Formula Editors for TP-based Systems.

State of the Art and Prototype Implementation in ISAC

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Marco Mahringer, B.Sc.

Betreuer: Dr. Technr. Walther Neuper, IIS Linz
Begutachter: FH-Prof. PD DI Dr. Stephan Dreiseitl

Dezember 2017
Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, December 21, 2017

Marco Mahringer, B.Sc.
## Contents

<table>
<thead>
<tr>
<th>Declaration</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kurzfassung</td>
<td>vi</td>
</tr>
<tr>
<td>Abstract</td>
<td>vii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 Why yet another editor?</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Why so diverse preliminaries?</td>
<td>2</td>
</tr>
<tr>
<td>1.3 The structure of the thesis</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Preliminaries</strong></td>
<td></td>
</tr>
<tr>
<td>2.1 Accessibility and Inclusion</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Standards in Formula Presentation</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 Survey on Available Types of Editors</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 MathML</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3 \LaTeX Technology</td>
<td>8</td>
</tr>
<tr>
<td>2.2.4 Standard Font</td>
<td>9</td>
</tr>
<tr>
<td>2.2.5 Spacing</td>
<td>10</td>
</tr>
<tr>
<td>2.2.6 Character Size in Sub-Terms</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Front-ends of Theorem Provers (TPs)</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1 State and Future of TP Front-ends</td>
<td>11</td>
</tr>
<tr>
<td>2.3.2 Isabelle’s Graphical User Interface</td>
<td>12</td>
</tr>
<tr>
<td>2.3.3 Isabelle’s Term Presentation</td>
<td>13</td>
</tr>
<tr>
<td>2.4 The Front-end of the \text{\textsc{IsAcC}}-Prototype</td>
<td>17</td>
</tr>
<tr>
<td>2.5 Requirements for the Editor Prototype</td>
<td>20</td>
</tr>
<tr>
<td>2.5.1 Downcut Decisions for Prototyping</td>
<td>20</td>
</tr>
<tr>
<td>2.5.2 User Requirements</td>
<td>21</td>
</tr>
<tr>
<td>2.5.3 Software Requirements and Scala</td>
<td>21</td>
</tr>
<tr>
<td><strong>3 Design Considerations for TP-based Editors</strong></td>
<td></td>
</tr>
<tr>
<td>3.1 Re-use of Existing Components</td>
<td>26</td>
</tr>
<tr>
<td>3.1.1 Adapt Isabelle’s AST Transformations</td>
<td>27</td>
</tr>
<tr>
<td>3.1.2 Integration into the \text{\textsc{IsAcC}} Architecture</td>
<td>28</td>
</tr>
</tbody>
</table>
3.2 Integration into the $\mathcal{ISAC}$-Prototype ............................ 29
  3.2.1 Formulas in Calculations ........................................ 29
  3.2.2 Formula Representation ............................................ 29
3.3 Interface for a Formula Editor ........................................ 30
  3.3.1 The Java interface .................................................. 30
  3.3.2 Use Cases ............................................................ 30
3.4 Datatypes ............................................................... 34
  3.4.1 Term, the mathematical Object .................................... 35
  3.4.2 Annotated Syntax Tree .............................................. 36
  3.4.3 Specific Decisions on ASTs ....................................... 38
3.5 Layout Classes for Sub-Terms ........................................ 40

4 Implementation of a Prototype Editor ................................. 42
  4.1 Classes and Coding Decisions ....................................... 42
    4.1.1 Code Structure of the Editor .................................. 42
    4.1.2 Integration into $\mathcal{ISAC}$ ................................. 44
    4.1.3 Java – Scala – Java ............................................. 47
  4.2 Symbols and Expressions in Java Swing ............................ 48
    4.2.1 Single Symbols ................................................... 49
    4.2.2 Expressions without Vertical Alignment ....................... 50
    4.2.3 Expressions with Vertical Alignment ........................... 51
  4.3 Dynamic Aspects ...................................................... 52
    4.3.1 Interactions between Components .............................. 52
    4.3.2 Mouse Events on the EditorPanel .............................. 53
    4.3.3 Interactions during Editing .................................... 56
  4.4 Test Environments .................................................... 60
    4.4.1 Test Environment for a Single Formula ....................... 60
    4.4.2 Test Environment for Working on a Calculation ............... 61

5 Summary, Conclusion and Future Work ............................... 62
  5.1 Summary .............................................................. 62
  5.2 Conclusions .......................................................... 68
  5.3 Preview to Future Work .............................................. 69

References ............................................................... 72
  Literature ........................................................................ 72
  Online sources ........................................................... 74
Kurzfassung

Diese Masterarbeit behandelt Neuland in drei verschiedene Richtungen. Erstens in die Richtung von zweidimensionaler Formeldarstellung und Visualisierung für Systeme basierend auf (Computer) Theorem Proving (nachfolgend abgekürzt als ’TP’). Zweitens beschäftigt sie sich mit ’accessible’ und ’inclusive’ Systemen. Die dritte Richtung beschäftigt sich mit der Integration von Scala mit Java Swing im 𝒆 уме-Prototypen.

Die erste Richtung wird dadurch relevant, dass TP-Systeme immer mehr in die Arbeitsumgebung von Ingenieuren eindringen. Während Ingenieure mathematische Formeln in der bekannten zweidimensionalen Darstellung erwarten, wurde TP von (meist) akademischen Experten für Software-Verifikationen verwendet, wo gut lesbare Formeln kein Thema sind.

Die zweite Richtung, also der ’accessible’- und ’inclusive’-Bereich, wurde durch eine Diplomarbeit an der FH Hagenberg, Fakultät für Softwaredirektik, im vergangenen Jahr vorbereitet. Die vorliegende Masterarbeit stellt eine elegante Lösung vor, die die bevorzugte Formelpräsentation für blinde sowie sehbehinderte Studierende aufrechterhält: Teil-Formeln auf der Braille (für blinde Studierende) werden eindeutig mit Rechtecken auf der zweidimensional gerenderten Formel (für sehende Studierende) hervorgehoben.


Die Masterarbeit lässt den Schluss zu, dass in allen drei Richtungen ’the proof of concept’ erfolgreich war. Design und Implementierung sind solide grundgelegt worden und die dabei gewonnenen Erfahrungen in der Arbeit dokumentiert. Weitere Entwicklung auf Basis dieser Arbeit erscheint vielversprechend.
Abstract

This thesis breaks new grounds in three different directions, first towards two-dimensional formula editors for systems based on (computer) theorem proving (abbreviated “TP” in the title), second towards accessible and inclusive editors and third towards combining Scala with Java Swing in building a respective prototype in \textsc{Isabelle}.

The relevance of the first direction is given by ongoing intrusion of TP into engineering practice: While engineers expect mathematical formulas in the well known two-dimensional representation, TP has (mostly) been used by academic experts for software verification, where nicely readable formulas are not an issue. Challenges arise when adopting specific requirements of TP experts, for instance, getting types of elements of a formula and respective definitions by mouse click (as known for program codes in IDEs).

The second direction towards accessible and inclusive editors has been prepared by a thesis at FH Hagenberg, in the preceding year. Taking the best from the preparations, this thesis proposes an elegant solution, which maintains the preferred formula presentation for both for blind and for sighted students: sub-terms on the braille (for the blind student) are unambiguously related to highlighted rectangles on the two-dimensionally rendered formula (for the sighted student).

The third direction, towards combining Scala with Java Swing in code for a formula editor and integration into \textsc{Isabelle}, took the major time and energy in this thesis. The implementation found an efficient and elegant combination between the two programming languages. All principles from \LaTeX{} are implemented in the prototype: decreasing font size according to levels of sub-terms, respective adaption of spaces, etc. The prototype also allows input of formulas from scratch as well as editing existing formulas.

The thesis allows the conclusion, that in all three directions the proof of concepts was successful. Solid foundations in design and in implementation have been provided, respective experiences are documented in the thesis. Further development based on these foundations appears promising.
Chapter 1

Introduction

The task to be accomplished by this thesis is in short: Research the state of the art of formula editors, design an accessible and inclusive version, provide a prototype implementation and integrate it into the ISAC-prototype, a new kind of educational tool.

The introduction has two main purposes: firstly it needs to explain the reasons for designing and implementing yet another formula editor and secondly it needs to explain, why there are so different preliminaries to accomplish before the very task of the thesis begins.

1.1 Why yet another editor?

This thesis prefers the naming “formula editor” for a software component rendering mathematical formulas in the usual two-dimensional representation and supporting input of such formulas. The naming “equation editor” seems preferred in American English, but is considered inappropriate here, because an equation is just a specific formula of type boolean.

Now the answer to the question is that...

there are novel user requirements for formula editors, introduced by the novelty of ISAC’s prototype.

The novelty of ISAC’s prototype is introduced by technology underlying the respective mathematics engine, technology from (computer) theorem proving (abbreviated TP in the sequel). In principle, TP is self-contained: all elements and all statements are derived from simpler elements and statements, ultimately by the few basic laws of formal logic. These derivations...
are implemented by “interactive proof assistants” in (more or less) traditional mathematical notation (predicate logic) — and thus can be read by humans without further translation.

So the technology triggers new kinds of interaction within formula editors: now a mathematical formula becomes the entry point for all underlying operators and types in the same way as source elements of a computer program are entry points for all underlying definitions, just by click on elements in an IDE. In the \( \mathcal{ISAC} \) project this feature is part of the concept “transparent systems”.

\( \mathcal{ISAC} \)’s mathematics engine is based on such TP technology and exploits respective power to support “dialogues between partners on an equal base”: both partners, the student and the system can negotiate parts of formulas, the system can give hints for subterms, etc. All the technical prerequisites have been identified in the \( \mathcal{ISAC} \)-project for building “systems that explain themselves” \cite{13,14}: not only underlying definitions become interactively available, but also intermediate steps of problem solving processes in engineering mathematics — a trigger for novel requirements also on formula editors.

Last not least, there are the requirements of accessibility and of inclusive learning prepared by another thesis \cite{8}. These two requirements are met unsatisfactorily by available formula editors. Work on these requirements in the thesis is supervised by the “Institut Integriert Studieren” at Johannes Kepler University in Linz, Austria.

1.2 Why so diverse preliminaries?

The answer to the first question above and respective explanations indicate a wider spectrum of challenges than anticipated by the title of the thesis.

Decision between Java Swing and Browsers was still open at the beginning of the project: \( \mathcal{ISAC} \) plans to make \( \mathcal{ISAC} \) as accessible for students as much as possible in the future. Depending on preliminary research within this thesis, the envisaged prototype editor should be implemented either in Java Swing or within browser technologies. So, both possibilities need to be investigated.

Not reinventing the wheel! This is the main challenge when approaching development within formula editors, where an uncountable number of implementations are already done.

As a matured technology, formula editors are backed with well-established theory and a wealth of respective publication. These have to be studied, in particular MathML (Mathematical Markup Language) and \LaTeX, which set the high standards for formula representation many decades ago.
Also a review of existing formula editors needs to clarify, how close available products are to the user requirements set for the thesis. The search can be confined to open source products, because the ZS4C prototype is itself open source and consists of (many!) subsystems, which are all open source, too. The sources have to be studied in order to estimate the effort required to adapt to the given user requirements. This introduction can presage, that this part of preliminary work was short: for all studied products the efforts were estimated very high.

**Isabelle is a trendsetter for GUIs in TP,** because it already implements much of “transparent systems” mentioned above, in particular IDE-like access to definitions via mouse click on elements of formulas.

Isabelle needs to be studied in order to prepare for integration the prototype editor into the ZS4C prototype, anyway. The ZS4C prototype is adopting Isabelle’s technologies more and more with the long term goal to contribute as Isabelle component. With respect to formula editors, fortunately, Isabelle implements respective technology in a standard way. This machinery needs to be studied and investigated for re-use by the editor envisaged in this thesis.

There is also the issue, that ZS4C presently uses a specific interface to Isabelle, not the standard interface. The specific interface transports terms, which contain types, sorts and all the other stuff required for reliable reasoning. Since ZS4C’s design delegates all reasoning to Isabelle, it is questionable, whether these terms are the right data structure for our formula editor, which serves ZS4C’s front-end (which is not concerned with any kind of reason). Thus these terms and respective transformations also will be studied.

**ZS4C’s comprehensible code base challenges integration** of the formula editor envisaged in this thesis. The code base results from research and development since more than a decade. The mathematics engine had been developed before the front-end was designed. The architecture of the latter is highly abstract — part of the success for integration of about thirty student projects (at the level of diploma and master theses) over the years without changes to the initial architecture.

According to the comprehensive requirements captured by the initial design and the many contributions, the code base is large and complex. The study of relevant architectural considerations and respective portions of code
will be documented in the thesis in order to justify the design decisions for the editor.

**A hint for reading** the thesis seems appropriate with respect to the diversity of preliminaries: The final decisions during implementation (as described in chapter 4 near the bottom of the thesis) require references back to respective design decisions in the preceding chapter, and further on, the latter require references back to respective requirements stated previously — and all those references end up in the preliminaries stated at the beginning in chapter 2. It is advised to take the many references into consideration in order to grasp the whole picture!

### 1.3 The structure of the thesis

The structure is as follows: Chapter 2 collects all the different prerequisites for the thesis: the high standards in formula presentation in section 2.2 are given by section 2.2.1 various types of editors, by section 2.2.2 MathML and last not least by section 2.2.3 LATEX. Section 2.3 briefly introduces the state of front-ends for (computer) theorem provers (TP) with focus on Isabelle: section 2.3.1 explains reasons for the transitional state of TP, section 2.3.2 presents Isabelle’s GUI and looks into the machinery as far as relevant. The last prerequisite is the \( \mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C} \) prototype in section 2.4 it’s initial state of the front-end and requirements for the formula editor are given in section 2.5.

Chapter 3 comes to architectural and design considerations, which start with the issue of re-use in section 3.1: re-use of Isabelle’s machinery for formula transformation in section 3.3.3 and integration into \( \mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C} \)’s existing front-end in section 3.1.2. Integration is discussed more specifically in a separate section 3.2 and conforms to an interface specified in section 3.3. Concern of design are also datatypes specific for editors and general considerations about visual representation of formulas in section 3.5.

Chapter 4 describes the implementation of the editor prototype. Section 4.1 addresses code structure, integration and choice of programming language. Section 4.2 explains how symbols are drawn by Java Swing, section 4.2 shows how a formula is rendered and section 4.3 describes interaction within editing a formula. Implementation required specific test setups addressed in section 4.4.

Finally, in chapter 5 a summary accounts for what has been achieved by this thesis, conclusions reflect the appropriateness of design and implementation and future work tells how to proceed within prototype development.
Chapter 2

Preliminaries

The above section 1.2 mentioned reasons for the large number of preliminaries. The details are given here, which are required as prerequisites for detailed design and implementation.

2.1 Accessibility and Inclusion

The requirements of accessibility and of inclusive learning have been prepared by another thesis [8]. “Accessibility” means, that formulas must be accessible by all humans interested in mathematics, also by visually impaired people. The latter cannot use a mouse, which is an indispensable device for input to a GUI at the present state of the art. How such GUIs have been made accessible, nevertheless, this explains [8] but is out of scope of the thesis at hand.

Visually impaired people can also not benefit from the two-dimensional formula presentation on paper and on screen as rendered by formula editors at the state of the art — so how help those people to understand formulas without a mouse and without the two-dimensional presentation, without respective expressiveness and conciseness, which results from hundreds of years in mathematical experience?

And “Inclusion” means, that blind students are not excluded from mainstream education and sit in one class together with peers. Inclusion leads to educational settings, were a blind student and another (“normal”) student want to cooperate in solving mathematical problems. And this might involve the situation, where both discuss properties of one and the same formula on a screen (or even on two different screens); for instance they might discuss, whether \( \frac{1+2(3+x)}{2(3+x)} \) can be canceled or not — and then they need to collaboratively refer to the numerator and the nominator, the arguments of multiplication, etc — How can mutual pointing happen, when the one prefers to point with the mouse to a two-dimensional representation and the other
only can use keys and read strings on the braille?

A surprisingly straightforward solution has been prepared by [8], dropping respective experiments with auditive information and specific representations of formulas as trees:

- Leave the sighted user with all habits in manipulating formulas and just support pointing to a part of a formula by highlighting the respective sub-term.
- Support navigation through a formula by specific keys and present the respective sub-terms as strings on the braille (with leading blanks indicating the depth of the sub-term) — and highlight the sub-term accordingly in the two-dimensional representation for the sighted user.

This solution includes even editing of a formula in collaboration between a sighted and a visually impaired person!

The only challenge left seems to be to dissolve the meaning of keys within different Swing components such that a blind person can use the keyboard as the only input device for a sophisticated mathematics tool like ZS4C.

2.2 Standards in Formula Presentation

Formulas are at the heart of technical communication, of engineering tools and of technical and scientific publications. The two standards for representing formulas, LATEX and MathML, are briefly introduced below. Then layout principles are discussed with respect to these two standards: mathematical fonts, spacing within formulas and character size within sub-terms.

2.2.1 Survey on Available Types of Editors

There are commercial formula editors like MathType [1] MathFlow [2] TechExplorer [3] or Wiris [11]. Being state-of-the-art, such editors are “what you see is what you get (WYSIWYG)”, i.e. they try to resemble common notation for formulas in two dimensions. “Common notation” is a challenging requirement, since there are differences in notation between pure mathematics and engineering mathematics, even between different engineering disciplines (e.g. electrical engineers prefer $j$ for the imaginary unit, others prefer $i$).

All these commercial editors cannot be integrated into ZS4C, because ZS4C’s front-end is not standard like a browser. So these editors fall outside the selection criteria of this thesis.

Open source editors can be adapted and integrated with more or less efforts. However, there are comparably few editors of the latter kind. The most prominent among these are MathJax\(^4\) or MathCast\(^5\). Several others arose from singular efforts and are not maintained reliably \(^22\).

But all these editors do not meet many of ISAC’s user requirements (introduced later one); so these fall out of scope of this thesis as well. However, there is much to learn from all these products, the most relevant points are addressed subsequently.

The internal structure of formulas is different for the two standards \LaTeX\ and MathML. The former is the eldest standard for mathematical formulas and still used for scientific papers and books. The latter has been introduced for use in internet browsers. The former presentation is merely character strings, the latter reflects more mathematical structure \(^18\).

Input of formulas is a separate challenge for editors. All of them provide two-dimensional representation of formulas, but few of them allow input directly within the two-dimensional representation. A frequent detour is input of formulas as strings, which afterwards are rendered in two dimensions. Such a detour is out of ISAC’s user requirements introduced below, and thus out of scope of this thesis.

Further requirements for ISAC’s formula editor are concerned with accessibility for visually impaired persons. These concerns have been clarified by a preceding master thesis \(^8\) and the thesis at hand builds on that work.

Most intuitive input of formulas is promised by handwriting recognition; however, most respective products are proprietary software, not open source and thus out of scope of this thesis. Few experiments in open source are not yet sufficiently serious for being envisaged by this thesis.

2.2.2 MathML

MathML is a standard issued by the World Wide Web Consortium (W3C), now available in version three \(^29\) within HTML5. MathML is a dialect of XML, thus formulas encoded in MathML can easily be embedded in XML documents and rendered in internet browsers.

MathML models the structure of a formula using specific XML tags. The listing on p.8 shows how identifiers are modeled by tag \(<\text{mi}>\), operators by tag \(<\text{mo}>\) and numerals by tag \(<\text{no}>\):

\[ y = x^{2k+1} \]

Figure 2.1: Example MathML

Another example are fractions. Since XML trees are a recursive data structure: a fraction can contain another fractions, etc.

The MathML shown in the listing on p.8 is “presentation format”, a format close to the structure of the rendered formula, also called “concrete syntax”. MathML defines a “content format”, too, for capturing the mathematical structure of a formula, the “abstract syntax”. The latter turned out irrelevant for this thesis and is thus not discussed further.

2.2.3 \( \LaTeX \) Technology

Modern standards of typesetting have been established by an academic non-commercial effort started by the mathematician and computer scientist Donald Knuth in 1976. He was disappointed by the poor standards of newly developed electronic publishing tools at that time. So he created one of his own and named it TeX. As a mathematician Knuth was particularly interested in high quality of formula presentation, so this has been integrated in TeX.

In contrary to MathML, in \( \LaTeX \) the input is a string; the specific syntax is intuitive even for complicated formulas and known by heart by most mathematicians:
This string needs to be parsed. The resulting parse tree remains invisible for the user. The formula defined by the string above is rendered this way:

\[ y = x^{2k+1} \]

This simple example already indicates the many intricacies involved in high-quality rendering of a mathematical formula: specific fonts help to distinguish letters, see for instance the strange \( x \); further details will be discussed in section 2.2.4 below. Then spacing between elements, just note the different spaces left and right of the equal sign = and compare it with the smaller spaces left and right of +; further details will be discussed in section 2.2.5 below. And last not least, the size of characters in the superscript \( 2k + 1 \) is decreased in a certain amount as compared to the basis; further details will be discussed in section 2.2.6 below.

All that details are determined automatically from a simple string by \LaTeX, so an author using this system can focus the content of his work and is not distracted by issues of layout.

2.2.4 Standard Font

Donald E. Knuth even took care of fonts, was the first person to define the shapes of characters by mathematical means, and developed a specific font, AMS Euler Font \[9\], in collaboration with Hermann Zapf from the American Mathematical Society (AMS). This font resembles handwriting of a mathematician. The book *Concrete Mathematics* \[5\] was the first book written using this font after integration into TeX.

Further notable fonts are MathML font and STIX font. Both are used by MathML and other typesetting systems.

Fig.2.2 shows a very simple formula in order to introduce two important notions for mathematical fonts, baseline and midline. The former is the line where letters like \( a, b \) or \( c \) are based on, but not descenders like \( f, g \) or \( j \). The latter is placed at 2/3 of the height and has a specific purpose. Mathematical symbols like +, \cdot etc are not placed on the baseline but on the midline (note that + goes beyond the baseline). This leads to nicely readable formulas, where + is at the same height as a fraction bar, for instance \( X + \frac{X}{X} \).

![Figure 2.2](image-url)
2. Preliminaries

2.2.5 Spacing

As mentioned, \LaTeX\ creates rendering of formulas from strings and determines spaces between elements of a formula automatically. In case automatic spacing appears not optimal, additional spacing information can be given by \texttt{\textbackslash}, \texttt{\textbackslash{}}, and \texttt{\textbackslash{}}; and other means.

MathML, however, makes spacing explicit by distinguishing specific tags for

- operations
- identifiers
- numbers
- relations

These tags determine spacing specifically for various combinations of elements in a formula: there can be a

- large space (LS)
- medium space (MS)
- small space (SS)

where the abbreviations in parentheses refer to Fig.2.3. Here a table assembles all combinations and the resulting spaces. The formula $x = a + b$

<table>
<thead>
<tr>
<th>Operator</th>
<th>Identifier</th>
<th>Number</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>MS</td>
<td>MS</td>
<td>MS</td>
</tr>
<tr>
<td>Identifier</td>
<td>MS</td>
<td>SS</td>
<td>SS</td>
</tr>
<tr>
<td>Number</td>
<td>MS</td>
<td>SS</td>
<td>SS</td>
</tr>
<tr>
<td>Relation</td>
<td>LS</td>
<td>LS</td>
<td>LS</td>
</tr>
</tbody>
</table>

Figure 2.3: Spaces between elements of a formula.

2.2.6 Character Size in Sub-Terms

The assignment of different sizes to different parts of one and the same formula has first been formalized in \LaTeX. There each element of a formula is contained in a box with specific width and height. So a formula becomes a tree of boxes, where boxes contain other boxes and so determine higher levels of nesting. Specific operators like fraction and exponentiation are assigned specific vertical arrangements, where the level of boxes is increased by one at a time. Fig.2.5 shows an example.

Each level reduces the size of embraced boxes. \LaTeX\ reduces box sizes for such sub-terms by 66,6%, but only three levels down and then remains
2. Preliminaries

Figure 2.4: Spaces created for string “x = a + b”.

Figure 2.5: Levels of boxes in \( \LaTeX \)

constant. MathML, in contrary, remains constant already down from level two. In both cases sub-terms remain readable in case of dynamic size change, see Fig. 2.6 on p. 11.

Figure 2.6: Space is reduced by \( \frac{2}{3} \) for each level, maxlevel = 2.

2.3 Front-ends of Theorem Provers (TPs)

This thesis uses the abbreviate “TP” for both for the academic discipline of automated theorem proving (ATP) and interactive theorem proving (ITP)
as well as for respective software systems. TP is an emerging technology, so neither basic namings are settled (like “TP”) nor user interfaces (while the logical foundations are about hundred years old).

2.3.1 State and Future of TP Front-ends

State of the art TPs combine ATP and ITP, like Isabelle [16], and are frequently called generic proof assistant. The word “generic” points at the fact, that even the basic logic can be determined within the system (if derivable from “natural deduction” [4]). In Isabelle “Higher Order Logic” is most elaborated and abbreviated Isabelle/HOL.

At present TPs appear not targeted to widespread use of SW-engineers (or of mathematicians), rather, TPs are concern of small groups of experts all around the world. However, the user group is rapidly growing — mechanized software verification is underway to engineering practice (and more slowly to mathematicians’ everyday practice).

The first front-end for TPs was the Emacs editor [21], for centuries the editor for programmers, and early adapted to TPs. Most power users of TP still prefer the Emacs interface to other interfaces. However, Emacs technology becomes outdated with the advent of parallel proof checking and with widespread usage in engineering. Another issue is integrating collaboration and version management with TPs.

The TP in the focus of the thesis, Isabelle, has the most advanced front-end. It already shows part of what is called “transparent system” in the ℐ𝒮𝒜ℂ project: all mathematical knowledge underlying any interaction in problem solving is directly accessible (and readable by humans). Fig.2.7 shows what knowledge Isabelle shows by mouse click: the axioms of a semigroup by click on +, the definition of integers as equivalence relation over natural numbers by click on int (where one also finds the proof, that integers form a semiring), a click on distrib_left the theorem of distributivity together with
2. Preliminaries

the proof, that this theorem holds for semigroups (and thus for semirings and thus for integers), etc.

2.3.2 Isabelle’s Graphical User Interface

The TP Isabelle \[16\] already exploits multi-core hardware and prepares for widespread usage in the practice of software engineers and mathematicians. Isabelle’s front-end has a specific interface, PIDE (prover IDE — integrated development environment) \[24\]. This interface is capable of parallel proof checking \[25\], linking editor content with definitions, etc. The main application of PIDE uses jEdit \[26\] as front-end (besides others under development for Eclipse and for standard browsers).

Here only those aspects of Isabelle/jEdit are presented, which are relevant for the thesis. Isabelle’s front-end used for software verification looks as shown in Fig.2.8 on p.13. Program code is a normal Isabelle term. For programming line-oriented presentation is appropriate. Isabelle uses line-oriented presentation also for mathematical formulas as shown in Fig.2.9 on p.14. This kind of presentation is still considered appropriate for TP. However, for widespread practice engineers will request the formula from Fig.2.9 presented as

\[
\sum_{i=0}^{n} i^3 = \left(\frac{n \cdot (n + 1)}{2}\right)^2
\]

Figure 2.8: Programs in Isabelle’s verification environment.
according to standards established by \LaTeX. Isabelle/\ISAC aims at widespread

\begin{verbatim}
\textbf{theorem} sum_of_cubes:
"4 * (\sum i::nat=0..n. i^3) = (\sum n * (n + 1))^2"
(is "(?P n" is "(?S n = _")
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{isabelle_formula}
\caption{Isabelle’s line-oriented formula presentation.}
\end{figure}

use and thus must go beyond Isabelle/jEdit in quality of formula presentation.

2.3.3 Isabelle’s Term Presentation

While Isabelle’s term representation appears simple, the respective internal machinery is standard technology, analogous to parsing of programming languages. The process from term representation at the GUI to the representation required for computer mathematics, and the all the way back, is shown in Fig. 2.10.

\begin{figure}[h]
\centering
\includegraphics[width=0.6\textwidth]{translation_process}
\caption{Translation from the graphical representation to term vice versa.}
\end{figure}

The process comprises the same separation of concerns between presentation format and internal format, which distinguishes MathML presentation format from content format as discussed in section 2.2.2. The transformations of the process in detail is as follows:

1. Pixels are handled by GUI-components and transformed into an AST, which is as close as possible to the graphical representation.
2. The AST close to graphical representation is transformed such, that its structure is as close to terms as possible.
3. The AST close to term representation is transformed into a term, Isabelle’s data structure for deduction and computation in the prover.
4. From the prover come, among others, terms (i.e. formulas, assumptions, programs, etc) which are transformed back to an AST the easiest way; thus the structure of this AST is (again) close to term representation.
5. The AST close to term representation is transformed back such, that its structure is as close to graphical representation as close as possible.
6. Finally, the resulting AST is transformed into graphical representation.
A specific issue for formula editors is raised by the fact that Isabelle’s collection of theories is extensible arbitrarily, including mathematical notation. For instance, lists are defined as follows in Isabelle, the datatype for deduction and computation (the operation \# :: 'a | a list => 'a list is defined with operator precedence 65) as well as the mathematical notation \( a\#[b, c] = [a, b, c] \), as shown in Fig. 2.11 on p. 15.

The consequences from these facts, alongside with others, for designing a prototype editor will be reflected by software requirements later on.

**Isabelle’s Formula Transformations**

In order to prepare discussion of Isabelle’s transformations between presentation format and internal format of formulas, the transformations are shown for the formula \([aaa,bbb,ccc]\), a list with three elements.

Annotated Syntax Trees (AST) are the central part in Fig. 2.10. ASTs are indispensable for handling “syntactic sugar” like parentheses of lists. In order to watch the process from Fig. 2.10, the term \([aaa,bbb,ccc :: int]\) is packed into a dummy lemma:

```
lemma "[aaa,bbb,ccc::int] = ddd"
oops
```

Now the AST, which comes from the input of the lemma looks like this (output in a LISP-like style implemented in Isabelle):

```
pre:
("const">HOL.Trueprop"
("\<const">HOL.eq"
("_list"
 ("_args" ("_constrain" aaa <position>))
 ("_args" ("_constrain" bbb <position>))
 ("_constrain" ("_constrain" ccc <position>)
 ("\<type>Int.int")))))
 ("_constrain" ddd <position>))
```

The process and the data structures used in formula representation are very close to these used in compiler construction.
This AST contains positions (not printed in detail above), which Isabelle uses for error messages such that the messages can be shown at the location, where the error comes from.

The above AST is transformed such that the structure is as close to the structure of a term (terms are those datastructures, which Isabelle uses for reasoning and calculating):

post:
("\"\"const\"HOL.Trueprop"
  ("\"\"const\"HOL.eq"
    ("\"\"const\"List.list.Cons" ("_constrain" aaa <position>))
    ("\"\"const\"List.list.Cons" ("_constrain" bbb <position>))
    ("\"\"const\"List.list.Cons"
      ("_constrain" ("_constrain" ccc <position>)
       "\<type>Int.int")
      "\"\"const\"List.list.Nil")))
  ("_constrain" ddd <position>)))

The term generated from this AST has a very similar structure:

Const ("HOL.eq", "int list => int list => bool") $
n (Const ("List.list.Cons", "int => int list => int list")$
  Free ("aaa", "int") $
  (Const ("List.list.Cons", "int => int list => int list")$
   Free ("bbb", "int") $
   (Const ("List.list.Cons", "int => int list => int list")$
    Free ("ccc", "int") $
    Const ("List.list.Nil", "int list")))$
  Free ("ddd", "int list")$

Each element of this term has a type: int list the list elements and "int => int list => int list" the list constructor. The types are automatically determined by Isabelle’s type inference \[2, 6\]. The types increase reliability of Isabelle’s mechanical reasoning and go far beyond the kind of types known from programming languages.

After an internal operation in Isabelle, from the above term the following AST is created, which contains the type information in order to allow to show it on user request:

pre:
("\"\"const\"HOL.Trueprop"
  ("\"\"const\"HOL.eq"
    ("\"\"const\"List.list.Cons" ("_constrain" ("_free" aaa)
       "\<type>Int.int")
    ("\"\"const\"List.list.Cons" ("_constrain" ("_free" bbb)
       "\<type>Int.int")

2. Preliminaries
2. Preliminaries

Then the translations work the other direction and create an AST, which is convenient for Isabelle to render the output.

```plaintext
(post: (
  ("\<\text{const}\>HOL.Trueprop"
  ("\<\text{const}\>HOL.eq"
    ("\_\_\_\_\_\_\_\_\text{list}\"
      ("\_\_args\" ("\_\_\_\_\_\_\_\_\_\text{constrain}\" ("\_\_\_\_\_\_\_\_\_\text{free}\" aaa) "\<\text{type}\>Int.int")
       ("\_\_args\" ("\_\_\_\_\_\_\_\_\_\text{constrain}\" ("\_\_\_\_\_\_\_\_\_\text{free}\" bbb) "\<\text{type}\>Int.int")
        ("\_\_\_\_\_\_\_\_\_\text{constrain}\" ("\_\_\_\_\_\_\_\_\_\text{free}\" ccc) "\<\text{type}\>Int.int")
      )
    )
  )
)
)
```

Note, that this structure for output is the same as the initial one from input, just positions have been replaced by types.

### 2.4 The Front-end of the İS4C-Prototype

İS4C is one of three projects on educational software for engineering mathematics based on TP technology: MathToys \[19\] classifies itself a “toy” while E-Math \[1\] already started as an economic enterprise. So even judgment of seriousness of such undertaking is not yet settled.

The İS4C project considers prototyping as a serious long-lasting \[7\] research effort driven by development of TP technology and by requirements of learning mathematics, presently focused on engineering education. Prototyping already clarified \[12\], that the features of TP are apt to generate a new generation of educational mathematics software. In the sequel we focus those aspects of İS4C, which are relevant for designing and implementing an editor, and start with the front-end.

The front-end is connected with the mathematics engine Isabelle/İS4C not via the standard interface \[24\], but by a specific one \[7\]. This special interface transports terms\[8\] as shown in Fig.2.12. The reader may note, that formulas from the front-end back to Isabelle are encoded as strings to be re-parsed and typed correctly\[9\].

---

\[7\]http://www.ist.tugraz.at/isac/History

\[8\]We use the word “term” for the data structure used for mathematical reasoning and “formula” as a general notion without reference to data structure.

\[9\]Re-parsing of formulas encoded as strings shall remain as is according to section 2.5.1
2. Preliminaries

Terms have first been exploited in \( \mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C} \) by [8]. Most of \( \mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C} \) still represents terms as strings. Both together is possible, because all formulas are transported by a Java object \( \text{Formula} \), which contains both, the formula as string and the formula as term (see the connection between Isabelle/\( \mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C} \) and \( \text{Bridge} \) in Fig.2.12).

The front-end manages various kinds of formulas, in examples, in formal specifications, in assumptions, in explanations etc. Fig.2.13 shows the front end in the state before beginning of this thesis: The headline on top shows

![Figure 2.12](image1.png)

**Figure 2.12:** Transport of formulas from the front-end to the back-end.

![Figure 2.13](image2.png)

**Figure 2.13:** The previous \( \mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C} \) front-end.

some buttons (\( \textit{NEW} \) for starting a calculation like in a computer algebra
system, *Examples* for calling prepared examples in the *Example browser*,
*Theories, Problems* and *Methods* calling respective hierarchies of mathe-
matics knowledge, *NEXT* and *AUTO* call for a next step and the final result,
respectively) — all these are not relevant for this thesis.

In the panel below the headline on the left is an example with number
7.70. A click on this number had started interactive problem solution in
the *Worksheet* in the middle of the panel. On the right side of the panel
a part of the *Theory browser* is visible, which has been called by the step
from \(-q \ x = -q_0\) to \(Q^' \ x = -q_0\) in the *Worksheet* (from line 3 to line 4).
The *Worksheet* is part of Isabelle/\(\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}\) in Fig.2.12 like all other components
shown in Fig.2.13 on p.18.

The *Theory browser* shows an explanation for this step 10.

The windows on the left and on the right are not concern of this thesis,
only the window in the middle and in the focus, the *Worksheet*. The *Work-
sheet* is the center of activity during interactive problem solving: from here
are called formal specification of the problem and of subproblems, methods
to solve problems, explanations for certain steps (as shown in the figure)
etc.

The *Worksheet* holds the steps during interactive problem solving, to-
gether called a *Calculation* in \(\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}\). Indentations of steps indicate sub-
problems (like the integration in Fig.2.13). Steps consist of two text lines,
one for the actual formula (with significant indentation) and the other for
the formal justification shifted to the right margin (not shown in the figure).
The red marker indicates, that there is no formula input and approved by
Isabelle/\(\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}\).

Further functionality is provided by the *Worksheet*: a right mouse click
on a formula opens a context menu with the following entries:

- **Assumptions**: lists the assumptions generated in this step.
- **Tactic applied**: the tactic provided by the system if the step has been
done by Isabelle/\(\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}\).
- **Intermediate steps**: opens a deeper level of formulas if available, e.g.
in a sub-problem.
- **Tree representation**: displays the respective formula in a representation
for visually impaired students as implemented by \[8\]. As soon as this
functionality will be covered by the editor under construction, this
entry will disappear.

A right mouse click on a tactic field presently opens a context menu listing
some \[11\] tactics to be applied to the selected step.

---

10 There is an inconsistency in variable naming: the shear force is named \(Q\) in the
*Worksheet* and \(V\) in the explanation. The reason is simple: the engineer involved in the
respective project \[15\] changed is mind quicker than authoring could accomplish within
time limits.

11 \(\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}\)'s dialogue guide is prepared to present different lists, as soon as a respective user
All in all, the Worksheet incorporates much functionality and accordingly is a complex software module with various kinds of input and of user interactions.

## 2.5 Requirements for the Editor Prototype

The preliminary work as described above leads to decisions by İŞAC’s project leader in order to cut down the extent of efforts for the practical part of the thesis.

Nevertheless, user requirements are stated in full coverage and without any curtailment. What can not be accomplished within this thesis, will be recorded in the final summary. But the software requirements capture consequences of the preliminary design decisions of the project leader. This particularly holds for decisions about the programming language Scala.

Use cases are postponed to a later chapter, where design considerations have clarified several issues already.

### 2.5.1 Downcut Decisions for Prototyping

In an early phase of work on this thesis the project leader stated the following decisions.

**Stay with Java Swing and postpone browsers.** The requirements for İŞAC, those for a formula editor in section 2.5 alone, reflect the possibility, that such software tools will migrate from universities down to high schools and will adapt accordingly. There front-ends for mobiles will be appropriate; but such migration require additional development efforts and will take time (and hopefully technologies on mobiles will have settled, and Java will not have decayed in the meanwhile).

This decision is also combined with the hope, that integration of İŞAC with Isabelle is also possible at the front-end: The Isar proof language \[28\] is designed on a high level of abstraction and implemented using standard concepts of compiler construction. So re-use (see section 3.1 below) of Isabelle/jEdit for Isabelle/İŞAC shall be considered. Probably Isabelle will have an optional browser-based interface in a few years, which would be in line with İŞAC’s migration to mobiles.  

**Restrict formula transformation to one direction,** the direction from the back-end to the front-end. This divides efforts in less than half. The model is implemented: a list with all relevant tactics, a list with applicable tactics (which seduces to blind trial&error clicking), a list adding some not-applicable tactics, etc.

\[12\] At present the major obstacle in re-using Isabelle/Isar for Isabelle/İŞAC is that İŞAC is a multi-user system, while Isabelle has no session management.
other direction, from the front-end to the back-end, stays as is — where formulas are reverted to strings in a format, which can be parsed correctly by Isabelle/\texttt{\textsc{Isar}}.

Input (i.e. in direction from the front-end to the back-end) of subterms requires a parser. This parser shall be implemented just for the use cases and with minimal effort.

\textbf{In this direction drop all but the naked formula,} which is: markers for line breaks, types and references to definitions. Such markers anticipate line breaks at appropriate subterms in case the width of formula representation is limited (e.g. in case the window width changes). Dropping types can get over demonstrations of a prototype. But dropping references to definitions is a real show stopper: major advantages of TP technology as emphasized in section 2.3 cannot be demonstrated in the prototype.

\subsection*{2.5.2 User Requirements}

The user is a student of a course in mathematics or in an engineering subject and doing respective homework. A student also can be visually impaired and thus raises specific requirements 8.

\textbf{UR 2.1 Editing a formula comprises:}

- input from scratch
- delete sub-terms in a formula
- add elements to a formula: variables, numerals and operators
- update arbitrary elements of a formula.

\textbf{UR 2.2 Formulas are IDE-like,} i.e. elements of formulas are linked with underlying definitions of operators and of types by mouse click.

\textbf{UR 2.3 Two-level feedback on input.} Input is finally terminated by a special key, which submits the input formula to Isabelle/\texttt{\textsc{Isar}} for checking logical consistency (e.g. the formula must be derivable from the current context). Such checks are costly and slowly — while there should be immediate feedback on missing parentheses, missing arguments of operators: so there should be another level of feedback restricted to the editor (i.e. without involving Isabelle/\texttt{\textsc{Isar}}).

\textbf{UR 2.4 Two kinds of access: set focus, edit.} A formula or a tactic under focus make various services available, which depend on the respective location in a calculation: for instance, show rules applicable at the formula. Different from this kind of access is editing and updating a formula or a tactic — for editing completely different services are required.
UR 2.5 A formula can be read-only without or with exception of some subterms. Such restriction of users’ options extends the dialogue’s ability in user-guidance. The read-only status is made visible.

UR 2.6 The set of operators is extensible. Mathematics is an evolving science. Thus the language of mathematics, as provided by Isabelle, is extensible — and Isabelle/\texttt{\LaTeX}'s editor thus must be extensible, too.

UR 2.7 Operators can be input by keys or by a context sensitive palette. This anticipates availability on hand-holds with small (touch) screen.

UR 2.8 Operators have the following properties, where some of them are described using “non-standard” names below:

1. **Arity**, the number of arguments: in $a + b$ the operator $+$ has two arguments, $a$ and $b$; in $\int_a^b x^2 \, dx$ there are four arguments (Isabelle prefers three arguments: the last two arguments are written as one term $\lambda x. f$); in $-1$ the $-$ has one argument, but it has two in $a - b$.

2. “Fixity”:
   (a) prefix, e.g. operator $\sin$ is prefix in $\sin x^2$ with argument $x^2$
   (b) postfix, e.g. $i +$ is postfix with argument $i$ in programs
   (c) mixfix, e.g. if $a < b$ then $a$ else $b$ is mixfix with arguments $a < b, a, b$

Note, that also programs are formulas (and in fact, Isabelle’s methods shall be presented via the editor in the future).

3. **Priority**: It is customary that $1 + 2 \cdot 3 = 7$ and not $1 + 2 \cdot 3 = 9$, because $\cdot$ has a higher priority than $+$. So priority is determined by natural numbers.

4. **Symbol**: mathematicians occasionally use strange symbols for specific operators like $\in, \subseteq, \otimes, \otimes^*, \star$ etc.

5. “Layout” in the graphical representation is already different for binary operators (operators with two arguments): operator $/$ produces $\frac{a}{b}$, operator $\cap$ produces $a \cap b$, etc.

6. “Syntactic sugar” concerns specific representations for certain formulas, for instance for lists as $[1, 2, 3]$.

UR 2.9 Operations like $+$, $\cdot$, $-$ etc. are binary. MathML presentation format (see section 2.2.2) reflects associativity of operations by $<\texttt{row}>$ and thus accommodates mathematical habits. \texttt{\LaTeX} insists on rigorous handling of formulas, for instance, when applying algebraic laws: thus $(a + b) + c$ and $a + (b + c)$ should not be the same. $+$ is left associative, so only for $(a + b) + c$ parentheses are omitted to $a + b + c$, and not for $a + (b + c)$. 
UR 2.10 *A formula can have several fill-in-gaps.* See UC[3.6] Fill-in-gaps shall also be used for guidance at input, for instance at input of a definite integral like \(\int_{a}^{b} f(x) \, dx\), the graphical representations appears with fill-in-gaps, here denoted by \(\framebox{}\).

UR 2.11 *Special keys let jump from gap to gap in a formula.* UR[3.6] shows several fill-in-gaps. The user is supported to place the cursor to other gaps without using the mouse.

UR 2.12 *Subterms can be marked by certain colors.* See UC[3.6] where the colored areas are indicated by rectangles. Note, that the colors can be related within certain formulas (e.g. lines 05, 06 and 07 in UC[3.6]).

UR 2.13 *The system can place the cursor to a subterm.* See UC[3.7] However, the user is free to manually set the cursor everywhere in a formula.

UR 2.14 *The user can set the cursor to any sub-term* in a formula independently from writable or read-only according to UR[2.5] In case of writable, editing occurs at the position of the cursor.

UR 2.15 *Input is immediately represented in two dimensions.* Input comes from the keyboard and appears at the cursor-position (UR[2.14]). Specific operations (e.g. division) immediately lead to specific representation (e.g. a horizontal fraction bar).

UR 2.16 *Line breaks regard the structure* such that operators with high priority are held together and breaks are in-between elements bound by lower priority. This feature also applies dynamically when the width of windows changes.

UR 2.17 *There are keys for navigating through the sub-terms* of a formula. These <keys> are in detail:

- <Alt> + <↓> next subterm on the same level
- <Alt> + <↑> previous subterm on the same level
- <Alt> + <→> one level down in the tree of sub-terms
- <Alt> + <←> one level up in the tree of sub-terms

These keys are necessary for the inclusion(see section [2.1]).

UR 2.18 *Navigation keys are preset and can be changed*

UR 2.19 *Hitting inappropriate keys* (e.g. <Alt> + ↑ at the root of the sub-terms’ tree) triggers auditive feedback.
2. Preliminaries

UR 2.20 (Sub-)terms are represented on the braille display as strings. Together with UR 2.17 even complicated formulas should become comprehensible. The braille indicates the number of a subterm within the sequence on one level (according to UR 2.17) by a natural number and the level by the number of leading blanks after this number (see section 2.1).

UR 2.21 Navigation on formulas is inclusive which means, that a sighted student and a visually impaired student are able to cooperate on the same formula: one via mouse and screen, the other via keys and braille display. Thus, marking of sub-terms by mouse and by navigation keys is analogous, and both trigger output to braille (see section 2.1).

UR 2.22 String-representation of formulas is variable. There are specific formats [8] for presentation of formulas on a braille display (see UR 2.20). The same format is used for input according to UR 2.15 — which might be different for sighted users and for visually impaired.

UR 2.23 A sub-term can be marked such, that both, a sighted student and a visually impaired student can identify the marked region (cf. UR 2.20) and use this information for cooperation (see §2.1).

UR 2.24 Font size can be adapted by the user as well as preset for specific purposes, e.g. demonstration via beamer. The size is the same for all formulas of a Worksheet.

2.5.3 Software Requirements and Scala

The following requirements reflect the leader’s decisions from §2.5.1 which immediately leads into using Scala. So a few words about this fairly novel programming language [17] and it’s relevance for this thesis. The use of Scala is enforced by the fact, that the envisaged editor exchanges formulas with Isabelle/ISCAC as Scala datastructures (more details about that will follow later). Good luck for prototyping within this thesis is that the editor’s GUI has to integrate with Java Swing — and that is what Scala has been built for: provide novel features (like type inference, functional programming, algebraic datatypes, etc) and remain compatible with approved features (Java Virtual Machine, Java Swing, etc). Particularly useful for implementing the editor was Scala’s smooth integration with Java Swing and pattern matching on the formula’s Scala datastructures; this gives short, readable and thus elegant code.

The software requirements following subsequently give a first look ahead into details of the envisaged editor.

SR 2.1 The editor is rendered by Java Swing. ISCAC has the potential to gradually migrate in usage from universities to high school. At the latter
acceptance of $\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}$ requires availability on mobiles. Software technology of mobiles seems not be settled sufficiently in order to decide details for implementation of a formula editor (JavaScript in standard browsers?). So $\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}$ will adhere to the existing front-end in Java Swing for some more years, and thus the editor uses this technology.

SR 2.2 *Scala code is a mirror of Isabelle’s ML code.* Formulas come as Scala structures from Isabelle to the front-end. Presently Isabelle’s handling of formulas is implemented in SML. So part of this handling has to be translated to Scala, and Isabelle’s well-tried structure is mirrored as much as possible. In particular the file structure and the identifiers are mirrored from SML to Scala.

SR 2.3 *Adopt Isabelle’s machinery for defining syntax.* Isabelle’s tools to define new operators appear elegant and exhaustive, such that the interfaces provided by these mechanisms should be used as much as possible also for a LaTeX-like editor. This is particularly related to UR 2.6 and UR 2.8.

SR 2.4 *Formulas are represented as ASTs.* Formulas come from Isabelle as Terms in a Scala structure, which is close to MathML content format (see section 2.2.2). For the editor a structure close to MathML presentation format (see section 2.2.2) is more appropriate. The respective standard structure are annotated syntax trees, abbreviated as “AST”. Isabelle’s ASTs (and also their translation to Scala) reflect subterms sufficiently in order to meet UR 2.12.

SR 2.5 *Boxes and cursor are set within Scala.* Boxes are set via substitutions found by a rewrite engine. While rewriting is done in a type-safe way in Isabelle/$\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}$, setting boxes is done in the front-end at lower costs. AST-AST-translations come with a rewrite engine, which is sufficient for this purpose (where theorems have been pre-selected by Isabelle/$\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}$).

SR 2.6 *Settings are determined in a property file.* Settings comprise colors of marked areas (c.f. UR 2.12), font size (c.f. UR 2.24), etc. Settings are stored in a (Java) property file at the place where the other files reside in the code.

SR 2.7 *Use cases are reflected in the repository* in the JUnit Test-Case in package isac.gui.mawen.TestUseCases.
Chapter 3

Design Considerations for TP-based Editors

The TP-based editor has special requirements which are shown in section 2.5. This chapter discusses design considerations in order to perform these requirements.

3.1 Re-use of Existing Components

Isabelle is a generic proof assistant (see section 2.3.1) with a generic graphical user-interface (see section 2.3.2) — where both are not only generic with respect to their functionality, but also with respect to their software architecture. So re-use of respective components and technology is an obvious offer for designing a TP-based editor in the $\mathcal{ISC}$-prototype. Section 2.3 shows the status quo of these technologies, where section 2.3.3 introduces Isabelle’s terms and ASTs.

User requirements do not mention type annotations in formulas. So an essential design decision is to drop types of formulas and of all the elements of a formula. Further experiences with the $\mathcal{ISC}$-prototype will show whether this decision restricts learning opportunities and whether it is worth the efforts implementing this feature.

With the decision to drop types the next decision is straightforward — we recall that Isabelle uses two structures of formulas:

- Terms for calculation and
- Annotated Syntax Trees (AST) for representation.

To guarantee the compatibility, the $\mathcal{ISC}$-prototype uses the same structures. For the visual representation of formulas we only need the AST structure — and without any type annotations, which simplifies things considerably (c.f. p.15). However, Isabelle actually has no interfaces to transfer an AST to the front-end, so we need to translate the respective Isabelle code explained
3. Design Considerations for TP-based Editors

in section 2.3.3 from SML to something, which can be integrated into the \( \mathcal{S}4C \)-prototype’s Java Swing — and this is in Scala already in use by PIDE and introduced in section 2.5.3.

Since the editor has to be integrated into the \( \mathcal{S}4C \)-prototype, re-use of respective components is obvious as well.

3.1.1 Adapt Isabelle’s AST Transformations

As shown in section 2.3.3 Isabelle uses one and the same machinery for both directions, the transformation from the front-end (graphical representation) to the back-end (typed term) and vice versa. We shall see, that only one direction was taken for translation to Scala and for adaption to the prototype editor.

Not only type annotations have been dropped within prototype development by this thesis (as mentioned above), also several components of Isabelle’s AST transformation machinery have been simplified for:

- Symbol tables, in Isabelle implemented as balanced 2-3-trees, are represented as Scala `Map`.
- Automated generation of rewrite rules for AST-AST-translations from convenient syntax definitions (see for instance Fig.2.11) is replaced by separated and explicit definition of respective rules. However, the format of these rules is an immediate translation from those in SML.
- Special Scala code for AST-AST-translations are circumvented by restricting the use cases such, that these are not required in the prototype.

Translation from Isabelle code written in SML to Scala code was straightforward. In particular, the Isabelle developer team established a functional style of programming, which has been pertained during translation for the \( \mathcal{S}4C \)-prototype (for details see the chapter on implementation below).

The design decision for dropping types allows for another decision: For the prototype editor only the direction from the back-end to the front-end is needed. The other direction is circumvented by continuing \( \mathcal{S}4C \)’s status quo, the representation of formulas by strings. The new editor’s structured operation on formulas is based on ASTs, but after delivery of an input formula to Isabelle/\( \mathcal{S}4C \) the ASTs are transformed into a string, which can be parsed back and typed by Isabelle. For typing \( \mathcal{S}4C \) maintains a logical context implemented by another student’s project [10]. Of course, this is inefficient, but makes this thesis maximally independent from other changes in the \( \mathcal{S}4C \) system.
3. Design Considerations for TP-based Editors

3.1.2 Integration into the $\mathcal{I}$S$\mathcal{A}$C Architecture

Before integration of the formula editor into $\mathcal{I}$S$\mathcal{A}$C can be considered, relevant components of the Worksheet need to be identified. $\mathcal{I}$S$\mathcal{A}$C's architecture provides the Worksheet with very few, but highly complex connections. This is shown in Fig. 3.1:

![Figure 3.1: $\mathcal{I}$S$\mathcal{A}$C's architecture for user interaction on formulas.](image)

This architecture differs from the MVC architecture\[3\] in that the dialog occupies a central role: the WorksheetDialog is part of a DialogGuide (not shown in the figure), which coordinates the WorksheetDialogs (each Worksheet has its WorksheetDialog) and the BrowserDialogs (for theories, problems and methods managing various interactions) according to a user model.

With respect to the Worksheet it controls each UserAction (even read/write access to sub-terms of a formula!) and decides on each information passed to the Worksheet, the main interface for interaction with students. One reason for this architecture is, that $\mathcal{I}$S$\mathcal{A}$C is designed for adaption to various learning scenarios, see. section 2.4.

According to this design, an input formula is packed into a UserAction, like all other user actions on the Worksheet, and sent to the WorksheetDialog for a first classification. Then the WorksheetDialog usually decides to pack the input with the respective position from the Worksheet and to forward to Isabelle/$\mathcal{I}$S$\mathcal{A}$C for checking correctness.

Isabelle/$\mathcal{I}$S$\mathcal{A}$C’s feedback is transformed into a CalcChanged event, which is inspected by the WorksheetDialog: there also might have been an error. In case of correctness the WorksheetDialog requests getFormulaeFromTo (an input formula might have deleted subsequent formulas). This request goes directly to the Bridge, which forwards the request to Isabelle/$\mathcal{I}$S$\mathcal{A}$C and receives a Term. This is transformed to an AST (in the Bridge, see Fig. 2.12) and packed into a CalcFormula, i.e. a formula with the
appropriate position(s) in the *Worksheet* (the plural of position is required for deleting, inserting, etc.).

### 3.2 Integration into the IScAc-Prototype

Chapter 1 has stated a comprehensive code base for the IScAc prototype, section 2.4 gave a description of the front-end and the role of the *Worksheet* and section 3.1.2 showed how architecture connects the *Worksheet* with the other components of IScAc. With this information design of integration can go into details. We shall see, that integration is smooth: it only concerns very few elements from the complex IScAc architecture.

#### 3.2.1 Formulas in Calculations

This thesis focuses formulas in the *Worksheet*, where solutions for problems are interactively constructed. Calculations in the *Worksheet* are displayed in a Java Swing *JTree*. This *JTree*’s uses a custom *TreeCellRenderer* and a *TreeCellEditor*. The *TreeCellRenderer* is for visualization purposes, while the *TreeCellEditor* is for editing — the usual separation of model and view [3].

In the case of a graphical editor this separation is more delicate: data manipulation, i.e. checking formulas and generating feedback, is done in the back-end by Isabelle/ISAc. Another kind of data manipulation is updating the structure of the formula (i.e. an AST) — this kind of manipulation, however, is part of rendering.

Both implementations have been nearly similar. The only difference was the feedback from the *CalcTreeCellEditor* to the *Worksheet* after all kinds of changes. Both built the same graphical structure as one can see in Fig. 3.2.

The *JTextFields* were editable in the *CalcTreeCellEditor* and locked in the *CalcTreeCellRenderer*. Also no Events were fired in the *CalcTreeCellRenderer*. This has been done by the Java Swing *JTree*. So no extra work needs to be done to lock the editor in the visualisation mode — this is mentioned here and will be important for the editor in a future chapter. The new formula editor is placed in the *CalcTreeCellRenderer* and in the *CalcTreeCellEditor*.

#### 3.2.2 Formula Representation

As shown in Fig. 3.2 a calculation is represented by a *JTree* in the *Worksheet*. Steps within such a calculation are represented by nodes in the tree. A *JTree* node has two *JTextFields*. The first one is the *FormulaTextField* and the second one is the *TacticTextField*. Both fields need to be replaced by the editor.
In order to feature smooth transition in development from the old text fields to the new editor, and in order to cope with cases, where the prototype editor does not work as expected, a property file allows the user to choose between the JTextField or the editor.

An EditorFactory determines on the basis of the property file to return the old JTextField representation or the new editor, which is encapsulated in the EditorPanel class. The EditorPanel is explained in section 4.1.

3.3 Interface for a Formula Editor

Here the above design considerations are driven towards implementation in two different ways, (1) by declaring an interface and (2) by fixing use cases.

3.3.1 The Java interface

This is the interface between the existing code of \texttt{\texttt{ISE}} and the new editor. As mentioned above, this interface also covers the previous implementation of the editor by JTextField.

3.3.2 Use Cases

This section is according to the standard notion. The numbering within the use cases indicate the level of priority. Use cases also reflect \texttt{\texttt{ISE}}’s prototype status and address specific demonstrations. Beginning with UC 3.7 the individual cases are ranked by priority.
3. Design Considerations for TP-based Editors

![Diagram showing EditorFactory and IEditor interfaces with methods like getTacticEditor() and getFormulaEditor()](image)

**Figure 3.3:** On the right side there is the graphical representation of the JTree and on the left side there is the structure of the UI Components.

**UC 3.1** *Operators ranked by urgency for prototype demonstration*

1. addition and subtraction $+$ $-$, multiplication $\cdot$, division $\frac{x}{y}$
2. differential operator $\frac{d}{dx}$
3. integral $\int_{a}^{b} f(x) dx$
4. further operators according to UR.2.0

**UC 3.2** *Variables ranked by urgency*

1. variables consisting of letters, digits and underscore
2. variables with subscripts
3. variables with foreign fonts (e.g. Greek)

**UC 3.3** *Numerals ranked by urgency*

1. integer numbers
2. floating point numbers
3. complex numbers

**UC 3.4** *Datatypes ranked by urgency*

1. lists like $[1, \frac{1}{2}, \int_{0}^{2} x \, dx]$ or $[\text{setzeRandbedingungen, Biegelinien}]$ (compare UC.3.8)
2. vectors like $\begin{pmatrix} x \\ y \end{pmatrix}$ or $(x \ y)$

3. matrices like $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ or $\begin{pmatrix} 1 & b \\ d & \int_0^2 x \, dx & f \end{pmatrix}$

**UC 3.5 Demo of simplification of fractions.** Significant formulas in the respective calculation are:

Simplify $\frac{1 - 1 / 2 - (2 / (x + 3) + 2 / (x - 3)) / (8 * x / (x ^ 2 - 9))}{1 - 1 / 2 - (2 / (x + 3) + 2 / (x - 3)) / (8 * x / (x ^ 2 - 9))}$:

The formula in graphical representation

$$1 - \frac{1}{2} - \frac{\frac{2}{x+3} + \frac{2}{x-3}}{\frac{8}{x} \cdot \frac{x}{x^2-9}}$$

demonstrates a specific benefit of section 2.2.3 the readability of complex formulas improved by different font sizes.

**UC 3.6 Input of specific operators is supported by fill-in-gaps.** Input of $a + \frac{b}{x}$ can proceed in several ways

1. Input starts always with an operator:
   
   (a) Input starts with “+”, this creates the rendering

   $\bigotimes + \bigcirc$

   where $\bigcirc$ denotes a gap and $\bigotimes$ denotes a gap with cursor.

   (b) After input of “a”, in the second GAP a “/” is input, this creates the rendering

   $a + \bigotimes$

   (c) After input of “b”, a special key moves the cursor (see UR 2.11) to the denominator:

   $a + \frac{b}{\bigcirc}$

2. Input of infix operators start with the first argument:

   (a) After input of “a”, the “+” is input which renders as

   $a + \bigotimes$
3. Design Considerations for TP-based Editors

(b) The input of “b / ” (ending with a blank) into the gap above renders as

\[ a + \frac{b}{\times} \]

(c) A special key works as in Pt.1c

3. The input variants Pt.1 and Pt.2 can be mixed arbitrarily.

**UC 3.7 Demo of user-guidance.** User-guidance involves marked sub-terms and fill-in-gaps. Below a next formula is suggested by a matching rule and a resulting formula. Given a student, who fails to apply the chain rule at the formula \( \frac{d}{dx} \sin(x^2) \) in line 05 within some arger calculation; then \( \mathcal{L}\mathcal{S}\mathcal{A}\mathcal{C} \) has several ways to help, for instance with output as in lines 06 and 07:

\[
04 \ldots
\]
\[
05 \frac{d}{dx} x + \frac{d}{dx} \sin(x^2)
\]
\[
06 \text{Rewrite} \left( \frac{d}{dx} \sin(x) \right) = \cos(x) \cdot \frac{d}{dx} x
\]
\[
07 \frac{d}{dx} x + \cos(x^2) \cdot \frac{d}{dx} x
\]
\[
08 \ldots
\]

In line 07 \( \mathcal{L}\mathcal{S}\mathcal{A}\mathcal{C} \) presents a formula with a fill-in gap and the CURSOR placed into this gap, indicated by \( \boxtimes \). Note that the first formula in this calculation is

\[ \text{Diff} \left( x^2 + \sin(3 \cdot x^4), x \right) \]

i.e. in graphical representation

\[ \text{Diff}(x^2 + \sin(3 \cdot x^4), x) \]

**UC 3.8 Demo of engineering mathematics.** This demo shows, how a problem in engineering math is decomposed into sub-problems, where each step towards a solution is justified. So each step can be related to some theorem, and each theorem can be accompanied by multimedia explanations. Significant formulas in the respective calculation are

\[
y x = \text{Integral} c_3 + 1 / (-1 \ast EI) \ast (c_2 \ast x + c / 2 \ast x \ast 2 + -1 \ast q_0 / 6 \ast x \ast 3) \text{ D} x
\]

Problem (Biegelinie, [setzeRandbedingungen, Biegelinien])
3. Design Considerations for TP-based Editors

\[ L \cdot q_0 = c, \quad 0 = (2 \cdot c_2 + 2 \cdot L \cdot c + -1 \cdot L^2 \cdot q_0) / 2, \quad 0 = c_4, \quad 0 = c_3 \]

The first and last formula in respective graphical representation are:

\[
y \cdot x = \int c_3 + \frac{1}{-1 \cdot EI} \cdot \left( c_2 \cdot x + \frac{c}{2 \cdot x^2} + -1 \cdot \frac{q_0}{6 \cdot x^3} \right) \cdot Dx
\]

\[ [L \cdot q_0 = c, \quad 0 = 2 \cdot c_2 + 2 \cdot L \cdot c + -1 \cdot L^2 \cdot q_0, \quad 0 = c_4, \quad 0 = c_3] \]

Graphical representation makes clear, that there are questionable results of simplification in \( \mathcal{I} \mathcal{S} \mathcal{A} \mathcal{C} \), e.g. \(-1 \cdot \frac{q_0}{6 \cdot x^3} \).

**UC 3.9 Demo manipulation of sub-terms.** Given the equation \( x + 1 + -1 * 2 = 0 \) from within an interactive calculation, support the following operations (also with respect to inclusion \( \text{UR}[2.21] \)):

1. Replace the subterm \(-1 * 2\) by subterm \(-2\)
2. Replace the subterm \(1 + -1 * 2\) by subterm \(1 - 2\). Note, that parentheses, if invisible, in associations of + are \((x + 1) + -1 * 2)!\)
3. Cut the subterm \(1 + -1 * 2\) on the left hand side of the equation and paste it on the right-hand side resulting in \(x + 0 = 0 - (1 + -1 * 2)\)

**UC 3.10 Programs are terms.** Thus language constructs like

\[
if \ n < 0 \ then \ fac \ n = n * fac(n - 1) \ else \ 1
\]

should be manageable by an editor. The operator is \(if\), which takes three arguments: \(0 < n, \ fac \ n = n * fac(n - 1)\) and \(1\). The envisaged editor, however, shall not take care of syntax high-lighting.

Program Simplify t_t =

```
Repeat {
  Try (Rewrite_Set kllammern_aufloesen False) @@
  Try (Rewrite_Set ordne_alphabetisch False) @@
  Try (Rewrite_Set fasse_zusammen False) @@
  Try (Rewrite_Set verschoenere False)) t_t
```

3.4 Datatypes

Section \[2.3.3\] demonstrated Isabelle’s transformation on annotated syntax trees (ASTs). Section \[2.4\] mentioned that \( \mathcal{I} \mathcal{S} \mathcal{A} \mathcal{C} \)’s interface to Isabelle/\( \mathcal{I} \mathcal{S} \mathcal{A} \mathcal{C} \) transports terms; these terms arrive as Scala data structures at the front-end side of libisabelle. In order to take profit from Isabelle’s general machinery of formula transformation, the whole machinery needs to be transferred from Isabelle’s SML to the Scala in the \( \mathcal{I} \mathcal{S} \mathcal{A} \mathcal{C} \) front-end.
3.4.1 Term, the mathematical Object

Formulas are represented as terms for the purposes of computer mathematics. Isabelle has two identical implementations of terms, one on the mathematics side implemented in SML and one on the front-end side implemented in Scala. The definition in Scala, already implemented by the interface mentioned in section 2.4, is this:

```scala
sealed abstract class Term
  case class Const(name: String, typ: Typ = dummyT) extends Term
  case class Free(name: String, typ: Typ = dummyT) extends Term
  case class Var(name: Indexname, typ: Typ = dummyT) extends Term
  case class Bound(index: Int) extends Term
  case class Abs(name: String, typ: Typ = dummyT, body: Term) extends Term
  case class App(fun: Term, arg: Term) extends Term
```

The term constructors relevant for a formula editor are `Const` encoding operators, `Free` encoding variables and `App` connecting operators with their arguments. `Var` is concerned with meta-variables in patterns, `Bound` and `Abs` are concerned with high-order terms, both not relevant for our purpose on editors as well as all the types `Typ`.

The term `[aaa, bbb, ccc]` used for demonstration in section 3.4.2 above is implemented as Scala `Term` as follows:

```scala
App(App(Const("List.list.Cons",
    Type("fun",List(Type("Int.int",List())),
    Type("fun",List(Type("List.list",
        List(Type("Int.int",List())))),
    Type("List.list",List(Type("Int.int",List()))))),
  Free("aaa",Type("Int.int",List()))),
App(App(Const("List.list.Cons",
    Type("fun",List(Type("Int.int",List())),
    Type("fun",List(Type("List.list",
        List(Type("Int.int",List())))),
    Type("List.list",List(Type("Int.int",List()))))),
  Free("bbb",Type("Int.int",List()))),
App(App(Const("List.list.Cons",
    Type("fun",List(Type("Int.int",List())),
    Type("fun",List(Type("List.list",
        List(Type("Int.int",List())))),
    Type("List.list",List(Type("Int.int",List()))))),
  Free("ccc",Type("Int.int",List()))),
```
Another Term \(\aaa + \frac{4}{\bbb}: \text{real}\) is implemented like this:

\[
\text{App(App(Const("Groups.plus_class.plus",}
\text{Type("fun",List(Type("Real.real",List())),}
\text{Type("fun",List(Type("Real.real",List())),}
\text{Type("Real.real",List()))))))},
\text{Free("aaa",Type("Real.real",List()))},
\text{App(App(Const("Fields.inverse_class.divide",}
\text{Type("fun",List(Type("Real.real",List())),}
\text{Type("fun",List(Type("Real.real",List())),}
\text{Type("Real.real",List()))))))},
\text{App(Const("Num.numeral_class.numeral",}
\text{Type("fun",List(Type("Num.num",List())),}
\text{Type("Real.real",List()))),}
\text{App(Const("Num.num.Bit0",}
\text{Type("fun",List(Type("Num.num",List())),}
\text{Type("Num.num",List()))),}
\text{App(Const("Num.num.Bit0",}
\text{Type("fun",List(Type("Num.num",List())),}
\text{Type("Num.num",List()))),}
\text{Const("Num.num.One",Type("Num.num",List()))),}
\text{Free("bbb",Type("Real.real",List()))})
\]

This term shows, that Isabelle internally relies on binary encoding of numbers: \text{Num.num.Bit0} and \text{Num.num.One} denote the binary number \(100_2 = 4_{10}\) (in reverse order). The reason for this encoding is typical for provers of the HOL-family: using arithmetic from some processor would introduce respective unreliability to Isabelle — while the binary representation is manipulated by Presburger arithmetic \[20\] and rewriting, which is proved correct within Isabelle.

Strings like "Groups.plus_class.plus" are compiled partly automatically by Isabelle due to logical dependencies ("Groups.thy", "plus_class", etc), see Fig.3.4. Logical dependencies are up to frequent changes in Isabelle. So these strings are not at all fixed. The translation from binary encoding to decimal encoding for human users is done by AST-AST-translations.

### 3.4.2 Annotated Syntax Tree

Isabelle’s formula transformations on ASTs have been introduced in section 2.3.3, SR 2.4. We translate the SML data structure into the following Scala datastructure one-to-one:

```scala
object Ast {
```
sealed abstract class Ast
    case class Constant(name: String) extends Ast
    case class Variable(name: String) extends Ast
    case class Appl(name: List[Ast]) extends Ast

def ...

Scala requires a sealed abstract class in order to inhibit modification by programmers later on. The structure is much simpler than a Term: Constant represents function constants (e.g. algebraic operators), Variable takes variable identifiers and numerals. A Variable can be matched with an arbitrary AST. Appl allows to recursively build a structure of ASTs by assembling them in a List of ASTs. The Term introduced on p.35 is represented as an AST as follows:

Appl[Constant("\"const\List.list.Cons\""),
    Appl[Constant("_constrain"),
        Appl[Constant("_free"), Variable("aaa")],
        Appl[Constant("_ofsort"),
            Appl[Constant("_tfree"), Variable("'a")],
            Constant("\"<class>HOL.type\""")],
    Appl[Constant("\"const\List.list.Cons\""),
        Appl[Constant("_constrain"),
            Appl[Constant("_free"), Variable("bbb")],
            Appl[Constant("_ofsort"),
                Appl[Constant("_tfree"), Variable("'a")],
                Constant("\"<class>HOL.type\""")],
        Appl[Constant("\"const\List.list.Cons\""),
            Appl[Constant("_constrain"),
                Appl[Constant("_free"), Variable("ccc")],
                Appl[Constant("_ofsort"),
                    Appl[Constant("_tfree"), Variable("'a")],
                    Constant("\"<class>HOL.type\""")],
                Constant("\"<const>\List.list.Nil\"")]]]

The above AST shows the costs for users’ convenience to request the types for any element in a formula any time.

After AST-AST-translations this AST is transformed to this version.
3. Design Considerations for TP-based Editors

Appl[Constant("_list"),
    Appl[Constant("_args"),
        Appl[Constant("_constrain"),
            Appl[Constant("_free"), Variable("aaa")],
            Appl[Constant("_ofsort"),
                Appl[Constant("_tfree"), Variable("'a"),
                    Constant("<^class>HOL.type")],
                Constant("<^class>HOL.type")],
            Appl[Constant("_constrain"),
                Appl[Constant("_free"), Variable("aaa")],
                Appl[Constant("_ofsort"),
                    Appl[Constant("_tfree"), Variable("a"),
                        Constant("<^class>HOL.type")],
                    Constant("<^class>HOL.type")],
            Constant("<^class>HOL.type")]]]

This version is more convenient for Isabelle’s conversion to an Isabelle-string in the GUI, see Fig.2.9. This thesis, too, creates a simpler version by Scala code, which however has the same structure shown below.

3.4.3 Specific Decisions on ASTs

Here are some decisions on the datastructure underlying graphical representation of formulas within the practical work of this thesis.

Make ASTs as simple as possible. Section 2.5.1 decided to “drop all but the naked formula, which is: markers for line breaks, types and references to definitions”. The AST actually created by the SML code in Isabelle has been shown on p.37, the AST created by the new Scala code is

Appl(List(
    Constant("List.list.Cons"),
    Variable("aaa"),
    Appl(List(
        Constant("List.list.Cons"),
        Variable("bbb"),
        Appl(List(
            Constant("List.list.Cons"),
            Variable("ccc"),
            Constant("List.list.Nil")))))
)

After AST-AST-translations this AST is transformed to this version:
Appl(List(
    Constant("_list"),
    Appl(List(
        Constant("_args"), Variable("aaa"),
        Appl(List(
            Constant("_args"), Variable("bbb"),
            Variable("ccc")))))))

Comparison with Isabelle’s ASTs including types on p.37 shows how much simpler this version is. Note, however, the little differences in structure between the former (SML code) and the latter (Scala code). This decision is in conflict to UR.2.2 and restricts UR.2.3.

Another decision (not mentioned in section 2.5.1) is to represent numerals not in binary format as shown on p.36 but in decimal format in a Variable, i.e. the editor works on this AST, where the graphical representation is $aaa + \frac{4}{bbb}$:

Appl(List(
    Constant("Groups.plus_class.plus"),
    Variable("aaa"),
    Appl(List(
        Constant("Fields.inverse_class.divide"),
        Variable("4"),
        Variable("bbb")))))

Represent specific functionality in ASTs. In order to keep the interface to the envisaged editor simple, functionality is packed into the data structure. Such functionality concerns marking of sub-terms, fill-in gaps and a possibility for the system setting the cursor.

Given the data structure introduced in section 3.4.2 specific Constants are inserted and deleted in the structure (without changing the mathematical semantics) as follows:

marked subterms

(a) $ast \leftrightarrow \text{Appl(List(}\text{Constant(}'BOX'\text{)}, \text{Constant(}'1'\text{)}, ast))$

(b) $ast \leftrightarrow \text{Appl(List(}\text{Constant(}'BOX.nnn'\text{)}, ast))$

fill-in gaps

$ast \leftrightarrow \text{Constant(}'GAP'\text{)}$

cursor set by system

$ast \leftrightarrow \text{Appl(List(}\text{Constant(}'CURSOR'\text{)}, ast))$

For marked subterms the variant (b) has been chosen, because Scala allows an if within patterns (which is not possible in SML). All above productions can be combined arbitrarily (as long as not restricted by constraints in implementation).
Marking of sub-terms are requested by UR.2.10, UR.2.11, and UR.2.23. Inclusive navigation on formulas, UR.2.21, is accomplished by presenting the marked sub-term on the braille (as a string).

Fill-in gaps are sub-terms offered to the user for input. This refers to UR.2.10, UR.2.11, and UC.3.6. Replacement of a sub-term by a fill-in gap can be done within Scala, so the mathematics engine Isabelle/ISC needs to be called.

The cursor can be set by the system according to UR.2.13.

At the beginning of input of a formula, the respective AST (i.e. an empty formula) is as follows:

\[
\text{Appl(List(}
\text{Constant("CURSOR")},
\text{Constant("GAP")))}
\]

3.5 Layout Classes for Sub-Terms

This thesis works on a limited number of use cases (see section 3.3.2), but explores challenges in generic solutions for formula editors. Isabelle provides users with elegant mechanisms to specify new syntax as briefly shown in section 2.3.3. Such mechanisms cannot be provided by this thesis; but we identify classes of layout, which shall be handled by different code and (hopefully) cover all kinds of \LaTeX’ rendering of formulas (we exclude tricks in raw \TeX).

Binary infix operators in line are +, *, −, = (within the use cases) and others. These operators are one certain line (if in a line at all – see the subsequent two classes). A specific challenge here is associativity: \((a+b)+c = a+b+c\) and \(a+(b+c)\) are different terms according to UR.2.9. This UR excludes MathML’s row introduced in section 2.2.3. Parentheses adjust their size to the enclosed sub-term

\[
a \cdot (b + c) + \left(\frac{d}{e} + f\right)
\]

Fraction-like operators are at least binomial coefficients \(\binom{N}{k}\) and fractions. \LaTeX reduces font size of numerators and nominators:

\[
a + \frac{\frac{h+i}{j}}{\frac{k}{j+k}}
\]
However, \( \text{\LaTeX} \) reduces size only down to three levels of subterms, the topmost \( \frac{b+c}{\frac{d}{e}} \) has the same size as the respective nominator \( f+g \) (the spaces between the fraction bars are larger, because \( f \) is larger than \( d \) or \( c \)).

**Exponentiation** shifts arguments not only vertically but also shifts arguments horizontally:

\[ x \quad \quad x^{x^{x^{x}}}. \]

Also in this class, \( \text{\LaTeX} \) reduces size only down to three levels: the topmost three \( x \) all have the same size.

**Prefix operators** like \( \sin, \cos, \text{solve}(x+1=2, x) \) have different kinds of arguments (terms, pairs as with \( \text{solve}, \text{etc.} \)) and different arity (i.e. a different number of arguments). This thesis omits postfix operators.

**List-like term collections** are lists \([a, b, c]\), tuples \((a, b, c)\) and enumerated sets \(\{a, b, c\}\). Set comprehension like \(\{n; \colon; n<100\}\) is omitted in this thesis.

**Matrix-like term collections** cover various numbers of lines and rows greater-equal one:

\[
\begin{pmatrix}
a & b & c \\
(\quad)
\end{pmatrix}
\quad
\begin{pmatrix}
a & b \\
c & d \\
(\quad)
\end{pmatrix}
\quad
\begin{pmatrix}
a \\
(\quad)
\end{pmatrix}
\quad
\begin{pmatrix}
\frac{b}{x} & b & x \\
c \\
\int_a^b x^2dx & y
\end{pmatrix}
\]

**Operators with sub- and super-scripts** and other fancy features which mathematicians invent, when they run out of traditional notation. A very special operator is the differential operator \( \frac{d}{dx}f \), where the two arguments \( x \) and \( f \) are placed very differently. And a specific variability is given by integrals with two, three or four arguments:

\[
\begin{align*}
\int x^2dx & \quad \int_C F ds & \quad \int_a^b x^2dx \\
\end{align*}
\]

The first example above is in the use cases (see section 3.3.2).
Chapter 4

Implementation of a Prototype Editor

This chapter establishes a connection between the design and the code implemented alongside with the thesis. First static aspects of the code are considered in section 4.1 together with general decisions for coding. Section 4.2 discusses relevant details of Java Swing, before dynamic aspects can be addressed in section 4.3. The final section 4.4 describes the procedure of implementation, which had to cope with the comprehensive code base of the \( \mathcal{SAC} \)-prototype.

4.1 Classes and Coding Decisions

Given by design and by preliminary decisions by the project leader, the implementation has to combine two programming languages, Java and Scala: formulas are given as data structures in Scala, and the formula editor is integrated into \( \mathcal{SAC} \)'s front-end written in Java Swing. Before an assignment of these two programming languages can be approached in section 4.1.3, a structure for the editor’s code is introduced in section 4.1.1 and respective integration into \( \mathcal{SAC} \) discussed in detail in section 4.1.2.

4.1.1 Code Structure of the Editor

The issue of a formula editor is to relate graphical representation with a comprehensive data structure representing a formula (actual an AST according to SR 2.4).

Graphical aspects, as imposed by SR 2.1, are addressed by the class EditorPanel and a class AstComponent provides connection to the data structure. Fig 4.1 shows the relation between these two components. The EditorPanel is a JPanel; it is a Java container and embeds the AstComponent (presenting the formula \( x = 2 - y \)). The kind of embedding is determined
by a *OverlayLayout*, which allows to set appropriate parameters for display according to events fired by *AstComponent*.

If the user edits $y$ or continues input after $y$, for instance, then there are two different ways of implementation:

1. Place a `JTextField` over the graphical representation such that respective parts of the formula are hidden (the figure displaces this `JTextField`). Parse the input string and insert the generated sub-AST into the AST. This has the advantage, that a `JTextField` provides much functionality out of the box: event handling, copy and paste, deletion, etc.

2. Listen to events on the keyboard and distinguish between alphanumeric characters one side and all the other characters on the other side. In case of subsequent alphanumeric characters continue with input of an integer number, case of another character change the structure of the formula \[1\] and render the AST accordingly.

The practical work invested considerable time for experiments with both variants.

The *AstComponent* assembles input to an AST and draws into a `JComponent` \[3\]. This is depicted in Fig 4.2. Utilities for drawing are separated into two classes, *CalcUtil* and *DrawUtil*. The former calculates position and size of each AST and returns a `Box` class. The latter draws the boxes. The `Box` class is analogous to the \LaTeX\ boxes mentioned in \[2.2.6\]. The method `paint` uses both, details will be discussed later.

\[1\] This includes several limitations, which do not generalise to a full-fledged editor: there are operators consisting of more than one character, there are not only integers but also decimal points, etc.
4. Implementation of a Prototype Editor

4.1.2 Integration into İŞAC

As shown in section 2.4, the main tool for interaction on formulas is called a Worksheet in İŞAC. A Worksheet represents calculations from engineering mathematics in a Java Swing JTree, where each line contains a step of calculation and certain nodes in the tree can be expanded in order to show calculations on sub-problems. For instance, in Fig.4.3 the node solve \((-1 + x = 0, x)\) could be expanded and show the steps solving this sub-problem.\(^2\) Formulas in this screen shot are still strings, modeled by JTextField s. In-between the steps of calculation there are lines, which are ready to display additional information requested by the user: the justification for the step, assumptions generated by this step, etc as shown in section 3.2.1. The sisac project leader decided, not to consider formulas in this field.

\(^2\)This example is the most beloved one in İŞAC’s test suite, because it comprises much of İŞAC’s functionality (not shown in the screen shot). The example also shows, how general solutions might look strange if applied to specific (even simple) examples: solve does what any computer algebra system can do, but it shows intermediate steps: it solves a large class of equations; it first generates a specific normal form \(-1 + x = 0\), which allows to determine the degree of the equation mechanically. This and other information is used to select the appropriate method for solving the recognised class of equations (such selection is done behind the scenes in this case and not shown in Fig.4.3).
Since it is not expected, that this master thesis can completely replace the old string representation of formulas without cutting down other features of ℐ𝒮𝒜𝒞 dramatically, the old string representation needs to be kept alive. For this purpose a **EditorFactory** is introduced according to section 3.2.2, which decides with respect to a property file:

```java
public class EditorFactory {
    private static boolean IsEditorVisible;
    static {
        try {
            InputStream is = WindowApplicationPaths.class
                    .getResourceAsStream("/properties/Editor.properties");
            if (is != null) {
                Properties p = new Properties();
                p.load(is);
                IsEditorVisible = p.getProperty("EDITOR")
                        .replace(" ", ").equalsIgnoreCase("MAWEN");
            }
        } catch (Exception e) {
            IsEditorVisible = false;
        }
    }
    public static IEditor getFormulaEditor() {
        if (IsEditorVisible) {
```

**Figure 4.3:** A Calculation is constructed in a **Worksheet**, i.e. a **JTree**.
The factory decides whether to return a `FormulaTextField` (preserve the previous state) or an `EditorPanel` introduced by this thesis. An example for the property file is in Fig. 4.4 (in conformance with SR 2.6). This example shows, that there are further data, now required by the new formula editor.

As described in section 3.2.1 model and view are separated into `TreeCellEditor` and `TreeCellRenderer`, respectively. These two classes were almost similar for `JTextField`s. Also with the new editor, for both a `Layout` is built by the `EditorFactory`:

```java
editor_ = EditorFactory.getFormulaEditor();
tacticText_ = new JTextField();
editorPanel_ = new JPanel();
editorPanel_.setOpaque(true);
editorPanel_.setBackground(UIManager.getColor("Tree.background"));
editorPanel_.setLayout(new BorderLayout());
editorPanel_.add(editor_.getComponent(), BorderLayout.CENTER);
editorPanel_.add(tacticText_, BorderLayout.SOUTH);
```

Now, with the new editor, both classes `TreeCellEditor` and `CalcTreeCellRenderer` need to observe varying height of formulas (while `JTextField` always had the same height). The new editor reports the height of an edited formula for rendering by the `JTree`’s repaint.

The implementation of the editor introduced two callbacks for the Worksheet

1. `notifyLocalCheck` and
2. `notifyIsaCheck`
where the latter continues the old behaviour and the former adds a new
behaviour: \texttt{notifyIsaCheck} sends the formula to Isabelle (while the AST is
transformed to a string which can be parsed by Isabelle, see section 2.5.1
and Fig.2.12).

New \texttt{notifyLocalCheck} implements an idea arising from the fact, that
much of Isabelle’s machinery for formula transformation is shifted to the
front-end and rewritten in Scala by this thesis. So there are means for local
syntax checks for formulas in order to disburden the back-end (centrally used
by many front-ends). These means could be used to handle units locally (and
thus avoid burdening Isabelle with defining types for each unit).

### 4.1.3 Java – Scala – Java

An interesting challenge for implementation work was to find a good com-
bination of Java and Scala. In Java and Java Swing \(\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}\)’s front-end is
implemented, and the new editor has to be integrated according to SR.2.1.
On the other hand, the formulas to be edited come as Scala datastructures,
as ASTs according to SR.2.4.

The challenge was to assign the parts of the editor’s code to that pro-
gramming language, which better supports the tasks to be programmed
respectively. The assignment turned out straightforward as follows.

- **Scala** was used for code handling Scala ASTs, elegantly exploiting the
  powerful \texttt{match}. Scala code seamlessly calls Java Swing. Examples are
  \texttt{CalcUtil} and \texttt{DrawUtil}.

- **Java** was used for handling the Java Swing components to embed-
ded into the \textit{Worksheet} together with respective event handling. An
  example is \texttt{EditorPanel} extending \texttt{JPanel}.

Referring to Fig.4.2 on p.44 we have: An AST is passed through the Java
\texttt{EditorPanel} to the Scala \texttt{AstComponent}. The AST is manipulated in \texttt{Cal-
cUtil} and \texttt{DrawUtil} by Scala’s \texttt{match}. After editing is finished the AST is
returned by the \texttt{EditorPanel} and the \texttt{Worksheet} (both in Java) is notified.

- **The directory structure** for handling ASTs is copied from Isabelle, as
  much as Isabelle’s machinery has been translated from SML into Scala so
  far. The purpose of the parallel structure is to ease further translation and
  maintenance (according to frequent changes in Isabelle’s low level machinery);
  this is in line with further narrowing between Isabelle and \(\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}\). Some
  classes are already implemented in libisabelle (see section 2.4); in order to
  not overwrite these, respective identifiers get an “X” as prefix, for instance
  \texttt{XSyntax}.

- **Two different coding standards** are the consequence of preparing for
  further narrowing between Isabelle and \(\mathcal{I}\mathcal{S}\mathcal{A}\mathcal{C}\): All Scala code translated from
Isabelle’s SML adheres to Isabelle’s coding standards [27]. Other Scala code and Java code adhere to ℐ𝒮𝒜𝒞’s coding standards [23]. Resulting conflicts in certain program files are taken into account.

4.2 Symbols and Expressions in Java Swing

This section describes, how nicely Java Swing implements the concept of \LaTeX’s boxes in a straight forward manner. So section 2.2.3 is specified for Java below, first in general, then for single symbols in section 4.2.1 for expressions with horizontal expansion in section 4.2.2 and for expressions with vertical expansion in section 4.2.3.

Rendering of formulas is done by overwriting the paint method of JComponent [3]. This method recursively scans an AST top down and draws the respective boxes. Before rendering the box sizes are calculated by CalcUtil bottom up over the AST. So each box can be drawn independently.

Coordinates and positioning in Swing are different from mathematical coordinate systems. In Swing the origin (0/0) is top left. Fig. ref shows that a rectangle is drawn top down while a sequence of characters is drawn bottom
up. This difference leads to difficulties when composing elements of a formula. For instance with fractions, because the width of both, numerator and nominator, needs to be determined before both of them can be positioned within the fraction. These difficulties are encountered best by separating computation of box sizes (in CalcUtil) from drawing boxes (in DrawUtil). Separation, in turn, enforces carrying lots of data, which capture the boxes’ properties. The implementation collects these data in graphical objects of class Graphics.

### 4.2.1 Single Symbols

The atoms of formulas are symbols. Drawing symbols appropriately for formulas involves specific technicalities. In Swing each symbol is sized in two layers.

The first layer is specified by the font-metric. The height of the layer is the maximal height for all symbols. In mono-spaced fonts every symbol has its own width. We call this layer “outer layer”. The second layer, called “inner layer”, is calculated from the maximal width and height of the vector graphic for each symbol. As an aside there should be mentioned, that there is a third layer, which is the maximal space for each character, but for the editor this is never used. Fig. 4.7 shows inner and outer layers for four symbols. There are two different classes of symbols for aligning several of them.

**Alignment of symbols** requires two concepts: the coordinate system introduced at the beginning of section 4.2 and layers introduced above. The coordinate system is already incorporated in the font and thus involved only implicitly.

Alignment is different for identifiers and for operators. Identifiers consist of letters, all of which “sit” on the baseline such that also descenders are as expected in section 2.2.4 for reading text. So Swing relives the programmer
The alignment of operators goes along the midline as mentioned in section 2.2.4. Thus assembling elements of formulas require specific care in order to have the plus in $a + b$ and the fraction bar in $\frac{a}{b}$ exactly on the midline. Respective code is found in CalcUtil.

**Interpolation and antialiasing** makes symbols more smooth which can be seen in Fig. 4.8. Swing supports interpolation by simply configure rendering this way:

```java
new RenderingHints(
    RenderingHints.KEY_TEXT_ANTIALIASING,
    RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
```

The effect of these few lines of code is important for readability of formulas, in particular if the font size is small:

![Without Interpolation](symbol_without_interpolation.png)

![With Interpolation](symbol_with_interpolation.png)

**Figure 4.8:** Compare a symbol with and without interpolation.

### 4.2.2 Expressions without Vertical Alignment

As mentioned above, assembling formulas from atomic boxes requires care. The simpler case is considered first, horizontal alignment as traditionally
4. Implementation of a Prototype Editor

<table>
<thead>
<tr>
<th>class identifier</th>
<th>( \text{\LaTeX} )</th>
<th>implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>horizontal</td>
<td>+ · − =</td>
<td>yes</td>
</tr>
<tr>
<td>fraction</td>
<td>( \frac{a}{b} )</td>
<td>yes</td>
</tr>
<tr>
<td>exponent</td>
<td>( a^b )</td>
<td>yes</td>
</tr>
<tr>
<td>prefix</td>
<td>( \sin, \cos, \tan (-a) )</td>
<td>yes</td>
</tr>
<tr>
<td>list</td>
<td>( [a, b, c] { n . n &gt; 1 } (a, b) )</td>
<td>partially</td>
</tr>
<tr>
<td>matrix</td>
<td>( (a \ b) )</td>
<td>no</td>
</tr>
<tr>
<td>spec-op</td>
<td>( \int x^2 dx \oint_C Fds \int_a^b x^2 dx )</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 4.1: Layout classes for operators.

used for the operators +, · and −. Given the example \( a + b \), the respective AST is:

\[
\text{Appl}(\text{Constant}("+"), \text{Variable}("a"), \text{Variable}("c"))
\]

Rendering this formula works as follows: First the box for \( a \) is calculated. The position of + is shifted right by the width of the \( a \)-box. Then the +-box is calculated and the width extended with additional space specific for + as show in Fig.4.9.

![Fig. 4.9: How to calculate the box for \( a + b \).](image)

The box enclosing the whole formula gets its width from adding the enclosed boxes and its height from the maximum of the enclosed boxes.

4.2.3 Expressions with Vertical Alignment

Readability of formulas is significantly improved by extending sequential lines to a second, vertical dimension as already mentioned in section 2.2.3 and section 2.2.6. The design identified several layout classes in section 3.5. The implementation provides the classes shown in Tab.4.1.

The table shows, that only the most important operators have been implemented. Another challenge not yet tackled, is to parameterise the classes
4. Implementation of a Prototype Editor

such that math authors can extend the classes without touching the editor’s code.

4.3 Dynamic Aspects

This section describes how the components introduced in section 4.1.1 interact with each other. In particular, section 4.3.1 explains calculation and rendering of a formula, section 4.3.2 explains the effects of mouse movements over a formula and section 4.3.3 explains editing of a formula via keyboard.

4.3.1 Interactions between Components

Swing performs rendering by use of `paint` methods, which in turn is called by every `repaint`.

Such a `paint` method is implemented for an `AstComponent`, the `JComponent` used for embedding a formula in the front-end’s Swing machinery. This `AstComponent.paint` takes an argument of type `Graphics` which is able to draw in Swing, if given appropriate data.

Appropriate data are a formula represented as an AST, so an AST is an argument of `AstComponent`. Within paint, boxes are calculated according to 4.2 by use of `CalcUtil`. In a first step for each symbol width and height are stored in `Settings.ast_Stringbounds`.

```scala
1 var ast_Stringbounds = Map.empty[String, (Rectangle, Rectangle, Rectangle)]
2 def fillStringbounds(ast: Ast, g: Graphics2D) {
3   fillStringboundsRec(Variable("x"), g)
4   fillStringboundsRec(Variable("x"), g)
5   fillStringboundsRec(Variable("i"), g)
6   fillStringboundsRec(Variable("xx"), g)
7   fillStringboundsRec(ast, g)
8 }
9 def fillStringboundsRec(ast: Ast, g: Graphics2D) : Unit = ast match {
10   case Constant(str) => {
11     val op = XSyntax.isa_ast(str)
12     g.setFont(new Font("CMCSC8", Font.PLAIN, CalcUtil.fontsizeOf(0)))
13     val RectLv0 = getStringBounds(g, op, 0f, 0f)
14     g.setFont(new Font("CMCSC8", Font.PLAIN, CalcUtil.fontsizeOf(1)))
15     val RectLv1 = getStringBounds(g, op, 0f, 0f)
16     g.setFont(new Font("CMCSC8", Font.PLAIN, CalcUtil.fontsizeOf(2)))
17     val RectLv2 = getStringBounds(g, op, 0f, 0f)
18     ast_Stringbounds = ast_Stringbounds + (op -> (RectLv0, RectLv1, RectLv2))
19   }
20   case Variable(str) => {
21     g.setFont(new Font("CMMI12", Font.PLAIN, CalcUtil.fontsizeOf(0)))
22     val RectLv0 = getStringBounds(g, str, 0f, 0f)
23     g.setFont(new Font("CMMI12", Font.PLAIN, CalcUtil.fontsizeOf(1)))
24     val RectLv1 = getStringBounds(g, str, 0f, 0f)
25   }
26 }
```
In a second step, `CalcUtil.assembleBoxes` assembles boxes according to `Settings.ast_Stringbounds`. After each successful calculation of a box in `CalcUtil.assembleBoxes`, a callback is performed (the purpose of the callback is explained later). Boxes for `Constant` and `Variable` are calculated immediately. `Appl` describing composed formulas are implemented with one or two arguments, which is conform to most of the layout classes (see section 3.5). `Appl` with more arguments are handled by the `map` function such, that a horizontal sequence is created.

The final step `DrawBox.draw` draws the whole AST recursively descending through all the sub-ASTs.

### 4.3.2 Mouse Events on the EditorPanel

For sighted people the mouse is considered the main tool for pointing at certain sub-terms in the new editor. Each mouse event over a formula triggers a repaint (!), so the display of the formula can be changed dynamically. There are three kinds of mouse events, `mouseDragged`, `mouseDown` and `mouseMoved`, each connected with a specific functionality:

**Selection of boxes** is connected with `mouseDragged`: drawing a rectangle over a certain area with the mouse causes identification of the respective box under the mouse by `AstComponent.mouseDragged`. Identification uses the four edges of the box. The movement creates a list of boxes, for which the smallest parent in the AST has to be found — see `AstComponent.findHighestBoxOf` below.

```scala
1  def findHighestBoxOf(rootbox: DrawBox, markedBoxes : List[DrawBox]) : Option[DrawBox] = {
2    if (markedBoxes != null && rootbox != null && rootbox.children != null) {
3      if (markedBoxes.forall(x => Box.Contains(rootbox, x))) &&
4        ! rootbox.children.exists(x => markedBoxes.forall(y => Box.Contains(x, y)))
5        ) {
6          return Some(rootbox)
7        } else if (markedBoxes.forall(x => Box.Contains(rootbox, x)) &&
8          rootbox.children.exists(x => markedBoxes.forall(y => Box.Contains(x, y)))) {
9            for( box <- rootbox.children) {
10              findHighestBoxOf(box, markedBoxes) match {
11                case Some(box) => return Some(box)
```

4. Implementation of a Prototype Editor
4. Implementation of a Prototype Editor

```java
    case None => {}
    }
    }
    return None
}
```

Given the formula $a + b \cdot x$ represented by this AST

```
Appl(Constant("+"),
    Variable("a"),
    Appl(Constant("*"), Variable("b"), Variable("c")))
```

selecting $a$ and $c$ gives as smallest parent in the AST

```
Appl(Constant("*"), Variable("b"), Variable("c"))
```

and the box of this parent is displayed `AstComponent.findHighestBoxOf`. Thus the above selection results in this AST according to section 3.4.3:

```
Appl(Constant("+"),
    Variable("a"),
    Appl(Constant("BOX.1"),
        Appl(Constant("*"), Variable("b"), Variable("c"))))
```

And Fig.4.10 on p.54 shows the respectively rendered formula.

![Figure 4.10: The above selection results in the rendering below.](image)

**Set the cursor** into a formula starts editing of specific parts of a formula: if `<Ctrl>+mouseDown` is done somewhere on the `EditorPanel`, the respective part of the formula is marked by the cursor. The cursor’s underline indicates the range for editing. Given the formula $aaa + bbb$ as an example, for different parts of the formula can be with respect to the position of the mouse, see Fig.reffig:four-selections. The mouse click notes the coordinates on the `EditorPanel`; these are used for searching matching boxes and calculating the respective AST (probably a parent in case no.4 below) by `AstComponent.mouseDown`. This AST gets marked by `CURSOR` according to section 3.4.3.
1. \(aaa\) if the mouse is placed somewhere in this range. Note, that the cursor can *not* be set to somewhere in the middle of \(aaa\) (for instance for editing a single character in a long identifier or a large number). So there are possibilities for later improvement.

2. + if the mouse is placed sufficiently close.

3. \(bbb\) if the mouse is placed somewhere in this range.

4. the whole formula \(aaa + bbb\) if the mouse is placed between \(aaa\) and + or between + and \(bbb\).

The rendering, as intuitive as shown above, is created from a CURSOR placed appropriately in the formula’s AST. So ongoing interaction, still without editing anything, updates the formula’s datastructure (as done analogously done with BOX as described in 4.3.2) as follows:

\[
\text{Appl}(\text{Constant}("+"), \text{Variable("a")}, \text{Variable("b")})
\]

is updated to

\[
\text{Appl}(\text{Constant}("+"),
\text{Appl}(\text{Constant}("CURSOR"), \text{Variable("a")}),
\text{Variable("b")})
\]

when the cursor is placed on “a”.

**Display boxes** in formula is connected with *mouseMoved*. First, in this last case the AST is never changed (no BOX, no CURSOR etc). So the way used in the previous two cases is not possible (i.e. just re-painting the whole AST). Highlighting of a proper sub-term under the mouse position is done by the following code in *AstComponent.paint*:

```scala
1 override def paint(g: Graphics) = {
2   ...
3   box = CalcUtil.assembleBoxes(ast, (box: DrawBox) => {
4     if (EventUtil.foreachBoxFunction != null) {
```

![Figure 4.11: Four selections according to mouse position.](image-url)
CalcUtil.assembleBoxes gets a callback as described in section 4.3.1, which works on each box as described by the following code:

```scala
var foreachBoxFunction : (Graphics, DrawBox) => Unit = null

def drawBoxAt(p: Point) = {
  foreachBoxFunction = EventUtil.doInBox(p, (g, box) => {
    Box.draw(box, g, false)
  })
}
```

`foreachBoxFunction` is used by `drawBoxAt` for highlighting a box at a position `p`, i.e. the mouse position. Now the whole functionality follows from the fact, that `AstComponent.paint` is called upon each `mouseMoved`.

### 4.3.3 Interactions during Editing

Effects of interaction on a formula with the mouse have been described in section 4.3.2 on p.53 and on p.54. Two variants of editing are presented analogously to section 4.1.1: (1) via a `JTextField` or (2) via keyboard events immediately. Both variants were implemented; here follows a description of experiences gained during implementation of both variants.

**Editing via `JTextField`** inherits all the built-in features from Java Swing: setting the cursor, updating elements of a string, cutting out substrings, copy & paste, etc). Implementation assigns a “null” *Layout* to `JTextField` which can be shifted to the location of the cursor. This field is set visible during editing and set invisible during rendering.

Although this variant appears convenient and straightforward for implementation, there are disadvantages with respect to usability: The datastructure of a string (i.e. an array of characters) is very different from the datastructure of a formula as discussed so far: Within a string the structure of a formula can easily be violated. For instance, from the formula “\(a \cdot (b+c)\)” the substring “\(\cdot (b+c)\)” can be cut out without efficient means to inhibit such an operation within the realm of strings.

Once a string is input, it needs to be parsed — and handling “\(\cdot (b+c)\)” reasonably would create nightmares. Experiences also showed, that the `JTextField` appears alien in context of a formula rendered in two dimensions.

This variant has been pursued until changesets [https://intra.ist.tugraz.at/hg/isac/rev/a2a220b0996b](https://intra.ist.tugraz.at/hg/isac/rev/a2a220b0996b) and then removed in favour of the variant below.
4. Implementation of a Prototype Editor

Editing via keyboard events is the other variant. This introduces the burden to take care of all key events. But the major goal of accessibility and inclusion, discussed in section 2.1, could not be achieved elegantly, see for instance UR.2.17, UR.2.20, UR.2.21, and UR.2.23.

Restricting the view to the EditorPanel, editing is started as described by “set the cursor” in section 4.3.2. After editing has started, key events are handled according to the state transition diagram in Fig.4.12. The diagram is read as follows:

![State transition diagram for key strokes on the editor.](image)

The two states **Number** and **Identifier** are related to Ast.Variable and may continue input of a longer number or identifier. The respective AST is either empty

\[
\text{Appl}(\text{Constant("CURSOR")}, \text{Constant("GAP")})
\]

or continues input on this kind of AST:

\[
\text{Appl}(\text{Variable(str)}, \text{Constant("CURSOR")})
\]

The other states are not adorned with cycles in Fig.4.12 and thus left by the subsequent key stroke. The state **Start** attracts all keys and immediately switches on to respective states.

---

3Replacing a whole sub-term following “selection of boxes” is not yet implemented.
The state **Operator** is related to a lookup of the map `XSyntax.ast_const` for fetching the appropriate AST structure. The parent of the selected AST is replaced with this structure. Into the structure the first argument of the operator is inserted and the cursor set to the second argument. Further arguments, if they exist, get `GAPs` as place holder according to SR.2.5. Fig 4.13 shows the steps required to input the formula $a + \frac{b}{c}$ from scratch.

![Figure 4.13: Create the formulas by typing “a+b/2”.

And Fig 4.14 on p.59 shows the steps required to input the formula $\frac{a+b}{c}$ from scratch. Note step five `<↑>`, which changes the scope of the cursor and applies multiplication to both elements of $a + b$.

There is a difference between

\[
\text{Appl}(\text{Cursor, ast})
\]
**Figure 4.14:** Create the formulas by typing $\frac{a+b}{c}$.

and

\[ \text{Appl} \text{ast, Cursor} \]

The former provides an insertion **before** the existing (sub-)term, the latter an insertion **after** the existing (sub-)term.

**Editing of operators** is different from editing numbers and identifiers beyond what is shown in Fig.4.12. The most simple case is, if the number of arguments is equal; see this example:
Appl(Constant("+"), Variable("a"), Variable("b"))

↓

Appl(Constant("/"), Variable("a"), Variable("b")))

A + is replaced by a /, which changes the rendering considerably; however, the structure of the AST remains as is.

More difficult is the situation, when the number of arguments is different, for instance, when sin is replaced by +:

Appl(Constant("sin"), Variable("a"))

↓

Appl(Constant("+"), Variable("a"), Constant("GAP"))

Then Constant("GAP") is a place holder waiting for input (while the cursor is also set to this position automatically). This mechanism generalises for an arbitrary number of arguments, as long as the number for the replacing operator is larger: keep the existing arguments and provide GAPs for the further ones. But what to do in the other way round is not yet clear.

Appl(Constant("+"), Variable("a"), Variable("b"))

↓

Appl(Constant("sin"), Variable("a"))

What should happen with Variable("b")? Here another requirement is raised, the requirement of undo.

4.4 Test Environments

After clarification of architecture and top level design, as described in chapter 3, implementation proceeded bottom up. Thus the first phase of implementation was concerned with rendering a single formula, the second with rendering formulas in a calculation and the third was concerned with input of formulas. The latter phase could re-use the test environments from the former two phases.

4.4.1 Test Environment for a Single Formula

The \texttt{\LaTeX} prototype still needs to be started from within the IDE, with separately launching four modules (see section 2.4) running in different virtual machines; and then many interactions are required to start a calculation and
come to a formula finally — doing this during experiments with rendering a single formula would be annoying. Thus there is a specific test environment for incremental coding for a single formula.

*TestEditorPanel* is such an environment which implements Junit Test-Case. This test is not included in ISAC’s test suite, because it launches a JFrame which cannot be checked by an assertion (due lack of respective tooling in ISAC at the time of working on this thesis).

### 4.4.2 Test Environment for Working on a Calculation

The goal of the thesis was to integrate the prototype editor into ISAC’s prototype. In order to avoid time consuming launch of ISAC and selection of a calculation, as mentioned above, a minimal test setup is required. Such a minimal setup turns out to be surprisingly complicated, but this thesis could re-use a test environment already implemented by [8] p.38 ff. Re-use of the existing test was connected with some improvements of the code.

The respective test environment is *TestWorksheetForMawen*; it requires three mocks:

1. *MockWorksheetDialog* mocks the WorksheetDialog, which handles all interactions on the Worksheet and also display of formulas (among others).

2. *MockDialogIteratorSIMPLIFY* mocks the DialogIterator, which iterates over a CalcTree according to interactions on the Worksheet. In the test case, the iterator runs over a CalcTree explained next.

3. *MockCalcTreeSIMPLIFY* contains fixed formulas to be displayed in the Worksheet within the test setup.

The few changes in the Worksheet and respective helper classes required to integrate the new editor were checked by ISAC’s test suite.
Chapter 5

Summary, Conclusion and Future Work

This thesis is assigned for prototyping a formula editor in \( \mathcal{LSAC} \), a prototype for a new kind of educational software, which builds upon the rapidly evolving TP, Isabelle. So there cannot be final results of the thesis. Moreover, designing and implementing an editor, which is accessible for visually impaired students as well as appropriate for inclusive learning (i.e. collaboration with sighted students), is a task which cannot be accomplished by one person.

So this final chapter is separated into three sections: first there is a register accounting what has been done and what could not be done accompanied with experiences from development. Second are conclusions drawn from the work actually done, and third there is a preview to future work.

5.1 Summary

This thesis is a feasibility study on a formula editor based on a theorem prover. The study involved much preliminary work on accessibility and inclusion section 2.1, on standards in formula presentation (\( \LaTeX \) and MathML) section 2.2, on the state of the art of front-ends in theorem provers (see section 2.3), on the predetermined architecture and code base for embedding the editor (see section 2.4) as well as predetermined user-requirements and software-requirements (see section 2.5) for a formula editor.

The prototype of the editor exploits all features of Java Swing which make trials with editing convenient: a formula can be moved by mouse and resized by mouse-wheel within the EditorPanel. The size dynamically determined for one formula is pushed to all other formulas of a Worksheet as soon as editing is quit. Although the editor reports the height of an edited formula, the JTrees’s renderer does not yet recognise varying heights. All
this behaviour is experimental and might be changed eventually.

The register what has been done and what could not be done is organised along the points integration into \( \mathcal{I} \mathcal{S} \mathcal{A} \mathcal{C} \), datastructure for rendering formulas, rendering in \( \LaTeX \) quality and finally interaction and editing of formulas. This sequence roughly reflects proceeding from an internal and technical view to what the user views on the surface.

**Integration into \( \mathcal{I} \mathcal{S} \mathcal{A} \mathcal{C} \)** was challenging but brings the benefits of existing data structures for formulas, respective transformations and an elaborated user interface for calculations. Fig. 5.1 shows, how example no.7 in the Example browser is started as an interactive calculation in a Worksheet and how from there underlying knowledge can be retrieved, for instance in the Theory browser.

![Figure 5.1: Interactive simplification visualized with the new editor.](image)

1. **. . . accomplished**
   (a) design and implementation of an interface for an accessible and inclusive editor
   (b) integration into \( \mathcal{I} \mathcal{S} \mathcal{A} \mathcal{C} \) \(2.4 \text{ (UR.2.5)}\) by adopting the \texttt{CalcTreeCell-Renderer} and \texttt{CalcTreeCellEditor}

2. **. . . partially accomplished**
   (a) two levels of feedback are implemented (UR.2.3): feedback by Isabelle is left as is and \texttt{LocalCheck} is only a stub
(b) translation from ASTs (after input) to terms for check by Isabelle/\texttt{ZS4C} is circumvented by submission of strings, which are parsed to terms in Isabelle

(c) braille support (UR.2.20, UR.2.21, (UR.2.22) has a stub in the editor’s interface and creates output on the console

3. \ldots \textbf{not yet tackled}

(a) Isabelle’s user-friendly syntax definitions should be connected with the editor for the purpose of extending the language of mathematics (UR.2.8)

(b) automated generation of rewrite rules for AST-AST-translations from syntax definitions

\textbf{Datastructure for rendering} in Isabelle are annotated syntax trees (AST), however the present interface between \texttt{ZS4C} and Isabelle only transports terms.

1. \ldots \textbf{accomplished}

(a) translated Isabelle’s AST from SML to Scala

(b) implemented translation from Term to AST in Scala

(c) extended Scala ASTs with CURSOR, GAP and BOX, i.e. ASTs hold data for dynamic actions

(d) properties of operators are defined centrally (UR.2.6)

2. \ldots \textbf{partially accomplished}

(a) CURSOR has no position

(b) parentheses according to operator priority (UR.2.16)

3. \ldots \textbf{not yet tackled}

(a) translation from AST to Term

(b) record positions for elements of formulas for the purpose of displaying terms on request

(c) make types visible in the editor by mouse click

i. keep types during translation from Term to AST

ii. additional mouse features in the editor

\textbf{Rendering in \LaTeX quality} establishes the first impression when approaching a formula editor in a software system for mathematics. Fig.5.2 shows the formula from Fig.5.1 in more detail.

1. \ldots \textbf{accomplished}

(a) draw boxes around sub-terms with different colors (UR.2.12)

(b) draw gaps and a cursor (UR.2.10, UR.2.13)
(c) size of sub-terms in fractions and powers is reduced according to respective levels
(d) automated selection of fonts depending on operators or identifiers of variables

2. **...partially accomplished**
   (a) rendering of specific operators
      i. integral works only with two arguments
      ii. unknown operators are displayed prefix with the arguments in a horizontal sequence
   (b) fine tuning of fonts

3. **...not yet tackled**
   (a) unary minus
   (b) keys for navigation cannot be changed (UR.2.17)
   (c) no auditive feedback (UR.2.19)
   (d) selected sub-terms cannot be deleted or replaced
   (e) the demo example “Biegelinien” does not work completely

**User Interactions and Editing** of formulas comprises input from scratch, updating certain elements of a formula and navigation on a formula. Fig.5.3 shows how a formula within a calculation is being edited.

1. **...accomplished**
   (a) key strokes lead to immediate display; for instance, input of + creates two gaps for the arguments
   (b) zooming and shifting of formulas for increased readablilty (UR.2.24)
   (c) input a formula from scratch
      i. addition, multiplication, subtraction
      ii. division, exponentiation

---

**Figure 5.2:** Example of a Simplification with doublefraction.

\[
\text{Simplify} \left( \frac{2}{x + 3} + \frac{2}{x \cdot 3} \right)
\]
(d) update certain elements of a formula (UR.2.1, 2.15)
   i. identifiers
   ii. operator with same number of parameter (e.g. change $a + b$ into $a - b$)
   iii. operator with fewer parameter (e.g. change $a - b$ into $\sin(a)$)
   iv. operator with more parameter (e.g. change $\sin(a)$ into $a - \bigotimes$)
   v. selection of sub-terms by mouse (UR.2.23)

(e) navigation on a formula (UR.2.11, UR.2.14, UR.2.17)
   i. CURSOR
   ii. BOX
   iii. display BOX-content to the console

(f) properties of operators are defined centrally (UR.2.6)

(g) parentheses according to operator priority (UR.2.16)

2. \ldots partially accomplished

(a) input a formula from scratch (UR.2.8, UR.2.9)
   i. copy and paste a whole formula is not included directly but possible
   ii. integral works only with two arguments
   iii. unknown operators were displayed prefix with the arguments in a horizontal sequence

(b) update certain elements of a formula
   i. input of operators only by keys, not by palette (UR.2.7)
ii. variables and numbers cannot be edited character by character, rather they need to be deleted from the end

iii. two levels of feedback are implemented, but *LocalCheck* is only a stub (UR 2.3)

(c) navigation on a formula

i. sub-terms can be selected via mouse click (UR 2.2), but subsequent clicks do not extend up to the next parent term

ii. braille feedback is only a stub (UR 2.20, UR 2.21, UR 2.22)

3. . . . not yet tackled

(a) input a formula from scratch

i. unary minus

(b) update certain elements of a formula

i. selected sub-terms cannot be deleted or replaced

(c) navigation on a formula

i. keys for navigation cannot be changed (UR 2.17)

(d) no auditive feedback (UR 2.19)

(e) the demo example “Biegelinien” does not work completely

Experiences with combining Scala and Java confirmed, that this combination works smoothly – with one exception, which appeared in the development under consideration: In Scala one and the same identifier can denote a class or an object, and Scala can distinguish between these – but Java cannot distinguish. Thus Java prompts the error “...cannot be resolved to a type” in the code, but compilation deals correctly with the situation and eclipse’s package explorer doesn’t indicate this situation as well.

These inappropriate error indications triggered a new point in *ZSCC*’s coding standards: The respective code line must be preceeded by /*err*/ for the reader who scans for red marks at the left margin.

Another inconvenience was caused by eclipse’s build management: editing code does not lead to proper re-compilation frequently – at least not with a comprehensive code base as in *ZSCC* plus the code added by this thesis as shown in Tab. 5.1 on p.68 (This table does not show the updates required for introducing the interface for the editor into *ZSCC* and adaptations to the test environment). So a complete re-build was required frequently, but this takes about ten seconds on a high-end workstation. In order to save time, another way out was to just cut respective code and paste it again.
Table 5.1: Lines of code added by the prototype editor.

<table>
<thead>
<tr>
<th>purpose</th>
<th>program language</th>
<th>lines of code (LoC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>production code</td>
<td>Java</td>
<td>246</td>
</tr>
<tr>
<td></td>
<td>Scala</td>
<td>2.118</td>
</tr>
<tr>
<td>test code</td>
<td>Java &amp; Scala</td>
<td>1.114</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td>3.478</td>
</tr>
</tbody>
</table>

5.2 Conclusions

The feasibility study for a TP-based, accessible and inclusive formula editor can be considered successful. The study provided the following insights:

The combination of Scala and Java is appropriate for constructing an editor: Java provides access to Swing, a powerful library since it has adopted the concepts of \LaTeX. And Scala’s `match` is perfect for operating on tree-structures like those of formulas. Thus integration into \Isabelle was straightforward.

This insight paves the way for a mid-term alternative to an \Isabelle front-end on tablets and handhelds \(^1\) – an editor implemented in Scala and Java Swing can also be integrated into Isabelle/jEdit, which is implemented in the same programming languages. \(^2\)

Editors can combine \LaTeX-quality and TP-requirements: TP increases demands on a formula editor but also provides appropriate technologies for constructing formula editors. The demands are indicating types for all elements of a formula and providing extensibility for the language of mathematics. The prototype demonstrates, that this can be done by mouse click on elements like in Isabelle/jEdit. The latter is less straightforward, but respective layout classes have been identified.

This insight allows to envisage development of \Isabelle towards a novel mathematics tool for engineering education.

Development efforts can be estimated when striving for a professional editor with formulas rendered in \LaTeX-quality, with accessibility for visually impaired students and including cooperation with sighted students, and with convenient input even without menus. Respective development can follow:

\(^1\)\Isabelle for such devices will come sooner or later, at least when respective technologies have settled

\(^2\)Then the challenge remains, to introduce session management, dialogue guidance and Lucas-Interpretation to Isabelle/PIDE.
the design and the architecture provided by this thesis; efforts are estimated with about one Man Year, for details see below.

This insight provides prerequisites for planning future development of \( \mathcal{ZSAC} \) towards a professional tool for engineering education.

5.3 Preview to Future Work

Future work has been identified in great detail by the feasibility study and the prototyping efforts of this thesis. Before approaching the list of details below, it is worth to mention that extensive testing of the formula editor is advisable to start future development with. Such testing has been dropped, because the editor’s distinguishing features of accessibility and inclusion could not be tested by lack of connection to a braille display (as mentioned in section 5.1 integration into \( \mathcal{ZSAC} \), no 2c).

The following enumeration collects all points which have been mentioned as not (completely) accomplished in section 5.1.

1. ad: Integration into \( \mathcal{ZSAC} \) on p.63
   (a) simple checks of a formula locally (without calling Isabelle): matching parentheses, etc.
   (b) extend AST-AST-translations to the full range of mathematical operators (\( \frac{d}{dx}, \int_a^b x^2 + x + 1 \, dx \), ...) and data structures (tupels, lists, matrices, etc)
   (c) implement the connection for the braille display and respective standards for presentation of mathematics to visually impaired persons
   (d) connect Isabelle’s user-friendly syntax definitions with the editor for the purpose of extending the language of mathematics (UR.2.8)
   (e) automated generation of rewrite rules for AST-AST-translations from syntax definitions

2. ad: Datastructure for rendendering on p.64
   (a) provide the other direction of translation: from AST back to Term and submit the latter to Isabelle (instead of strings to be re-parsed)
   (b) record positions for elements of formulas for the purpose of displaying types on request
   (c) make types visible in the editor by mouse click
      i. keep types during translation from Term to AST
      ii. additional mouse features in the editor
3. ad: Rendering in \( \LaTeX \) quality on p.64
   (a) rendering of operators \( \mathbf{2.8} \) according to UR.\( \mathbf{2.9} \)
   (b) unary minus
   (c) fine tuning of fonts: \( \LaTeX \) uses various fonts; the prototype distinguishes only between font for letters, for numbers and for operators; but much work is still open here: small fonts in lower levels must be more bold, the binary \( - \) sign is differen from \( \LaTeX \), etc.
   (d) no auditive feedback for guiding navigation (UR.\( \mathbf{2.19} \))

4. ad: User Interactions and Editing on p.65
   (a) re-consider design questions, which became clear during prototyping:
      i. How is it possible to point to a formula?
         A. by mouse click
         B. by rectangular selection by mouse
         C. by arrow keys
         And how do these points correlate with respect to sighted people and visually impaired people?
      ii. What can be done at the position pointed to?
         A. focus a sub-term (by viewing on screen or an braille). But what is meant by focus in case of \( \mathbf{4(a)iA}\)?
         B. delete the sub-term and \textit{always} replace by “GAP”?
         C. edit: How different are keystrokes to be handled with respect to the different answers to \( \mathbf{4(a)i}\)?
         D. switch to a level up or down in the formula’s tree or scan elements on one and the same level
      iii. how compelling should the term structure be presented to users? Presently \( (a+b) \cdot (c+d) \) can be input without touching parentheses, just by selecting sub-terms accordingly — should/can this be the only way to input complicated formulas?
   (b) re-design the state-transition diagram in Fig.\( \mathbf{4.12}\) according to answers to the above questions in \( \mathbf{4A}\)
   (c) re-consider \textbf{mouse actions} on a formula: entering edit mode by triple click is clumsy. Changing to single click requires intricate changes in \textit{Worksheet} interaction (concerning \textit{activeFormula} etc.) — this issue became visible by experimenting with the editor, but concerns features \textit{outside} the editor.
(d) re-consider handling of marked sub-terms: marking is separated from editing, which seem to confuse users; marked sub-terms can neither be deleted nor replaced; also the use of <F2> for replacement appears clumsy.

(e) re-consider parentheses: input of “(” is not considered in the state transition diagram Fig.4.12. However, \((a+b)^c\) can be input, if the cursor is extended (by arrow keys) to \(a+b\) as described in Fig.4.14.

(f) re-consider dynamic change of font size: this turned out helpful for complicated formulas on the Worksheet. However, presently the changed size is adopted by all open Worksheets. This is unhandy, but there are other issues like personal settings as opposed to system setting (for workstations or hand-holds).

(g) an <Undo> function is required; Is it possible to use a particular Java functionality here?

(h) keys for navigation cannot be changed (UR.2.17)

The above list together with the experiences made during this thesis lead to a final judgment as follows:

- LSAC’s code base and the one of Isabelle (which serves as source and model for the editor’s AST-AST-transformations) is complex and voluminous such, that time required for getting acquainted will exceed the scope of a thesis.

- The work required for development towards a professional tool comprises much coding and little theoretical work such that the work seems not appropriate for an academic thesis.

Thus further development of the prototype editor appears inappropriate for student projects.
References

Literature


References


### Online sources