using XML as a presentation medium (the easy way).

2. Using XML as the interchange medium (the hard/correct way).

The first solution would involve the output of an XML file readable by existing generic tools. One of the first XML dialects was actually Math-ML which was devised specifically for mathematical content. There are pre-constructed Math-ML tools which allow for the manipulation, production and presentation of Math-ML documents.

So a small modification of the texoutput module would be the ability to export XML files which are Math-ML valid. This idea however, was soon

abandoned after witnessing the maturity of the XML platform in general. XML is no doubt a revolutionary idea however, the technology itself is rather new. The tools available are rather primitive and with the exception of XML parsers, the rest of the software tools are limited. In fact some of the XML tools prefer to build temporary on already established solutions in order to handle the output process. For example there are tools which convert XML files to PDF, passing from LaTeX first. That is, the XML converter uses stylesheets to convert the XML file to LaTeX source, and the rest of the conversion is handled by the native LaTeX tool-chain which has quality PDF export. Naturally this would seem pointless in our case since we can already export directly LaTeX files.

Of course there are tools which will convert XML to PDF files in one step, but they cannot match the LaTeX tool-chain which is well established and very mature. So in the end there is not any real advantage for using XML as a presentation medium.

It is important to note that this usage of XML is a side effect of the technology. The primary reason behind the invention of XML was information (document) exchange and the support of architecture independent Web services. Web services powered by XML can be produced and consumed by any machine on the network regardless of the underlying platform technology. So instead of using Math-ML (which just deals with mathematical content) for *presentation* of Minlog proofs, we should use XML for the *storage* of Minlog proofs[STL00].

Basically this means that we need to define a special Minlog XML Schema (or DTD) which will describe the structure of an XML document in Minlog terms, and not in plain mathematical terms. The result would be an XML document with a root element of `<proof>` and special elements/attributes describing the individual proof manipulation steps (e.g. all elimination, implication introduction e.t.c). This is a very difficult undertaking which can prove a disaster if taken lightly. This solution depends on two important concepts:

1. Availability of assorted tools.

2. A well chosen/defined XML Schema.

If we define our own XML Schema for Minlog we are essentially creating a new XML dialect which needs the full software stack in order to be useful. That is:

- XML export (this Minlog module).

- XML import (another Minlog module).
- XML parser/validator.
- XML converter for this Schema/DTD.
- XML stylesheets for various formats.
- Native or generic viewers for the final result.

Of these the only thing available are the XML parsers. We could make then the module to export an XML file following our Minlog specific Schema, but this file would be completely useless without the rest of the software tools. We could verify that this XML file was actually valid, but nothing else. The stylesheets are those that really allow us to visualise and understand an XML file. Another Minlog module needs to be implemented too, in order to read XML files and import the proof described in the Minlog system.

The definition of the XML Schema is also very important. The worst thing that can happen here is the definition of individual XML Schemas from each Proof system (e.g. Isabelle, Coq e.t.c.) in complete isolation from each other. This would beat the whole concept of XML for information exchange since now one would have to convert files between the different XML dialects. In fact the choice of the XML Schema should be a common decision between the involving parties, in order to have in the end a unified way to represent proofs. This would potentially allow for the exchange of proofs between different systems. We do not claim that this can happen easily. At least in the beginning, only the intersection of available features should be formalised and as systems mature the XML Schema could be refined when needed.

It should be clear now, that choosing quickly an XML Schema just to "advertise" that the module supports XML export, would be completely naive and premature. Technically the modifications that need to be done in the module code for XML export, are not major. In fact, with the clear separation of the two stages in the code, exporting XML elements from the intermediate proof form is trivial programmatically. But the problem here is not the programmatical effort, but rather a common agreement between the members of the Logic field in Computer Science.

## 9.2 Possible extensions

The tex module is no exception to the rule. Although the bulk of the needed functionality is implemented, there is always some room for improvements. What is needed first is quality assurance for the current code. Rigorous

testing from advanced Minlog users with complex proofs is expected to reveal a lot of limitations of the current implementation. It is also possible that expert Scheme programmers can identify many common problems inside the code which need corrections.

What follows next is probably the inclusion of all advanced Minlog features which did not make it into the current implementation. Some of the advanced Minlog users will almost certainly find the current implementation limited, and lack of needed features will be detrimental for them.

Finally the crown jewel of Minlog could become the ability to export XML proofs readable by other interactive proof systems too. With an additional XML import module, the Minlog system will become a robust and extensible tool among the research community.

# Part V

# Appendix and References

# Appendix A

# Example sessions

This chapter includes a small collection of various examples that demonstrate the status of the finished module. In simple cases we make use of the automatic proof search feature, while in more complex ones we list explicitly the needed steps.These are the same examples that were used during development of the module, so the reader is encouraged to try his/her own ideas.

For each example we include:

- The Minlog commands for its creation.
- The output of the `display-proof` function.
- The output of the `display-proof-expression` function. That is, the $\lambda - form$ of the proof.
- The intermediate list form.
- The finished (rendered) LaTeX presentation.

## A.1 General example

This example shows most concepts mentioned in the text. Assumption grouping, assumptions numbering, subproofs, mixed reasoning e.t.c

The Minlog commands (included in `all1.scm`) are:

```
(add-pvar-name "A" "B" (make-arity (py "alpha")))
(add-var-name "x" (py "alpha"))

(set-goal (pf "(all x. A x -> B x) -> all x A(x) -> all x B(x)"))
(search)
```

The output of the `display-proof` function is:

```
> (dp)
; .....all x.A x -> B x by assumption u13
; .....x
; ....A x -> B x by all elim
; .....all x A x by assumption u14
; .....x
; ....A x by all elim
; ...B x by imp elim
; ..all x B x by all intro
; .all x A x -> all x B x by imp intro u14
; (all x.A x -> B x) -> all x A x -> all x B x by imp intro u13
```

The $\lambda - form$ of the proof is:

```
> (dpe)
(lambda (u13) (lambda (u14) (lambda (x) ((u13 x) (u14 x)))))
```

The intermediate list form is:

```
> (express-proof (input-proof))
(mult-intro
  "$(\\forall x.\\,A x \\to B x) \\to \\forall x\\, A x \\to \\forall x\\, B
x$"
  ((inline-all-intro "$B x$" (x))
   (inline-imp-intro
     "$\\forall x\\, B x$"
     "$\\forall x\\, A x$"
     u14)
   (inline-imp-intro
     "$\\forall x\\, A x \\to \\forall x\\, B x$"
     "$\\forall x.\\,A x \\to B x$"
     u13))
  (imp-elim
    "$B x$"
    (all-elim
      "$A x \\to B x$"
      (avar "$\\forall x.\\,A x \\to B x$" u13)
      "x")
    "$A x \\to B x$"
    (all-elim "$A x$" (avar "$\\forall x\\, A x$" u14) "x")
    "$A x$"))
```

And the LaTeX rendering is:
Formula to be proved: $(\forall x.\, Ax \to Bx) \to \forall x\, Ax \to \forall x\, Bx$
*Proof:* We can assume $\forall x.\, Ax \to Bx[1]$, $\forall x\, Ax[2]$, $x$. Then we have to show $Bx$. We can derive this from $Ax$ and $Ax \to Bx$.

- $\bullet_0$ We instantiate assumption [2] with $x$ in order to obtain $Ax$.
- $\bullet_0$ We instantiate assumption [1] with $x$ in order to obtain $Ax \to Bx$.

## A.2 The main example of the text

This example is examined in detail by the document text. Its proof tree is also included in the text showing backward and forward reasoning.

The Minlog commands (included in `example1.scm`) are:

```
(add-var-name "f" (py "alpha=>alpha"))
(add-predconst-name "A" "B" (make-arity (py "alpha")))
(add-var-name "x" "y" "z" (py "alpha"))
(set-goal (pf "all f . ( (all x. A(f(x)) -> B(x)) ->
  all y A(y) -> all z B(z))"))
(search)
```

The output of the `display-proof` function is:

```
> (dp)
; ......all x.A(f x) -> B x by assumption u13
; ......z
; .....A(f z) -> B z by all elim
; ......all y A y by assumption u14
; ......f z
; .....A(f z) by all elim
; ....B z by imp elim
; ...all z B z by all intro
; ..all y A y -> all z B z by imp intro u14
; .(all x.A(f x) -> B x) -> all y A y -> all z B z by imp intro u13
; all f.(all x.A(f x) -> B x) -> all y A y -> all z B z by all intro
```

The $\lambda - form$ of the proof is:

```
> (dpe)
(lambda (f)
  (lambda (u13)
    (lambda (u14) (lambda (z) ((u13 z) (u14 (f z)))))))
```

The intermediate list form is:

```
> (express-proof (input-proof))
(mult-intro
  "$\\forall f.\\,(\\forall x.\\,A(f x) \\to B x) \\to \\forall y\\, A y \\to
\\forall z\\, B z$"
  ((inline-all-intro "$B z$" (z))
   (inline-imp-intro
     "$\\forall z\\, B z$"
     "$\\forall y\\, A y$"
     u14)
   (inline-imp-intro
     "$\\forall y\\, A y \\to \\forall z\\, B z$"
     "$\\forall x.\\,A(f x) \\to B x$"
     u13)
   (inline-all-intro
     "$(\\forall x.\\,A(f x) \\to B x) \\to \\forall y\\, A y \\to \\forall
z\\, B z$"
     (f)))
  (imp-elim
    "$B z$"
    (all-elim
      "$A(f z) \\to B z$"
      (avar "$\\forall x.\\,A(f x) \\to B x$" u13)
      "z")
    "$A(f z) \\to B z$"
    (all-elim "$A(f z)$" (avar "$\\forall y\\, A y$" u14) "f z")
    "$A(f z)$"))
```

And the LaTeX rendering is:

Formula to be proved: $\forall f.\,(\forall x.\,A(fx) \to Bx) \to \forall y\,Ay \to \forall z\,Bz$

*Proof:* We can assume $f$, $\forall x.\,A(fx) \to Bx$[1], $\forall y\,Ay$[2], $z$. Then we have to show $Bz$. We can derive this from $A(fz)$ and $A(fz) \to Bz$.

- $\bullet_0$ We instantiate assumption [2] with $fz$ in order to obtain $A(fz)$.
- $\bullet_0$ We instantiate assumption [1] with $z$ in order to obtain $A(fz) \to Bz$.

## A.3 Distribution Law

The Minlog commands (included in `distr.scm`) are:

```
(add-predconst-name "A" "B" "C" (make-arity))
(define distr (pf "(A -> B -> C) -> (A -> B) -> A -> C"))
(set-goal distr)
(search)
```

The output of the `display-proof` function is:

```
> (dp)
; .....A -> B -> C by assumption u13
; .....A by assumption u15
; ....B -> C by imp elim
; .....A -> B by assumption u14
; .....A by assumption u15
; ....B by imp elim
; ...C by imp elim
; ..A -> C by imp intro u15
; .(A -> B) -> A -> C by imp intro u14
; (A -> B -> C) -> (A -> B) -> A -> C by imp intro u13
```

The $\lambda - form$ of the proof is:

```
> (dpe)
(lambda (u13)
  (lambda (u14) (lambda (u15) ((u13 u15) (u14 u15)))))
```

The intermediate list form is:

```
> (express-proof (input-proof))
(mult-intro
  "$(A \\to B \\to C) \\to (A \\to B) \\to A \\to C$"
  ((inline-imp-intro "$C$" "$A$" u15)
   (inline-imp-intro "$A \\to C$" "$A \\to B$" u14)
   (inline-imp-intro
     "$(A \\to B) \\to A \\to C$"
     "$A \\to B \\to C$"
     u13))
  (imp-elim
    "$C$"
    (imp-elim
      "$B \\to C$"
      (avar "$A \\to B \\to C$" u13)
      "$A \\to B \\to C$"
      (avar "$A$" u15)
```

```
    "$A$")
  "$B \\to C$"
  (imp-elim
    "$B$"
    (avar "$A \\to B$" u14)
    "$A \\to B$"
    (avar "$A$" u15)
    "$A$")
  "$B$"))
```

And the LaTeX rendering is:

Formula to be proved: $(A \to B \to C) \to (A \to B) \to A \to C$

*Proof:* We can assume $A \to B \to C$[1], $A \to B$[2], $A$[3]. Then we have to show $C$. We can derive this from $B$ and $B \to C$.

- $\bullet_0$ We can easily derive $B$ from assumption [2] and assumption [3].
- $\bullet_0$ We can easily derive $B \to C$ from assumption [1] and assumption [3].

## A.4 Multiple all elimination

This example demonstrates the all elimination grouping since it contains 4 all elimination in a row.

The Minlog commands (included in `fourall.scm`) are:

```
(add-pvar-name "A" "B" (make-arity (py "alpha")))
(add-var-name "x" (py "alpha"))
(add-var-name "y" (py "alpha"))
(add-var-name "z" (py "alpha"))
(add-var-name "w" (py "alpha"))

(set-goal (pf "(all x,y,z,w (A(x) & B(x)) -> all x A(x))"))
(search)
```

The output of the `display-proof` function is:

```
> (dp)
; .......all x,y,z,w.A x & B x by assumption u13
; .......x
; ......all y,z,w.A x & B x by all elim
; ......y
; .....all z,w.A x & B x by all elim
```

```
; .....z
; ....all w.A x & B x by all elim
; ....w
; ...A x & B x by all elim
; ..A x by and elim left
; .all x A x by all intro
; (all x,y,z,w.A x & B x) -> all x A x by imp intro u13
```

The $\lambda - form$ of the proof is:

```
> (dpe)
(lambda (u13) (lambda (x) (car ((((u13 x) y) z) w))))
```

The intermediate list form is:

```
> (express-proof (input-proof))
(mult-intro
  "$(\\forall x,y,z,w.\\,A x \\land B x) \\to \\forall x\\, A x$"
  ((inline-all-intro "$A x$" (x))
   (inline-imp-intro
     "$\\forall x\\, A x$"
     "$\\forall x,y,z,w.\\,A x \\land B x$"
     u13))
  (and-keep-left
    "$A x$"
    (all-elim
      "$A x \\land B x$"
      (all-elim
        "$\\forall w.\\,A x \\land B x$"
        (all-elim
          "$\\forall z,w.\\,A x \\land B x$"
          (all-elim
            "$\\forall y,z,w.\\,A x \\land B x$"
            (avar "$\\forall x,y,z,w.\\,A x \\land B x$" u13)
            "x")
          "y")
        "z")
      "w")))
```

And the LaTeX rendering is:

Formula to be proved: $(\forall x, y, z, w.\, Ax \land Bx) \to \forall x\, Ax$

*Proof:* We can assume $\forall x, y, z, w.\, Ax \land Bx[1],\ x$. Then we have to show $Ax$. We instantiate assumption [1] with $x$, $y$, $z$, $w$ in order to obtain $Ax \land Bx$. Keeping only the left part of this we get $Ax$.

## A.5 Nested subproofs

This example shows two levels of subproofs and also conjunction introduction and elimination.

The Minlog commands (included in `and4.scm`) are:

```
(add-predconst-name "Q" "R" "S" (make-arity))
(set-goal (pf "(Q & R & S) -> (S & R & Q)"))
(search)
```

The output of the `display-proof` function is:

```
> (dp)
; ....Q & R & S by assumption u13
; ...S by and elim right
; .....Q & R & S by assumption u13
; ....Q & R by and elim left
; ...R by and elim right
; ..S & R by and intro
; ....Q & R & S by assumption u13
; ...Q & R by and elim left
; ..Q by and elim left
; .S & R & Q by and intro
; Q & R & S -> S & R & Q by imp intro u13
```

The $\lambda - form$ of the proof is:

```
> (dpe)
(lambda (u13)
  (cons (cons (cdr u13) (cdr (car u13))) (car (car u13))))
```

The intermediate list form is:

```
> (express-proof (input-proof))
(imp-intro
  "$Q \\land R \\land S \\to S \\land R \\land Q$"
  u13
  "$Q \\land R \\land S$"
  "$S \\land R \\land Q$"
  (and-intro
    "$S \\land R \\land Q$"
    (and-intro
      "$S \\land R$"
```

```
    (and-keep-right "$S$" (avar "$Q \\land R \\land S$" u13))
    (and-keep-right
      "$R$"
      (and-keep-left
        "$Q \\land R$"
        (avar "$Q \\land R \\land S$" u13))))
  (and-keep-left
    "$Q$"
    (and-keep-left
      "$Q \\land R$"
      (avar "$Q \\land R \\land S$" u13)))))
```

And the LaTeX rendering is:
Formula to be proved: $Q \land R \land S \to S \land R \land Q$
*Proof:* Let's assume $Q \land R \land S$[1].We now need to prove $S \land R \land Q$. We prove both parts of this.

- $\bullet_0$ We prove both parts of $S \land R$.
  - $\bullet_1$ Keeping only the right part of assumption [1] we get $S$.
  - $\bullet_1$ Keeping only the left part of assumption [1] we get $Q \land R$. Keeping only the right part of this we get $R$.
- $\bullet_0$ Keeping only the left part of assumption [1] we get $Q \land R$. Keeping only the left part of this we get $Q$.

## A.6 The Peirce formula

This example shows the additional printout of global assumptions which are used in the proof.

The Minlog commands (included in `peirce.scm`) are:

```
(add-predconst-name "A" "B" (make-arity))
(define peirce-formula (pf "((A -> B) -> A) -> A"))
(set-goal peirce-formula)
(assume 1)
(use "Stab-Log")
(assume 2)
(use 2)
(use 1)
(assume 3)
(use "Efq-Log")
```

```
(use 2)
(use 3)
```

The output of the `display-proof` function is:

```
> (dp)
; ..((A -> bot) -> bot) -> A by global assumption Stab-Log
; ....A -> bot by assumption u15
; .....(A -> B) -> A by assumption u13
; .......bot -> B by global assumption Efq-Log
; ........A -> bot by assumption u15
; ........A by assumption u18
; .......bot by imp elim
; ......B by imp elim
; .....A -> B by imp intro u18
; ....A by imp elim
; ...bot by imp elim
; ..(A -> bot) -> bot by imp intro u15
; .A by imp elim
; ((A -> B) -> A) -> A by imp intro u13
```

The $\lambda - form$ of the proof is:

```
> (dpe)
(lambda (u13)
  (|Stab-Log|
    (lambda (u15)
      (u15 (u13 (lambda (u18) (|Efq-Log| (u15 u18))))))))
```

The intermediate list form is:

```
> (express-proof (input-proof))
(imp-intro
  "$((A \\to B) \\to A) \\to A$"
  u13
  "$(A \\to B) \\to A$"
  "$A$"
  (imp-elim
    "$A$"
    (aconst
      "$((A \\to \\bot) \\to \\bot) \\to A$"
      "Stab-Log"
      "global-assumption")
```

```
"$((A \\to \\bot) \\to \\bot) \\to A$"
(imp-intro
  "$(A \\to \\bot) \\to \\bot$"
  u15
  "$A \\to \\bot$"
  "$\\bot$"
  (imp-elim
    "$\\bot$"
    (avar "$A \\to \\bot$" u15)
    "$A \\to \\bot$"
    (imp-elim
      "$A$"
      (avar "$(A \\to B) \\to A$" u13)
      "$(A \\to B) \\to A$"
      (imp-intro
        "$A \\to B$"
        u18
        "$A$"
        "$B$"
        (imp-elim
          "$B$"
          (aconst "$\\bot \\to B$" "Efq-Log" "global-assumption")
          "$\\bot \\to B$"
          (imp-elim
            "$\\bot$"
            (avar "$A \\to \\bot$" u15)
            "$A \\to \\bot$"
            (avar "$A$" u18)
            "$A$")
          "$\\bot$"))
      "$A \\to B$")
    "$A$"))
"$(A \\to \\bot) \\to \\bot$"))
```

And the LaTeX rendering is:

Formula to be proved: $((A \to B) \to A) \to A$

*Proof:* Let's assume $(A \to B) \to A$[1].We now need to prove $A$. We can derive this from global-assumption *Stab-Log*. It remains to show $(A \to \bot) \to \bot$. Let's assume $A \to \bot$[2].We now need to prove $\bot$. We can derive this from assumption [2]. It remains to show $A$. We can derive this from assumption [1]. It remains to show $A \to B$. Let's assume $A$[3].We now need to prove $B$.

We can derive this from global-assumption *Efq-Log*. It remains to show $\bot$. We can easily derive this from assumption [2] and assumption [3].

**Global assumption(s) used:**
**Efq-Log:** $\bot \to B$
**Stab-Log:** $((A \to \bot) \to \bot) \to A$

## A.7 Simple induction on natural numbers.

A simple example of induction. Term rewriting exists, but only for the base case.

The Minlog commands (included in `ind1.scm`) are:

```
(mload "../lib/nat.scm")
(set-goal (pf "all n,m.n + m = m + n"))
(assume "n")
(ind)
(normalize-goal)       ;abbreviation: ng
(use "Truth-Axiom")
(assume "m" "IH")
(ng)
(use "IH")
```

The output of the `display-proof` function is:

```
> (dp)
; ....allnc n.n+0=0+n -> (all n27.n+n27=n27+n -> n+Succ n27=Succ n27+n) -> all
n26 n+n26=n26+n by axiom Ind
; ....n
; ...n+0=0+n -> (all n27.n+n27=n27+n -> n+Succ n27=Succ n27+n) -> all n26
n+n26=n26+n by allnc elim
; ...T by axiom Truth-Axiom
; ..(all n27.n+n27=n27+n -> n+Succ n27=Succ n27+n) -> all n26 n+n26=n26+n by
imp elim
; ....n+m=m+n by assumption IH15
; ...n+m=m+n -> n+m=m+n by imp intro IH15
; ..all m.n+m=m+n -> n+m=m+n by all intro
; .all n26 n+n26=n26+n by imp elim
; all n,n26 n+n26=n26+n by all intro
```

The $\lambda - form$ of the proof is:

```
> (dpe)
(lambda (n)
  ((((|Ind| n) |Truth-Axiom|)
   (lambda (m) (lambda (|IH15|) |IH15|))))
```

The intermediate list form is:

```
> (express-proof (input-proof))
(ind "$\\forall m\\, n+m=m+n$"
     (aconst "$T$" "Truth-Axiom" "axiom")
     "$n+0=0+n$"
     "$T$"
     #t
     (mult-intro
       "$\\forall m.\\,n+m=m+n \\to n+m=m+n$"
       ((inline-imp-intro "$n+m=m+n$" "$n+m=m+n$" |IH15|)
        (inline-all-intro "$n+m=m+n \\to n+m=m+n$" (m)))
       (avar "$n+m=m+n$" |IH15|))
     "$\\forall n27.\\,n+n27=n27+n \\to n+Succ n27=Succ n27+n$"
     "$\\forall m.\\,n+m=m+n \\to n+m=m+n$"
     (aconst
       "$\\forall^{nc} n.\\,n+0=0+n \\to (\\forall n27.\\,n+n27=n27+n \\to
n+Succ n27=Succ n27+n) \\to \\forall n26\\, n+n26=n26+n$"
       "Ind"
       "axiom")
     "$(Pvar nat)^4 0 \\to (\\forall n27.\\,(Pvar nat)^4 n27 \\to (Pvar
nat)^4(Succ n27)) \\to \\forall n26\\, (Pvar nat)^4 n26$"
     "$n+0=0+n \\to (\\forall n27.\\,n+n27=n27+n \\to n+Succ n27=Succ n27+n)
\\to \\forall n26\\, n+n26=n26+n$")
```

And the LaTeX rendering is:

Formula to be proved: $\forall m\, n + m = m + n$

*Proof:* To prove $\forall m\, n + m = m + n$ we will use induction.
Base case: $n + 0 = 0 + n$.
We can prove the base case using axiom *Truth-Axiom* where $n + 0 = 0 + n$
$\searrow^* \swarrow T$.

Step: $\forall n27.\, n + n27 = n27 + n \to n + Succn27 = Succn27 + n$.
We can assume $m$, $n + m = m + n[1]$. Then we have to show $n + m = m + n$.
We can use assumption [1].

## A.8 Induction on lists

This example shows induction on lists. It proves that a list and its reverse have the same length. (Example by Dr.Monika Seisenberger)

The Minlog commands (included in `lists.scm`) are:

```
(mload "../lib/nat.scm")
(mload "../lib/list.scm")
(add-var-name "a" "b" (py "alpha"))
(add-var-name "s" "w" (py "list alpha"))
(add-program-constant "ListRev"
      (py "list alpha => list alpha") 1)
(add-token
  "Rev" 'prefix-op
  (lambda (x) (make-term-in-app-form
(make-term-in-const-form
  (let* ((const (pconst-name-to-pconst "ListRev"))
 (tvars (const-to-tvars const))
 (listtype (term-to-type x))
 (type (car (alg-form-to-types listtype)))
 (subst (make-substitution tvars (list type))))
    (const-substitute const subst #f)))
x)))
(add-display
  (py "list alpha")
  (lambda (x)
    (if (term-in-app-form? x)
      (let ((op (term-in-app-form-to-op x)))
(if (and (term-in-const-form? op)
 (string=? "ListRev"
   (const-to-name (term-in-const-form-to-const
    op))))
  (list 'prefix-op "Rev"
(term-to-token-tree (term-in-app-form-to-arg x)))
  #f))
      #f)))


(add-computation-rule (pt "Rev (Nil alpha)") (pt "(Nil alpha)"))
(add-computation-rule (pt "Rev (a::w)") (pt "(Rev  w) :+: (a:)"))
```

```
(set-goal (pf "all s. Lh s = Lh (Rev s)"))
(ind)
(ng)
(use "Truth-Axiom")
(assume "a" "s" "ih")
(ng)
(use "ih")
```

The output of the `display-proof` function is:

```
> (dp)
; ..Lh(Nil alpha)=Lh Rev(Nil alpha) -> (all a35,s36.Lh s36=Lh Rev s36 ->
Lh(a35::s36)=Lh Rev(a35::s36)) -> all s37 Lh s37=Lh Rev s37 by axiom Ind
; ..T by axiom Truth-Axiom
; .(all a35,s36.Lh s36=Lh Rev s36 -> Lh(a35::s36)=Lh Rev(a35::s36)) -> all s37
Lh s37=Lh Rev s37 by imp elim
; ....Lh s=Lh Rev s by assumption ih15
; ...Lh s=Lh Rev s -> Lh s=Lh Rev s by imp intro ih15
; ..all s.Lh s=Lh Rev s -> Lh s=Lh Rev s by all intro
; .all a,s.Lh s=Lh Rev s -> Lh s=Lh Rev s by all intro
; all s37 Lh s37=Lh Rev s37 by imp elim
```

The $\lambda - form$ of the proof is:

```
> (dpe)
((|Ind| |Truth-Axiom|)
 (lambda (a) (lambda (s) (lambda (ih15) ih15))))
```

The intermediate list form is:

```
> (express-proof (input-proof))
(ind "$\\forall s\\, Lh s=Lh Rev s$"
     (aconst "$T$" "Truth-Axiom" "axiom")
     "$Lh(Nil alpha)=Lh Rev(Nil alpha)$"
     "$T$"
     #t
     (mult-intro
       "$\\forall a,s.\\,Lh s=Lh Rev s \\to Lh s=Lh Rev s$"
       ((inline-imp-intro "$Lh s=Lh Rev s$" "$Lh s=Lh Rev s$" ih15)
        (inline-all-intro "$Lh s=Lh Rev s \\to Lh s=Lh Rev s$" (s))
        (inline-all-intro
```

```
          "$\\forall s.\\,Lh s=Lh Rev s \\to Lh s=Lh Rev s$"
          (a)))
       (avar "$Lh s=Lh Rev s$" ih15))
     "$\\forall a38,s39.\\,Lh s39=Lh Rev s39 \\to Lh(a38::s39)=Lh
Rev(a38::s39)$"
     "$\\forall a,s.\\,Lh s=Lh Rev s \\to Lh s=Lh Rev s$"
     (aconst
       "$Lh(Nil alpha)=Lh Rev(Nil alpha) \\to (\\forall a35,s36.\\,Lh s36=Lh Rev
       "Ind"
       "axiom")
     "$(Pvar list alpha4)^4(Nil alpha4) \\to (\\forall (alpha4)_34,(list
alpha4)_33.\\,(Pvar list alpha4)^4(list alpha4)_33 \\to (Pvar list
alpha4)^4((alpha4)_34::(list alpha4)_33)) \\to \\forall (list alpha4)_32\\,
(Pvar list alpha4)^4(list alpha4)_32$"
     "$Lh(Nil alpha)=Lh Rev(Nil alpha) \\to (\\forall a38,s39.\\,Lh s39=Lh Rev
s39 \\to Lh(a38::s39)=Lh Rev(a38::s39)) \\to \\forall s40\\, Lh s40=Lh Rev
s40$")
```

And the LaTeX rendering is:

Formula to be proved: $\forall s\, Lhs = LhRevs$

*Proof:* To prove $\forall s\, Lhs = LhRevs$ we will use induction.

Base case: $Lh(Nilalpha) = LhRev(Nilalpha)$.

We can prove the base case using axiom *Truth-Axiom* where $Lh(Nilalpha) = LhRev(Nilalpha) \searrow^* \swarrow T$.

Step: $\forall a41, s42.\, Lhs42 = LhRevs42 \rightarrow Lh(a41 :: s42) = LhRev(a41 :: s42)$.

We can assume $a$, $s$, $Lhs = LhRevs[1]$. Then we have to show $Lhs = LhRevs$. We can use assumption [1].

# Bibliography


[ACP00] Abel A., Chang B.E., Pfenning F., *Human-Readable Machine-Verifiable Proofs for Teaching Constructive Logic*, Computer Science Department, Carnegie Mellon University, 2000.

[AGDA] Chalmers University of Technology, *The proof system Agda*, `http://www.cs.chalmers.se/~catarina/agda/`

[AR82] Adams N.I., Rees J., *T: a dialect of Lisp or LAMBDA: The ultimate software tool*, ACM conference proceedings on Lisp and functional programming, 1982, pp: 114-122.

[AR88] Adams N.I., Rees J. *Object-oriented programming in Scheme*, ACM conference proceedings on Lisp and functional programming, 1988, pp:277-288

[BAC78] Backus J., *Can programming be liberated from the Von Neumann style?*, ACM Communications, Volume 21, Issue 8, 1978.

[BBSSZ98] Benl H., Berger U., Schwichtenberg H., Seisenberger M., Zuber W., *Proof theory at work: Program development in the Minlog system*, Automated Deduction, W. Bibel and P.H. Schmitt, eds., volume 2, Kluwer 1998

[BJ86] Bartley D.H., Jensen J.C., *The implementation of PC Scheme*, ACM conference proceedings on Lisp and functional programming, 1986, pp: 86-93.

[BRO87] Brooks Frederick P., *No Silver Bullet: Essence and Accidents of Software Engineering*, Computer, volume 20, issue 4, April 1987, pp: 10-19.

[BU01] Berger Ulrich, *Programming with abstract data types lecture notes*, Computer Science Department, University of Wales Swansea, 2001.

[CHEZ6] Cadence Research Systems, *Chez Scheme*, `http://www.scheme.com/`

[CHU41] Church A., *The calculi of Lambda Conversion*, Princeton University Press, Annals of mathematics studies, Volume 6, 1941.

[COQ] The LogiCal group, *Coq formal proof management system*, `http://pauillac.inria.fr/coq/`

[CRO4] Crosilla L., *A tutorial for Minlog, version 4.0*, `www.mathematik.uni-muenchen.de/~minlog/minlog/tutor.ps`

[EA00] Eliens A., *Principles of Object-Oriented Software Development (second edition)*, Addison Wesley, ISBN: 0-201-39856-7, 2000, pp: 239-276.

[FSHARP] Microsoft Research, *F# programming language*, `http://research.microsoft.com/projects/ilx/fsharp.aspx`

[GIMP] GNU Image Manipulation program, *Script-Fu and plug-ins for The GIMP*, `http://www.gimp.org/docs/scheme_plugin/`

[HAN90] Hanson Chris, *Efficient stack allocation for tail-recursive languages*, ACM conference proceedings on Lisp and functional programming, France, June 1990, pp: 106-118.

[HBC99] Holland-Minkley A.M., Barzilay R., Constable R.L., *Verbalization of High-Level Formal Proofs*, Department of Computer Science, Cornell University, 1999.

[HUD89] Hudak P., *Conception, evolution and application of functional programming languages*, ACM Computing Surveys, volume 21, issue 3, September 1989, pp: 359-411.

[HUG89] Hughes J., *Why functional programming matters*, Computer Journal, Volume 32, Issue 2, 1989.

[ISA] University of Cambridge, *Isabelle generic theorem proving environment*, `www.cl.cam.ac.uk/Research/HVG/Isabelle/`

[KRA86] Kranz David et al, *Orbit: an optimising compiler for Scheme*, ACM SIGPLAN proceedings on Compiler construction, volume 21, issue 7, July 1986, pp: 219-213.

[LOGIK] University of Munich, *The logic group homepage*, `http://www.mathematik.uni-muenchen.de/~logik/`

[MAEHL62] McCarthy J., Abrahams P.W., Edwards D.J., Hart T.P., Levin M.I., *Lisp 1.5 programmer's manual*, M.I.T. press, 1962.

[MINLOG] Schwichtenberg Helmut et al., *The Minlog system*, `http://www.minlog-system.de`

[MN02] Meyer A., Nagpal R., *Mathematics for Computer Science lecture notes*, Massachusetts Institute of Technology, 2002.

[MREF] Schwichtenberg Helmut et al., *Minlog Reference manual*, `www.mathematik.uni-muenchen.de/~minlog/minlog/ref.ps`

[NUPRL] Cornell University, *The PRL Project*, `www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`

[RAM94] Ramsdell John D., *Scheme: The Next Generation*, Lisp Pointers, volume 7, issue 4, 1994, pp: 13-14.

[RD92] Rees J., Donald B., *Program Mobile robots in Scheme*, IEEE international conference proceedings on Robotics and Automation, 1992, pp: 2681-2688.

[RRS5] Adams N.I. et al, *Revised*$^5$ *report on the algorithmic language Scheme*, ACM SIGPLAN Notices, volume 33, issue 9, September 1998.

[RSST96] Robertson N., Sanders D.P., Seymour P., Thomas R., *A new proof of the four-colour theorem*, Electronic research announcements of the American Mathematical Society, volume 2, issue 1, August 1996.

[SCH91] IEEE Standards Board, *IEEE Std 1178-1990 IEEE Standard for the Scheme Programming Language*, Institute of Electrical and Electronic Engineers Inc., New York, 1991.

[SCHW99] Schwichtenberg H., *Proof Theory lecture notes*, Mathematisches Institut der Universitaet Muenchen, sommersemester 1999.

[SF89] Springer G., Friedman D. P., *Scheme and the art of programming*, M.I.T. Press, ISBN: 0-262-19288-8, 1989, Foreword by G. L. Steele Jr.

[SG93] Steele G.L. Jr., Gabriel R. P., *The Evolution of LISP*, ACM SIGPLAN Notices, volume 28, issue 3, March 1993, pp: 231-270.

[STE77] Steele G.L. Jr., *Debunking the expensive procedure call myth or, procedure call implementations considered harmful or, LAMBDA: The Ultimate GOTO*, ACM conference proceedings, Seattle, October 1977, pp: 152-162.

[STL00] St.Laurent S., *XML Elements of style*, McGraw-Hill, ISBN: 0-07-212220-X, 2000.

[XIM] Ximian(now a part of Novell), *The Evolution suite*, `http://www.ximian.com/products/evolution/`