Formula Editors for TP-based Systems. State of the Art and Prototype Implementation in *ISAC*

Marco Mahringer, BSc

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Software Engineering

in Hagenberg

im September 2017

© Copyright 2017 Marco Mahringer, BSc

This work is published under the conditions of the *Creative Commons License Attribution–NonCommercial–NoDerivatives* (CC BY-NC-ND)—see http://creativecommons.org/licenses/by-nc-nd/3.0/.

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, September 30, 2017

Marco Mahringer, BSc

Contents

Kurzfassung

Diese Thesis bearbeitet Neuland in drei verschiedenen Richtungen. Erstens in Richtung zweidimensionaler Formel-Darstellung und Visualisierung für Systeme basierend auf (Computer) Theorem Proving (Abkürzung "TP" im Titel). Zweitens beschäftigt sich mit "accessibile" und "inclusive" Systeme. Die Dritte in Richtung beschäftigt sich mit der Integration von Scala mit Java Swing im ISAC Prototypen.

Die Relevanz der ersten Richtung ergibt sich wie folgt: TP-Systeme selbst sind immer noch Werkzeuge für Experten, die eine schnelle Eingabe bevorzugen und noch nicht darauf Wert legen, Formeln schön dargestellt zu sehen. Jedoch ist TP-basierte Software dabei, in die Arbeitsumgebung von Ingenieuren einzudringen. Und Ingenieure erwarten Präsentationen von mathematischen Formeln im üblichen zweidimensionalen Format. Die Stärken von TP sollten beibehalten werden: insbesondere der Zugriff auf die Typen aller Elemente einer Formel per Mausklick sowie die Verknüpfung mit entsprechenden Definitionen, wie sie aus den IDEs der Programmierer bekannt sind.

Um für Ingenieurs-Anwendungen bereit zu werden, hat der Proof-Assistent Isabelle sein Front-End benutzerfreundlich gemacht. Isabelle ist das Logik-basierte Back-End für ISAC, den Prototypen einer neuen Generation von TP-basierte Lernsoftware und somit das System der Wahl für das Prototyping eines Editors in dieser Thesis.

Die Identifizierung von Schnittstellen für einen Editor in ISAC ist durch die schnelle Entwicklung von Isabelle und durch die umfangreiche und unorthodoxe Dokumentation eine besondere Herausforderung . Isabelles GUI nutzt die Programmiersprache Scala, also muss diese Thesis sich mit dieser Sprache auseinander setzen. Hilfreich für das Prototyping ist, dass Isabelle sich weitgehend an Standard der Software-Technologie hält; insbesondere für die Umwandlung von Formeln aus dem internen Format in ein Präsentations-affines Format. Dabei werden Technologien aus dem Compilerbau verwendet, wie zum Beispiel annotierte Syntaxbäume und Transformation durch “rewriting” auf diesen Bäumen.

Die zweite Richtung, also der "accessibile" und "inclusive" Bereich, wurde durch eine Diplomarbeit an der FH Hagenberg, Fakultät für Softwaretechnik, im vorangegangenen Jahr perfekt vorbereitet. Die Auswertung dieser Arbeit gibt klar vor, wie der Editor in dieser Thesis entworfen werden soll:

Einerseits können sehbeeinträchtigte Personen nur Strings über ein Braille Display lesen, andererseits werden komplizierte Formeln als Strings unverständlich. Um diesen Widerspruch aufzulösen, schlug die vorangegangene Thesis vor, Formeln als Bäume darzustellen, in denen mittels Pfeiltasten durch die Teilformeln (dargestellt als *kürzere!* Strings) navigiert wird. Diese Darstellung hat den Nachteil, dass sie für Sehende vorerst unverständlich ist und somit der “inclusion” widerspricht.

Die vorliegende Thesis stellt eine elegante Lösung vor, die die bevorzugte Formelpräsentation für beide, für blinde wie sehbehinderte Studierende, aufrechterhält: Teil-Formeln auf der Braille (für blinden Studierende) werden eindeutig mit Rechtecken auf der zweidimensional gerenderten Formel (für sehende Studierende) hervorgehoben. Die Integration der Treiber für die Braille samt zugehöriger Standards bleibt späteren Arbeiten überlassen.

Die dritte Richtung beschäftigt sich mit der Kombination von Scala und Java Swing. Diese Kombination sowie die Integration des Editors in ISAC kostete den größten Teil an Zeit und Energie dieser Thesis. Die Kombination ist als grundsätzlich machbar bekannt, wurde aber noch nie zuvor bei der Implementierung eines Formel-Editors ausprobiert. Die Kombination hat sich letztendlich als einfacher herausgestellt als erwartet: Scala’s `match` ist praktisch für das Scannen der Formelstruktur, und in Swing ließ sich die Box-Technologie von LaTeX einfach nachbauen.

Obwohl die Codebasis von ISAC umfangreich ist, nachdem mehr als dreißig Thesen zum Code beigetragen haben, war die Identifizierung einer Schnittstelle für den Editor einfach und entsprechende Änderungen im Code minimal. Der ISAC-Prototyp kann nun mit dem neuen Prototyp des Editors präsentiert werden.

Alle Prinzipien von LaTeX sind im Prototyp-Editor implementiert: abnehmende Schriftgröße nach Ebenen der Teilformeln, jeweilige Anpassung von Leerzeichen usw. Der Prototyp erlaubt die Eingabe von neuen Formeln sowie die Bearbeitung vorhandener Formeln. Die praktische Erfahrung mit dem Editor zeigt jedoch, dass die Benutzbarkeit für professionelle Anwender (einschließlich Studenten) noch erhebliche Anstrengungen erfordern wird.

Die These 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. Weiter 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 ISAC.

The relevance of the first direction is given as follows: TP systems themselves are still tools for experts who prefer quick input and do not yet take care of nicely rendered formulas. However, intrusion of TP-based components into workbenches for engineers can be foreseen. And engineers expect mathematical formulas presented in a two-dimensional format – while the strengths of TP should be maintained, in particular the access to the types of all elements of a formula via mouse click on the rendered formula, as well as linking to respective definitions as known from programmers' IDEs.

Making TP-based systems ready for engineers' workbenches appears just in time particularly for the proof assistant Isabelle, which has the most user-friendly front-end. And Isabelle is the logic-based back-end for ISAC, the prototype for a new generation of TP-based educational systems and the system of choice for prototyping an editor within this thesis.

Identifying interfaces for an editor with Isabelle is a specific challenge due to Isabelle's rapid development and the highly elaborate but unorthodox documentation. Isabelle's GUI uses the programming language Scala, so this thesis has to deal with this language. Fortunate for prototyping with Isabelle is, that is uses standards of software technology; in particular, for conversion of formulas from the internal format to a presentation-affine format it employs the standard technologies from compiler construction, annotated syntax trees and translation by rewriting on these trees.

The second direction towards accessible and inclusive editors has been greatly prepared by a thesis at FH Hagenberg, faculty of software engineering, in the preceding year. The evaluation within this thesis gave a clear direction how to proceed:

Visually impaired people can only read strings via their Braille display. Complicated formulas become incomprehensible as strings, but flexible ac-

cess (via arrow keys) to subterms as elements (again represented as strings) of a tree can make the structure of a formula comprehensible. This has been shown by the preceding thesis (while other techniques like auditive information has been shown inefficient).

The preceding thesis' suggestion for presenting the tree also to a sighted student for the sake of collaboration in inclusive educational settings (i.e. with collaboration between a blind and a sighted student on one and the same formula), this suggestion has also be abandoned. This subsequent thesis provides 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). Integration of drivers for the Braille together with implementation of respective mathematics standards is left to subsequent development.

The third direction, towards combining Scala with Java Swing in code for a formula editor and integration into ISAC, took the major time and energy in this thesis. Combining Scala with Swing is known as feasible in principle, but that has never been tried out in implementing a formula editor. The combination was more pleasant than anticipated: Scala's `match` is convenient for scanning the structure of a formula, and Swing was straight forward since it has perfectly adopted the box technology from LaTeX.

Although ISAC's code base is already comprehensive, and although more than thirty theses had contributed to the code, identification of an interface for the editor was simple and respective changes in the code were minimal. The ISAC-prototype now can be presented with the new prototype of the editor.

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. Practical use of the editor, however, shows that considerable efforts are still required towards a professional editor for professional users (including students).

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 too..*

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?

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 [1] 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

[1] The ISAC-project's research and development goes on since more than one decade http://www.ist.tugraz.at/isac/History driven by in an interdisciplinary team http://www.ist.tugraz.at/isac/Credits.

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 ISAC project this feature is part of the concept "transparent systems".

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 ISAC-project for building "systems that explain themselves" [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 [6]. These two requirements are met unsatisfactorily by available formula editors. Work on these requirements in the thesis is supervised by the "Institute 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: ISAC plans to make 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 reinvent 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 ISAC 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 par 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 ISAC prototype, anyway. The ISAC 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 ISAC 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 ISAC's design delegates all reasoning to Isabelle, the question arises, whether these terms are the right data structure for our formula editor, which serves ISAC's front-end (which is not concerned with any kind of reason). Thus these terms and respective transformations also will be studied.

ISAC'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 [2]. The mathematics engine had been developed [3], 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 [4]) 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

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

[3] Development of the mathematics engine Isabelle/ISAC is still separated into the repository https://intra.ist.tugraz.at/hg/isa, while the thesis under consideration is reflected by the repository https://intra.ist.tugraz.at/hg/isac.

[4] http://www.ist.tugraz.at/isac/Credits

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 §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 somewhere in the preliminaries stated at the beginning in §2. So the reader 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 Sect.§2.2 are given by §2.2.1 various types of editors, by §2.2.2 MathML and last not least by §2.2.3 LaTeX. §2.3 briefly introduces the state of front-ends for (computer) theorem provers (TP) with focus on Isabelle: §2.3.1 explains reasons for the transitional state of TP, §2.3.2 presents Isabelle's GUI and looks into the machinery behind the scenes as far as relevant. The last prerequisite is the ISAC prototype in §2.4; after a brief introduction of it's aims in §2.4.1, it's initial state of the front-end is described in §2.4.2 and requirements for the formula editor are given in §2.5.

Chapter §3 comes to architectural and design considerations, which start with the issue of re-use in Sect.§3.1: re-use of Isabelle's machinery for formula transformation in §2.3.3 and integration into ISAC's existing front-end in §3.1.2. Integration is discussed more specifically in a separate section §3.2 and conforms to an interface specified in §3.3. Concern of design are also datatypes specific for editors and general considerations about visual representation of formulas in §3.5.

§4 describes the implementation of the editor prototype. §4.1 addresses code structure, integration and choice of programming language. §4.2 explains how symbols are drawn by Java Swing, §2.2 shows how a formula is rendered and §4.3 describes interaction within editing a formula. Implementation required specific test setups addressed in §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 §1.2 mentioned reasons for the large number of preliminaries. Here details are given, 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 [6]. "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 [6] 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 $\frac{1+2\cdot(3+x)}{2\cdot(3+x)}$ 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 [6], 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 high-lighting 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 high-light 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 ISAC.

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 [9]. Beeing 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 ISAC, because ISAC's frontend is not standard like a browser. So these editors fall outside the selection criteria of this thesis.

[1] http://www.dessci.com/en/products/mathtype/

[2] http://www.mathtype.com/en/products/mathflow/default.htm

[3] http://tex.loria.fr/outils/readme-techexpl.html

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 [23].

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 masters thesis [6] 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 [30] 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 <mi>, operators by tag <mo> and numerals by tag <no>:

[4] https://www.mathjax.org/

[5] http://mathcast.sourceforge.net/home.html

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <mi>y</mi>
  <mo>=</mo>
  <msup>
    <mi>x</mi>
    <mrow>
      <mn>2</mn>
      <mi>k</mi>
      <mo>+</mo>
      <mn>1</mn>
    </mrow>
  </msup>
</math>
```

<mrow> collects elements simply in a horizontal sequence, for instance $2k+1$ as shown in Fig.2.1. Other tags are used to determine position, height and width of characters and spaces, and also level and style of sub-terms. An example is exponentiation as shown in Fig.2.1.

$$y = x^{2k+1}$$

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 none 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 LaTeX is input as a string; the specific syntax is intuitive even for complicated formulas and known by heart by most mathematicians:

```
$y=x^{2k+1}$
```

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 §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 §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 §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 [7], in collaboration with Hermann Zapf from the American Mathematical Society (AMS). This font resembles handwriting of a mathematician. The book *Concrete Mathematics* [4] 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 +, · etc are not placed on the baseline but on the

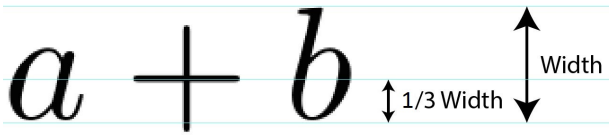

Figure 2.2: Placing of the plus symbol.

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 $X + \frac{X}{X}$

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 \, and \; 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 $x = a + b$

	Operator	Identifier	Number	Relation
Operator	MS	MS	MS	LS
Identifier	MS	SS	SS	LS
Number	MS	SS	SS	LS
Relation	LS	LS	LS	LS

Figure 2.3: Spaces between elements of a formula.

involves spacing as shown in Fig. 2.4.

2.2.6 Character Size in Sub-Terms

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

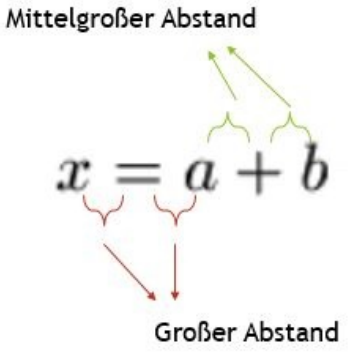

Figure 2.4: Spaces created for string "x = a + b".

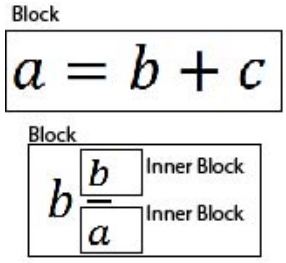

Figure 2.5: Levels of boxes in LaTeX

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.

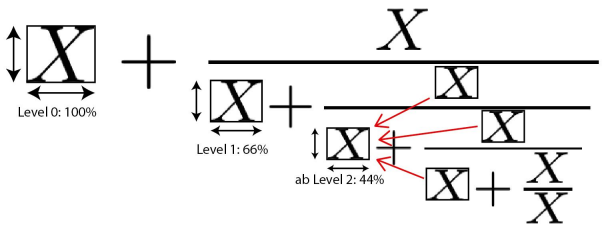

Figure 2.6: Space is reduced by 2/3 for each level, maxlevel = 2.

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" [3]). 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 [22], 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

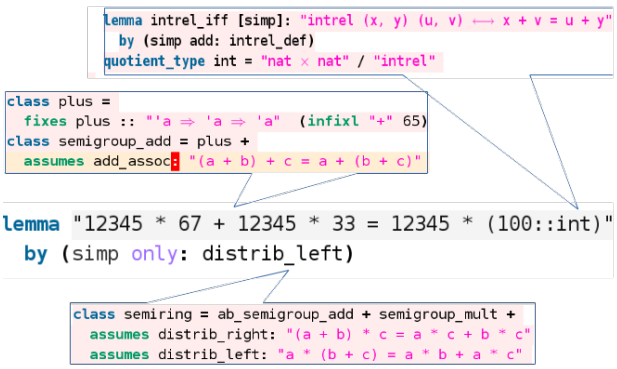

Figure 2.7: Isabelle shows underlying knowledge transparently.

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 seminring), a click on *distrib_left* the theorem of distributivity together with

the proof, that this theorem holds for semigroups (and thus for seminrings 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) [25]. This interface is capable of parallel proof checking [26], linking editor content with definitions, etc. The main application of PIDE uses jEdit [27] 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

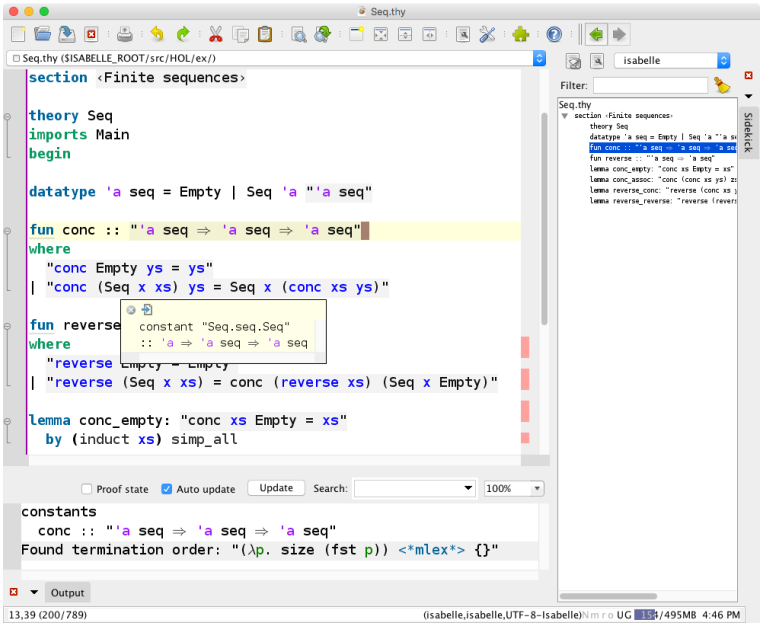

Figure 2.8: Programs in Isabelle's verification environment.

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

$$\sum_{i=0}^{n} i^3 = \frac{(n \cdot (n+1))^2}{4}$$

according to standards established by LaTeX. Isabelle/ISAC aims at widespread

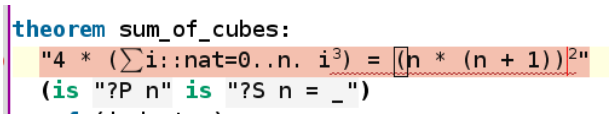

Figure 2.9: Isabelle's line-oriented formula presentation.

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.

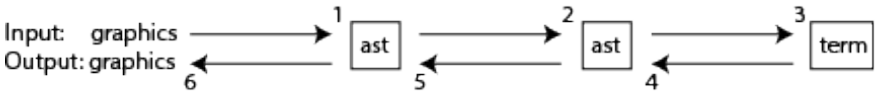

Figure 2.10: Translation from the graphical representation to term vice versa.

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 §2.2.2. The process in detail is as follows:

1. transformation: Pixels are handled by GUI-components and transformed into an AST, which is as close as possible to the graphical representation.
2. transformation: The AST close to graphical representation is transformed such, that its structure is as close to terms as possible.
3. transformation: The AST close to term representation is transformed into a term, Isabelle's data structure for deduction and computation in the prover.
4. transformation: 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. transformation: The AST close to term representation is transformed back such, that its structure is as close to graphical representation as

close as possible.

6. transformation: 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 $\#$ :: $'a \Rightarrow' alist \Rightarrow 'a\ list$ 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.

```
datatype (set: 'a) list =
    Nil  ("[]")
    | Cons (hd: 'a) (tl: "'a list")  (infixr "#" 65)
```

Figure 2.11: Definition of `list` plus notation.

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. [6].

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>)))
```

[6] 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 datastructure, 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>)))
```

Actually, the term generated from this AST has a very similar structure:

```
Const ("HOL.eq", "int list => int list => bool") $
  (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")
```

Note, that 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 [1, 5]. The types increase reliability of Isabelle's mechanical reasoning and go far beyond the kind of types known from programming languages.

After 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")
        ("\<^const>List.list.Cons" ("_constrain" ("_free" ccc) "\<^type>Int.int")
          "\<^const>List.list.Nil")))
    ("_constrain" ("_free" ddd) ("\<^type>List.list" "\<^type>Int.int"))))
```

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

```
post:
("\<^const>HOL.Trueprop"
  ("\<^const>HOL.eq"
    ("_list"
      ("_args" ("_constrain" ("_free" aaa) "\<^type>Int.int")
```

```
      ("_args" ("_constrain" ("_free" bbb) "\<^type>Int.int")
        ("_constrain" ("_free" ccc) "\<^type>Int.int"))))
  ("_constrain" ("_free" ddd) ("\<^type>List.list" "\<^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 *ISAC*-Prototype

The academic field of TP (combining ATP and ITP) attracts much attention from the side of software verification and from formal specification of technical systems; thus efforts in TP are focused on these requests. Attention from the side of education is still weak. *ISAC* is one of three projects on educational software for engineering mathematics based on TP technology: *MathToys* [**url-mathtoys**] classifies itself a "toy" while *E-Math* [**url-emath**] already started as an economic enterprise. So even judgment of seriousness of such undertaking is not yet settled.

2.4.1 Prototype for a New SW-Generation

The *ISAC* 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. The following features are considered essential:

Deduction covers math's distinguishing feature The distinguishing feature of mathematics is formal reasoning, this feature distinguishes it from other scientific disciplines and makes math a core method for many of them. Thus didactic experts stress importance of reasoning on all levels of math education. The technology of deduction appears as the most appropriate basis for educational math software. This is, however, presently not acknowledged in practice: All educational math software except the three mentioned above are built upon Computer Algebra (including Gröbner Bases for ATP in Dynamic Geometry).

An implicit consequence of the fact, that deductive technologies address most essential aspects of mathematics is, that such technologies promise a wider coverage in software support for engineering mathematics, in particular for specifying problems, identifying and arranging subproblems.

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

Mathematics knowledge is *self-contained* Fig.2.7 on p.12 showed, what wealth of knowledge TP can reveal by mouse click behind the simple `lemma "12345 * 67 + 12345 * 33 = 12345 * 100"`:

1. + does *not* refer to some program code computing sums, rather they are logical entities related to axioms of abstract algebra.
2. `int` does *not* refer to types of number representation, rather they refer to the established deduction of integers from naturals ($(x, y) \equiv (u, v) \longleftrightarrow x + v = u + y$). And there is the proof, that integers are associative, etc.
3. Each step, even a simple addition, is justified by some law (in this case `distrib_left`) in traditional notation of mathematics.

So mathematics knowledge is *self-contained*: each object or fact is mechanically justified, deduced from the axioms of logic in the end — so TP is apt to establish "complete model of mathematics".

Presentation can be close to traditional notation Another appealing feature is, that TP can *present* mathematics in traditional notation every educated person is familiar with (while the ability to *construct* knowledge remains demanding) — so no need to learn some programming, some new language or the like.

Of course, there are sophisticated deductive technologies like automated provers, SMT, Gröbner Bases, etc, which involve specific representation of formulas — but these representations can be hidden by a proof assistant (Isabelle, for instance, nicely hides the internals of Sledgehammer [20] and employs proof reconstruction [19]). So TP is apt to establish "transparent models" of mathematics.

ATP is most powerful in checking user input Given a logical context in a typed system, user input creates a proof situation: can the input formula be derived from the context using certain knowledge?

ATP provides the most powerful technologies to decide this question! Of course, the question is undecidable in principle (for instance in Isabelle/HOL), but experience in the ISAC-project [**url-isac-history**] shows, that engineering problems need to be broken down to sub-problems anyway and within these, most reasoning is much simpler than in proofs. Thus decisions for *derivable* or *not* can be given in most cases.

So, TP is apt to establish "interactive models" of mathematics.

An extension to TP: "Next-step guidance" One requirement appears indispensable for "systems that explain themselves": students can input steps in constructing a `Solution` on their own for learning by trial &

error — and get feed-back from the system. For checking correctness of input, ATP is the most powerful technology as mentioned above.

However, what if a student gets stuck, which is a frequent situation in learning by trial & error? Then the system has to provide a next step — and this requirement is beyond the scope of TP. So the ISAC-project uses Lucas-Interpretation [10, 11] for suggesting a next step and envisages a comprehensive dialogue component for adaptive user guidance.

Relevance of TP features for education is is given by the fact, that Isabelle/ISAC is a "complete" [8] , transparent and interactive model of mathematics. So ISAC is not primarily built for *teaching*, but for *learning* mathematics by interacting with the "model", by investigation and by trial and error — learning happens the same way as chess masters use chess playing software for developing new strategies.

2.4.2 ISAC's Front-end

The front-end is connected with the mathematics engine Isabelle/ISAC *not* via the standard interface [25], but by a specific one [**libisabelle**]. This special interface transports terms[9] 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 [10].

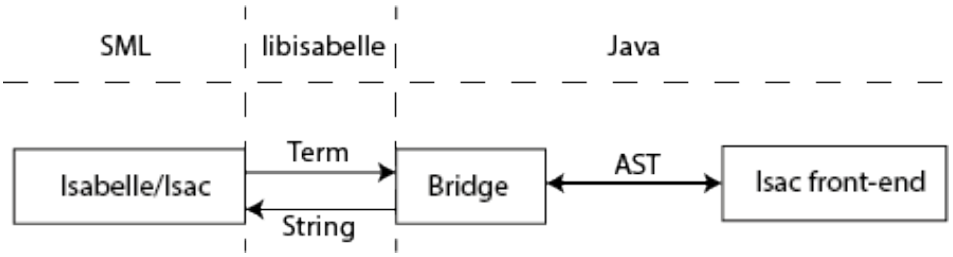

Figure 2.12: Transport of formulas from the front-end to the back-end.

Terms have first been exploited in ISAC by [6]. Most of ISAC still represents terms as strings. Both together is possible, because all formulas are transported by a Java object *Formula*, which contains both, the formula as string and the formula as term (see the connection between Isabelle/ISAC and *Bridge* in Fig.2.12).

[8] "Complete" not in the sense of the rigorous notion in model theory.

[9] We use the word "term" for the data structure used for mathematical reasoning and "formula" as a general notion without reference to data structure.

[10] Re-parsing of formulas encoded as strings shall remain as is according to §2.5.1

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

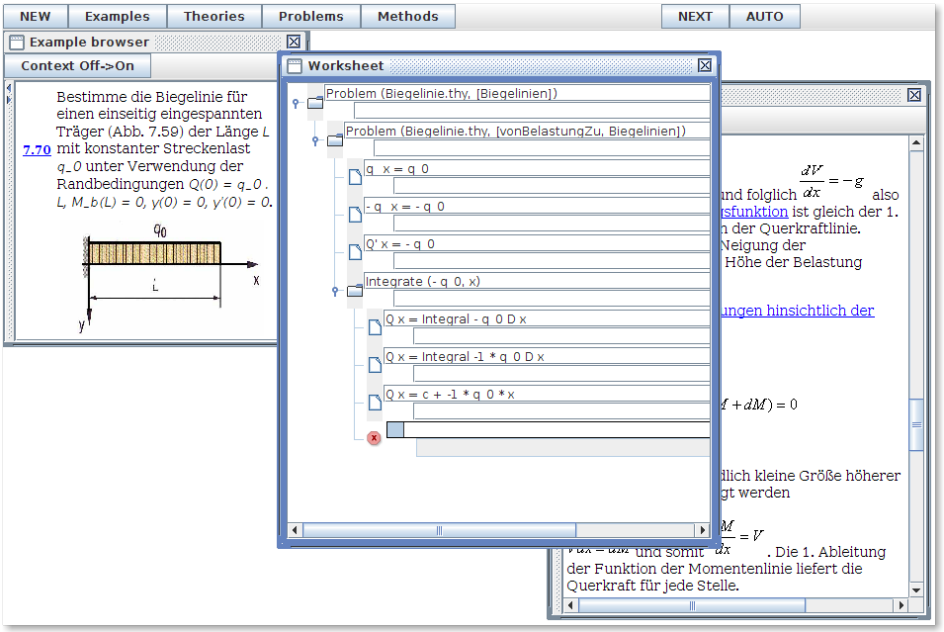

Figure 2.13: The previous *ISAC* front-end.

some buttons (*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 mathematics 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/*ISAC* in Fig.2.12 like all other components shown in Fig.2.13.

The *Theory browser* shows an explanation for this step [11].

The windows on the left and on the right are not concern of this thesis,

[11] 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.

only the window in the middle and in the focus, the *Worksheet*. The *Worksheet* 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, together called a *Calculation* in ISAC. Indentations of steps indicate subproblems (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/ISAC.

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/ISAC.
- 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 [6]. 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[12] tactics to be applied to the selected step.

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 ISAC'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 not can 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.

[12] ISAC's dialogue guide is prepared to present different lists, as soon as a respective user 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.

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 ISAC, those for a formula editor in §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 ISAC with Isabelle is also possible at the front-end: The Isar proof language [29] is designed on a high level of abstraction and implemented using standard concepts of compiler construction. So re-use (see §3.1 below) of Isabelle/-jEdit for Isabelle/ISAC shall be considered. Probably Isabelle will have an optional browser-based interface in a few years, which would be in line with ISAC's migration to mobiles. [13]

Restrict formula transformation to one direction, the direction from the back-end to the front-end. This divides efforts in less than half. The other direction, from the front-end to the backend, stays as is — where formulas are reverted to strings in a format, which can be parsed correctly by Isabelle/ISAC.

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

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 §2.3 cannot be demonstrated in the prototype.

[13] At present the major obstacle in re-using Isabelle/Isar for Isabelle/ISAC is that ISAC is a multi-user system, while Isabelle has no session management.

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 [6].

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.

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.

UR 2.3 ***Two-level feedback on input.*** Input is finally terminated by a special key, which submits the input formula to Isabelle/ISAC 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/ISAC).

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/ISAC'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-helds 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, ISAC'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 $\in, \sqsubseteq, \otimes, \otimes_\star, \otimes^\star$ etc.
5. *"Layout"* in the graphical representation is already different for binary operators (operators with two arguments): operator / produces $\frac{a}{b}$, operator $\widehat{}$ produces a^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 §**??** reflects associativity of operations by <row> and thus accommodates mathematical habits. ISAC 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 $\int_{[]}^{[]} [] d[]$ the graphical representations appears with fill-in-gaps, here denoted by []. See UC.**??**

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.**??**. 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 2 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 §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.

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 §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 §2.1).

UR 2.22 ***String-representation of formulas is variable.*** There are specific formats [6] 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/ISAC 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 JavaSwing.*** ISAC has the potential to gradually migrate in usage from universities to high school. At the latter acceptance of ISAC 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 ISAC will adhere to the existing front-end in JavaSwing 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 §**??**. For the editor a structure close to MathML presentation format §**??** 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/ISAC, 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/ISAC).

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 files at the place where the other files reside in the code.

SR 2.7 ***Use cases are reflected in the repository*** in the *JUnit TestCase* in package *isac.gui.mawen.TestUseCases*.

Chapter 3

Design Considerations for TP-based Editors

The TP-based Editor has special requirements witch are shown in §2.5. In this chapter discusses design consideration in order to perform these requirements.

3.1 Re-use of Existing Components

Isabelle is a *generic* proof assistant §2.3.1 with a *generic* graphical user-interface §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 ISAC-prototype. §2.3 shows the status quo of these technologies, where §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 ISAC-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 straight forward – we recall that Isabelle uses two structures of Formulas:

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

To guarantee the compatibility, the ISAC-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 in §2.3.3 from SML to something, which can be integrated into the ISAC-

prototype's Java Swing — and this is Scala already in use by PIDE [27] and introduced in §2.5.3.

Since the editor has to be integrated into the ISAC-prototype, re-use of respective components is obvious as well.

3.1.1 Adapt Isabelle's AST Transformations

As shown in §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 straight forward. In particular, the Isabelle developer team established a functional style of programming, which has been pertained during translation for the ISAC-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 ISAC'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/ISAC the ASTs are transformed into a string, which can be parsed back and typed by Isabelle. For typing ISAC maintains a logical context implemented by another student's project [8]. Of course, this is inefficient, but makes this thesis maximally independent from other changes in the ISAC system.

3.1.2 Integration into the ISAC Architecture

Before integration of the formula editor into ISAC can be considered, relevant components of the *Worksheet* need to be identified. ISAC's architecture

provides the *Worksheet* with very few, but highly complex connections. This is shown in Fig.3.1:

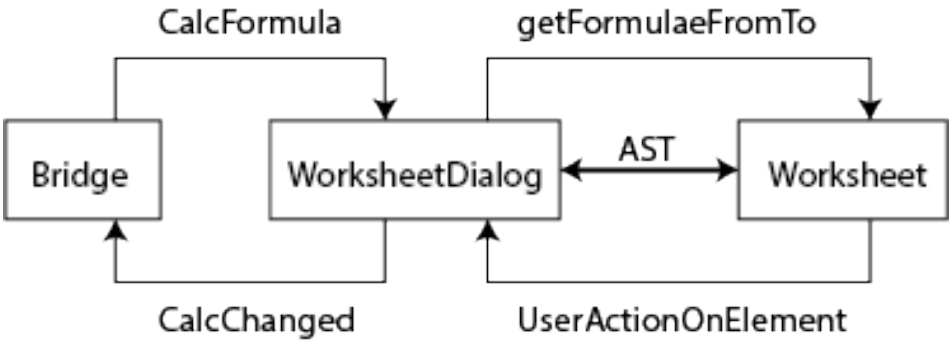

Figure 3.1: ISAC's architecture for user interaction on formulae.

This architecture differs from the MVC architecture [2] in that the dialog occupies a central role: the *WorksheetDialog* is part of a *DialogGuide* (not shown in the figure), which coordinates the *WorksheetDialog*s (each *Worksheet* has its *WorksheetDialog*) and the *BrowserDialog*s (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 ISAC is designed for adaption to various learning scenarios, see. §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/ISAC for checking correctness.

Isabelle/ISAC'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* allows the *Worksheet* to request *getFormulaeFromTo* (an input formula might have deleted subsequent formulas). This request goes directly to the *Bridge*, which forwards the request to Isabelle/ISAC 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 ISAC-Prototype

§1 has stated a comprehensive code base for the ISAC prototype, §2.4.2 gave a description of the front-end and the role of the *Worksheet* and §3.1.2 showed how architecture connects the *Worksheet* with the other components of ISAC. 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 ISAC 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 custom a *TreeCellRenderer* and a *TreeCellEditor*. The *TreeCellRenderer* is for visualization purposes, while the *TreeCellEditor* is for Editing — the usual separation of model and view [2].

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.

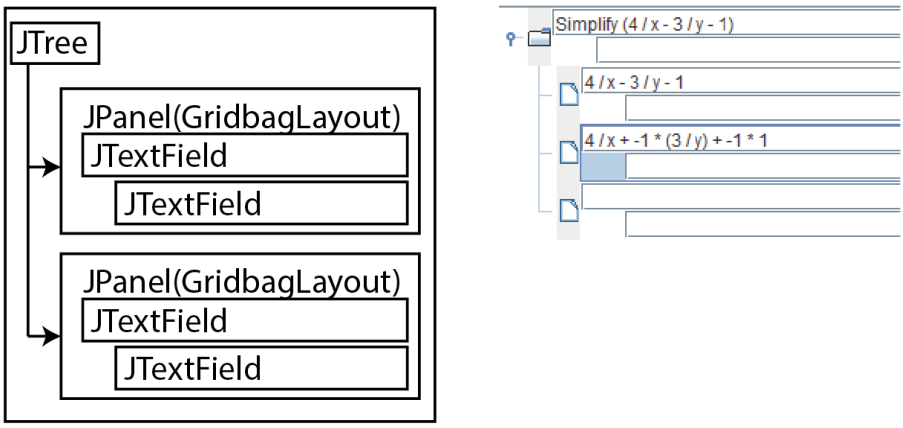

Figure 3.2: An EditorFactory abstracts to one interface for both, the old string-representation and the new editor.

The JTextFields were editable in the *CalcTreeCellEditor* and locked in

the *CalcTreeCellRenderer*. Also no Events were fired in the *CalcTreeCell-Renderer*. 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 calculations 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 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.

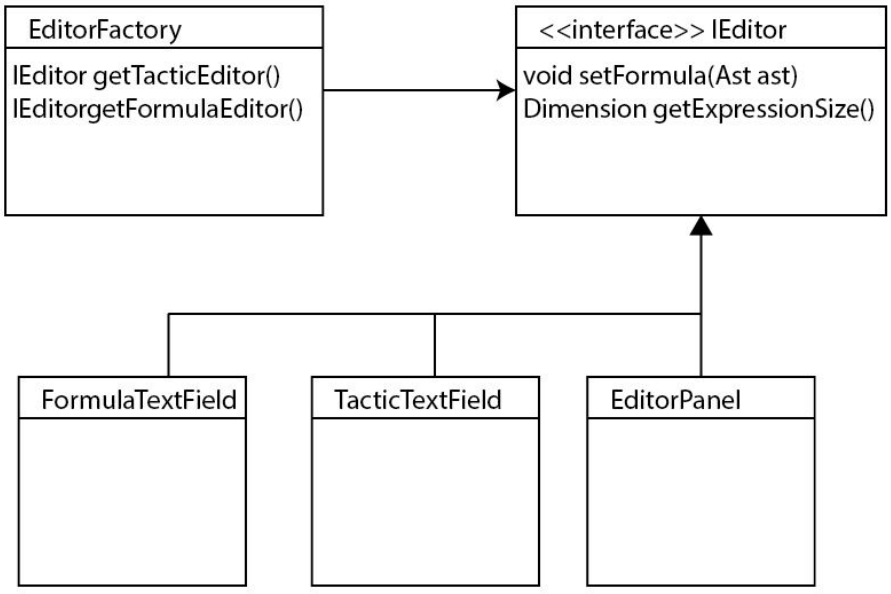

Figure 3.3: On the right side there is the Graphical Representation of the JTree and on the left side the structure of the UI Components.

An EditorFactory determines on the basis of the property file to return the old *JTextField* Representation or the new editor, which is encapsuled in the *EditorPanel* class. The *EditorPanel* is explained in §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 ISAC 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 ISAC's prototype status and address specific demonstrations. Beginning with UC.3.7 the individual cases are ranked by priority.

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

1. addition and subtraction $+ -$, multiplication $\cdot$, division $\frac{x}{y}$
2. differential operator $\frac{d}{d\,x}$
3. $\int_a^b f\,dx$
4. further operators according to UR.2.6

UC 3.2 ***variables ranked by urgency*** are

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 $[setzeRandbedingungen, Biegelinien]$ (compare UC.3.8)
2. vectors like $\begin{pmatrix} x \\ y \end{pmatrix}$ or $\begin{pmatrix} x & y \end{pmatrix}$
3. matrices like $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ or $\begin{pmatrix} 1 & b & \frac{1}{2} \\ d & \int_0^2 x\,dx & f \end{pmatrix}$

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

```
Simplify (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

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

demonstrates a specific benefit of §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 $a + \frac{b}{c}$ can proceed in several ways

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

   $$\otimes + \odot$$

   where $\odot$ denotes a gap and $\otimes$ denotes a gap with cursor.
   (b) After input of "a", in the second GAP a "/" is input, this creates the rendering

   $$a + \frac{\otimes}{\odot}$$

   (c) After input of "b" a special key moves the cursor (see UR.2.11) to the denominator:

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

2. Input of infix operators start with the first argument:
   (a) After input of "a" the "+" is input which renders as

   $$a + \otimes$$

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

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

   (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 subterms 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 ISAC has several ways to help, for instance with output as in lines 06 and 07:

```
04 ...
05    d/dx x + d/dx sin(x^2)
06                          Rewrite (d/d bdv sin(u) = cos(u) * d/d bdv u)
07    d/dx x + cos(x^2) * d/dx ⊗
08 ...
```

In line 07 ISAC presents a formula with a fill-in gap and the CURSOR placed into this gap, indicated by $\bigotimes$. Note that the first formula in this calculation is

```
Diff (x ^ 2 + sin (3 * x ^ 4), x)
:
```

i.e. in graphical representation

$$Diff(x^2 + sin(3 \cdot x^4),\ x)$$

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 = Integral c_3 + 1 / (-1 * EI) * (c_2 * x + c / 2 * x ^ 2 + -1 * q_0 / 6 * x ^ 3) D x
:
Problem (Biegelinie, [setzeRandbedingungen, Biegelinien])
:
[L * q_0 = c, 0 = (2 * c_2 + 2 * L * c + -1 * L ^ 2 * q_0) / 2, 0 = c_4,  0 = c_3]
:
```

The first and last formula in respective graphical representation are:

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

$$[L \cdot q_0 = c,\ 0 = \frac{2 \cdot c_2 + 2 \cdot L \cdot c + -1 \cdot L^2 \cdot q_0}{2},\ 0 = c_4,\ 0 = c_3]$$

Graphical representation makes clear, that there are questionable results of simplification in ISAC, 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 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

$$\mathit{if}\ n<0\ \mathit{then}\ \mathit{fac}\ n=n*\mathit{fac}(n-1)\ \mathit{else}\ 1$$

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

```
Program Simplify t_t =
  Repeat (
    Try (Rewrite_Set klammern_aufloesen False) @@
    Try (Rewrite_Set ordne_alphabetisch False) @@
    Try (Rewrite_Set fasse_zusammen False) @@
    Try (Rewrite_Set verschoenere False)) t_t
```

3.4 Datatypes

§2.3.3 demonstrated Isabelle's transformation on annotated syntax trees (ASTs). §2.4.2 mentioned that ISAC's interface to Isabelle/ISAC 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 ISAC 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 §2.4.2, is this:

```
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 §3.4.2 above is implemented as Scala `Term` as follows:

```
App(App(Const("List.list.Cons",
    Type("fun",List(Type("Int.int",List()),Type("fun",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
    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",Li
      Free("ccc",Type("Int.int",List()))),
      Const("List.list.Nil",Type("List.list",List(Type("Int.int",List()))))))))
```

Another `Term` $aaa + \frac{4}{bbb::real}$ is implemented like this:

```
App(App(Const("Groups.plus_class.plus",
    Type("fun",List(Type("Real.real",List()),Type("fun",List(Type("Real.real",List()),Type("Real.real"
  Free("aaa",Type("Real.real",List()))),
  App(App(Const("Fields.inverse_class.divide",
      Type("fun",List(Type("Real.real",List()), Type("fun",List(Type("Real.real",List()), Type("Real.r
    App(Const("Num.numeral_class.numeral",Type("fun",List(Type("Num.num",List()),Type("Real.real",List
      App(Const("Num.num.Bit0",Type("fun",List(Type("Num.num",List()),Type("Num.num",List()))),
        App(Const("Num.num.Bit0",Type("fun",List(Type("Num.num",List()),Type("Num.num",List()))),
          Const("Num.num.One",Type("Num.num",List()))))  ),
    Free("bbb",Type("Real.real",List())))
```

This term shows, that Isabelle internally relies on binary encoding of numbers: `Num.num.Bit0` and `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 [21] 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 §2.3.3, SR.2.4. We translate the SML data structure into the following Scala datastructure one-to-one:

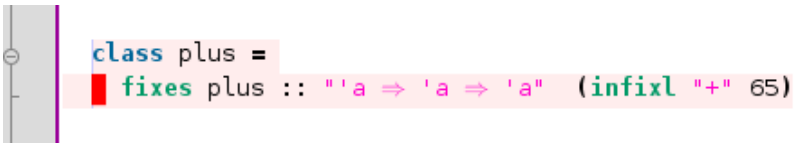

Figure 3.4: Definition of + in Isabelle.

```
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.37 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.

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

```
Appl[Constant("_constrain"), Appl[Constant("_free"), Variable("ccc")],
  Appl[Constant("_ofsort", Appl[Constant("_tfree"), Variable("'a")],
    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. §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.38; 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.38 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 §2.5.1) is to represent numerals not in binary format as shown on p.37 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 §3.4.2, specific *Constant*s are inserted and deleted in the structure (without changing the mathematical semantics) as follows:

marked subterms
(a) *ast* $\longleftrightarrow$ *Appl(List(Constant("BOX"), Constant("1"), ast))*
(b) *ast* $\longleftrightarrow$ *Appl(List(Constant("BOX.nnn"), ast))*
fill-in gaps
ast $\longleftrightarrow$ *Constant("GAP")*
cursor set by system
ast $\longleftrightarrow$ *Appl(List(Constant("CURSOR"), 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.2.12 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/ISAC needs to be called.

The cursor can be set by the system according to UR.2.13. ISAC's user guide (which presently is a stub prepared for setting the cursor, see for instance UC.3.7,

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

```
Appl(List(
  Constant("CURSOR"),
  Constant("GAP")))
```

3.5 Layout Classes for Sub-Terms

This thesis works on a limited number of use cases §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 §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 $a+(b)+c) \neq a+b+c$ are different terms according to UR.2.9. This UR excludes MathML's `row` introduced in §2.2.2. 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:

$$a + \cfrac{\cfrac{\frac{b+c}{d+e}}{f+g}}{h+i} \over j+k$$

However, LaTeX reduces size only down to three levels of subterms, the topmost $\frac{b+c}{d+e}$ 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 vertically:

$$x^{x^{x^{x}}}$$

Also in this class LaTeX reduces size only down to three levels: the topmost three x all have the same size.

Prefix operators like sin, cos, $solve(x+1=2,\ x)$ have different kinds of arguments (terms, pairs as with *solve*, 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 $\{n];.];n<100\}$ is omitted in this thesis.

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

$$\begin{pmatrix} a & b & c \end{pmatrix} \qquad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \qquad \begin{pmatrix} a \\ b \\ c \end{pmatrix} \qquad \begin{pmatrix} \frac{b}{\frac{c}{d}} & b & x \\ e & & \\ c & \int_a^b x^2 dx & y \end{pmatrix}$$

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:

$$\int x^2 dx \qquad \oint_C F\, ds \qquad \int_a^b x^2 dx$$

The first example above is in the use cases §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 §4.1 together with general decisions for coding. §4.2 discusses relevant details of Java Swing, before dynamic aspects can be addressed in §4.3. The final section §4.4 describes the procedure of implementation, which had to cope with the comprehensive code base of the ISAC-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 ISAC's front-end written in Java Swing. Before an assignment of these two programming languages can be approached in §4.1.3, a structure for the editor's code is introduced in §4.1.1 and respective integration into ISAC discussed in detail in §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

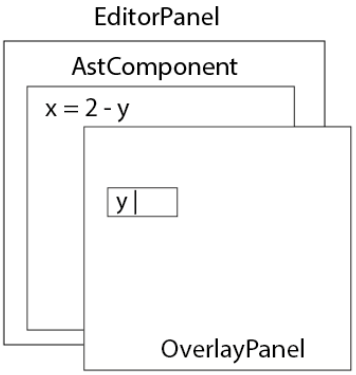

Figure 4.1: OverlayLayout comprises AstComponent and OverlayPanel.

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* [2]. 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 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.

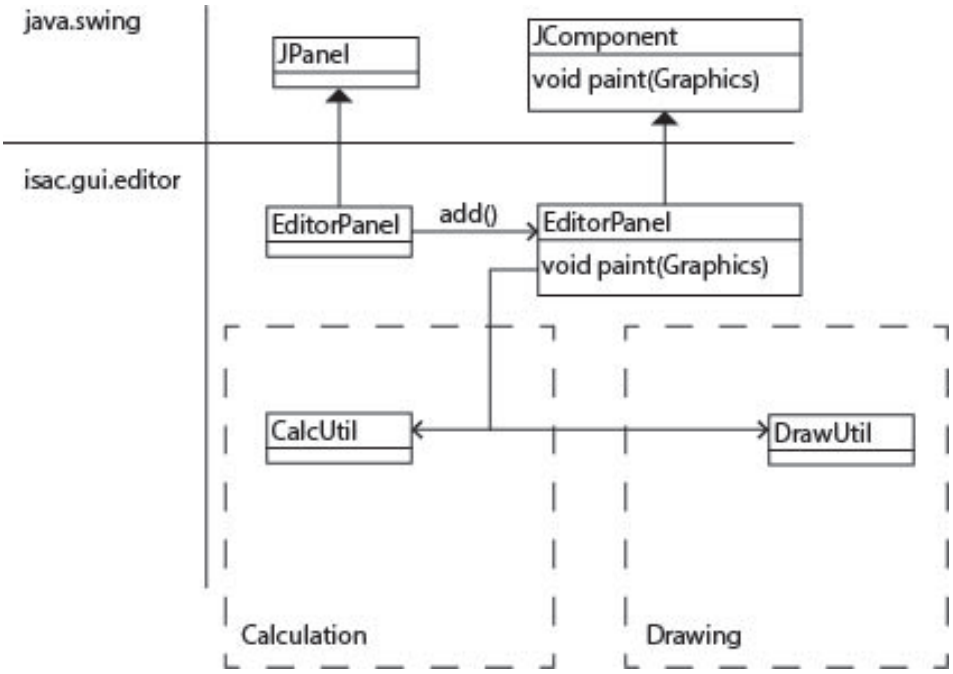

Figure 4.2: Strucure of the Formula Editor.

4.1.2 Integration into ISAC

As shown in §2.4.2, the main tool for interaction on formulas is called a *Worksheet* in ISAC. 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 §3.2.1. The sisac project leader decided, *not* to consider formulas in this field.

[2] This example is the most beloved one in ISAC's test suite, because it comprises much of ISAC'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 `solve`s a large class of equations; it first generates a specific normal form `-1 + x = 0`, which allows to determine the degree of the 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).

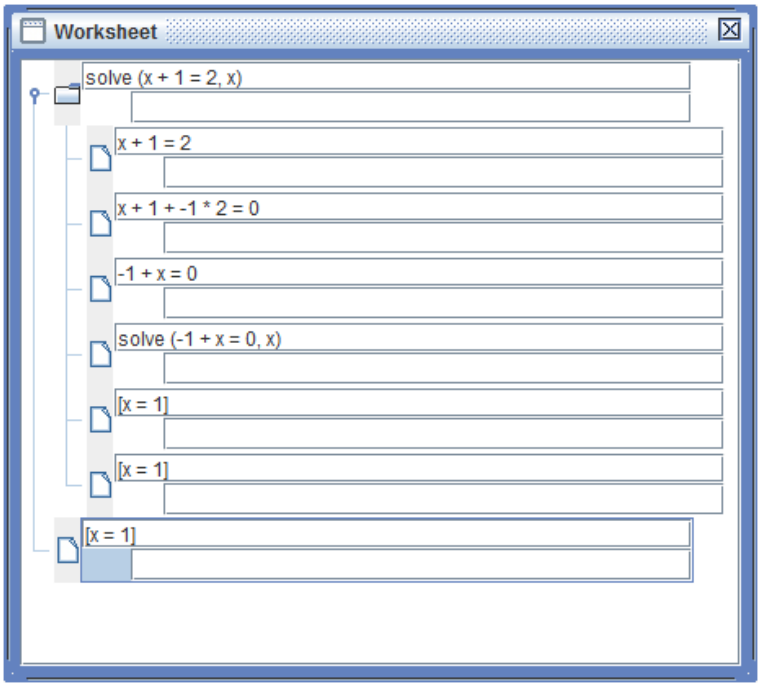

Figure 4.3: A Calculation is constructed in a *Worksheet*, i.e. a *JTree*.

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 §3.2.2, which decides with respect to a property file:

```
1 public class EditorFactory {
2 private static boolean IsEditorVisible;
3   static {
4     try {
5       InputStream is = WindowApplicationPaths.class
6         .getResourceAsStream("/properties/Editor.properties");
7       if (is != null) {
8       Properties p = new Properties();
9       p.load(is);
10      IsEditorVisible = p.getProperty("EDITOR")
11        .replace(" ", "").equalsIgnoreCase("MAWEN");
12    }
13  } catch (Exception e) {
14      IsEditorVisible = false;
15    }
16  }
17  public static IEditor getFormulaEditor() {
18    if (IsEditorVisible) {
```

```
19        return new EditorPanel();
20      }
21      return new FormulaTextField();
22    }
23 }
```

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

```
#=JTextField MAWEN
EDITOR=JTextField
# true false
IS_DRAG_ALLOWED=true
IS_ZOOM_ALLOWED=true
# float values
START_ZOOM_FACTOR=1
START_X=15
START_Y=25
```

Figure 4.4: Example for the factory's property file.

shows, that there are further data, now required by the new formula editor.

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

```
1 editor_ = EditorFactory.getFormulaEditor();
2 tacticText_ = new JTextField();
3 editorPanel_ = new JPanel();
4 editorPanel_.setOpaque(true);
5 editorPanel_.setBackground(UIManager.getColor("Tree.background"));
6 editorPanel_.setLayout(new BorderLayout());
7 editorPanel_.add(editor_.getComponent(), BorderLayout.CENTER);
8 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* und
2. *notifyIsaCheck*.

where the latter continues the old behaviour and the former adds a new behaviour: *notifyIsaCheck* sends the formula to Isabelle (while the AST is transformed to a string which can be parsed by Isabelle, see §2.5.1 and Fig.2.12).

New *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 backend (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 combination of Java and Scala. In Java and Java Swing ISAC'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 programming language, which better supports the tasks to be programmed respectively. The assignment turned out straight forward as follows.

- **Scala** was used for code handling Scala ASTs, elegantly exploiting the powerful `match`. Scala code seamlessly calls Java Swing. Examples are *CalcUtil* and *DrawUtil*.
- **Java** was used for handling the Java Swing components to embedded into the *Worksheet* together with respective event handling. An example is *EditorPanel* extending *JPanel*.

Referring to Fig.4.2 on p.45 we have: An AST is passed through the Java *EditorPanel* to the Scala *AstComponent*. The AST is manipulated in *CalcUtil* and *DrawUtil* by Scala's `match`. After editing is finished the AST is returned by the *EditorPanel* and the *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 ISAC. Some classes are already implemented in libisabelle (see §2.4.2); in order to *not* overwrite these, respective identifiers get an "X" as prefix, for instance *XSyntax*.

Two different coding standards are the consequence of preparing for further narrowing between Isabelle and ISAC: All Scala code translated from

Isabelle's SML adheres to Isabelle's coding standards [28]. Other Scala code and Java code adhere to ISAC's coding standards [24]. Resulting conflicts in certain program files are taken into account.

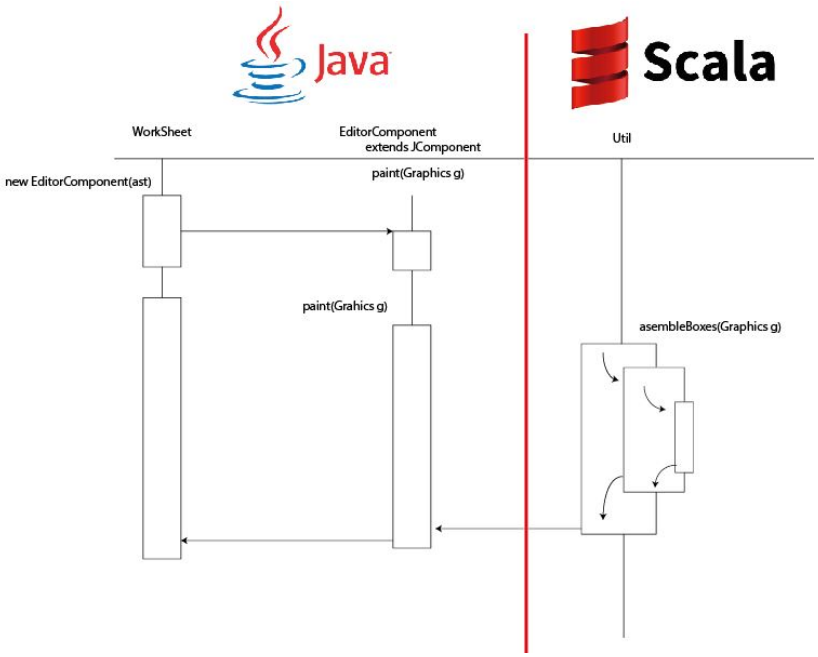

Figure 4.5: Calculation is done by Scala, UI-Component are created in Java.

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 §2.2.3 is specified for Java below, first in general, then for single symbols in §4.2.1 for expressions with horizontal expansion in §4.2.2 and for expressions with vertical expansion in §4.2.3.

Rendering of formulas is done by overwriting the *paint* method of *JComponent* [2]. 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

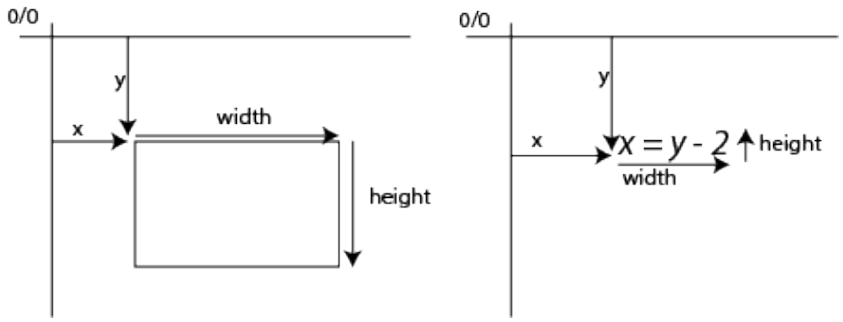

Figure 4.6: Two different coordinate systems in Swing.

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 boxed 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 box". The second layer, called "inner box", is calculated from the maximal with 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.4.8 shows inner and outer boxed for four symbols. There

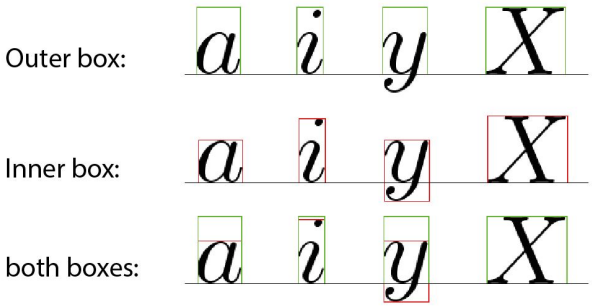

Figure 4.7: Layers for drawing symbols.

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 §4.2 and boxes 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 §2.2.4 for reading text. So Swing relives the programmer from all respective work.

The alignment of operators goes along the midline as mentioned in §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. Swing supports interpolation by simply configure rendering this way:

```
1    new RenderingHints(
2       RenderingHints.KEY_TEXT_ANTIALIASING,
3       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:

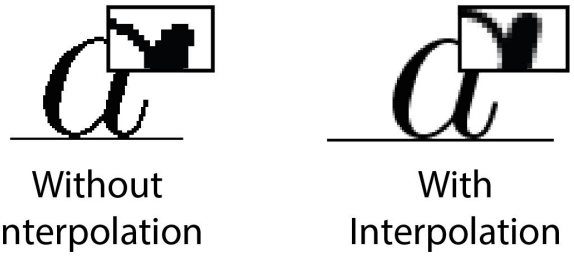

Figure 4.8: Compare a symbol with and without interpolation.

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 used for the operators $+, \cdot$ and $-$. Given the example $a + b$, the respective AST is:

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

class identifier	*LaTeX*	*implemented*
`horizontal`	$+ \cdot - =$	yes
`fraction`	$\frac{a}{b}$	yes
`exponent`	a^b	yes
`prefix`	$\sin \cos \tan (-a)$	yes
`list`	$[a, b, c]\ \{n \,.\, n > 1\}\ (a, b)$	partially
`matrix`	$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$	no
`spec-op`	$\int x^2 dx\ \oint_C F\, ds\ \int_a^b x^2 dx$	no

Table 4.1: Layout classes for operators.

Rendering this formula works as follows: First the box for a is calculated. The positition 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:

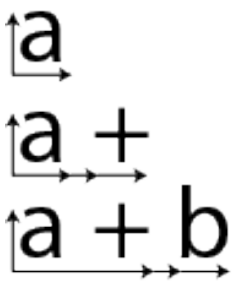

Figure 4.9: How to calculate the box for $a + b$.

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 §2.2.3 and §2.2.6. The design identified several layout classes in §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 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 §4.1.1 interact with each other. In particular, §4.3.1 explains calculation and rendering of a formula, §?? explains the effects of mouse movements over a formula and §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*.

```
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(")"), 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     g.setFont(new Font("CMMI12", Font.PLAIN, CalcUtil.fontsizeOf(2)))
26     val RectLv2 = getStringBounds(g, str, 0f,0f)
27     ast_Stringbounds = ast_Stringbounds + (str -> (RectLv0, RectLv1,
       RectLv2))
28   }
```

```
29   case Appl(asts) => asts.foreach(x => fillStringbounds(x, g))
30 }
```

In a second step *CalcUtil.assembleBoxes* assembles boxes according to *Settings.ast_Stringbounds*. After each successful calculation of a box in *CalcUtil.assembleBoxes.p* a call-back is performed (the purpose of the call-back is explained later). Boxes for *Constant* and *Variable* are calculated immediately. *Appl* describing composed formulas are implemented with one or two arguments, which is conformant to most of the layout classes §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.

```
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)
12          case None => {}
13        }
14      }
15    }
16  }
```

```
17   return None
18 }
```

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 §3.4.3:

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

And Fig.4.10 on p.55 shows the respectively rendered formula.

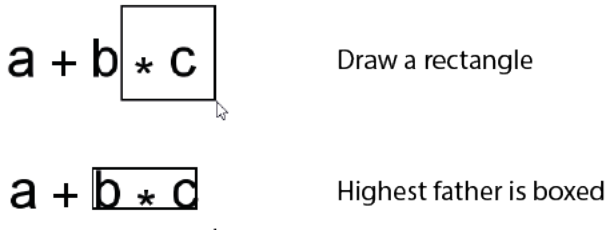

Figure 4.10: The above selection results in the rendering below.

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 §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.

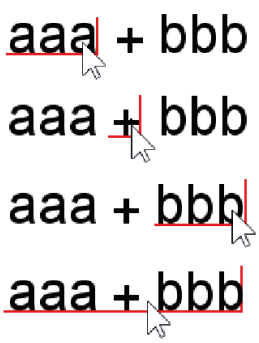

Figure 4.11: Four selections according to mouse postion.

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 on p.4.3.2) as follows:

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

is updated to

```
Appl(Constant("+"), Appl(Constant("CURSOR"), Variable("a")), 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*:

```
1 override def paint(g: Graphics) = {
2   ...
3   box =  CalcUtil.assembleBoxes(ast, (box: DrawBox) => {
4     if (EventUtil.foreachBoxFunction != null) {
5       EventUtil.foreachBoxFunction(g2, box)
6     }
7   })
8   ...
9 }
```

CalcUtil.assembleBoxes gets a call-back as described in §4.3.1), which works on each box as described by the following code:

```
1 var foreachBoxFunction : (Graphics, DrawBox) => Unit = null
2
3 def drawBoxAt(p: Point) = {
4   foreachBoxFunction = EventUtil.doInBox(p, (g, box) => {
5       Box.draw(box, g, false)
6     }
7   )
8 }
```

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 §4.3.2 on p.54 and on p.55. Two variants of editing are presented analogously to §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 straight forward 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 "$\cdot(b+$" 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 "$\cdot(b+$" 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 changeset https://intra.ist.tugraz.at/hg/isac/rev/a2a220b0996b and then removed in favour of the variant below.

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 §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 §4.3.2. [3] After editing has started, key events are handled according to the state transition diagram in Fig.4.12. The diagram is read as follows:

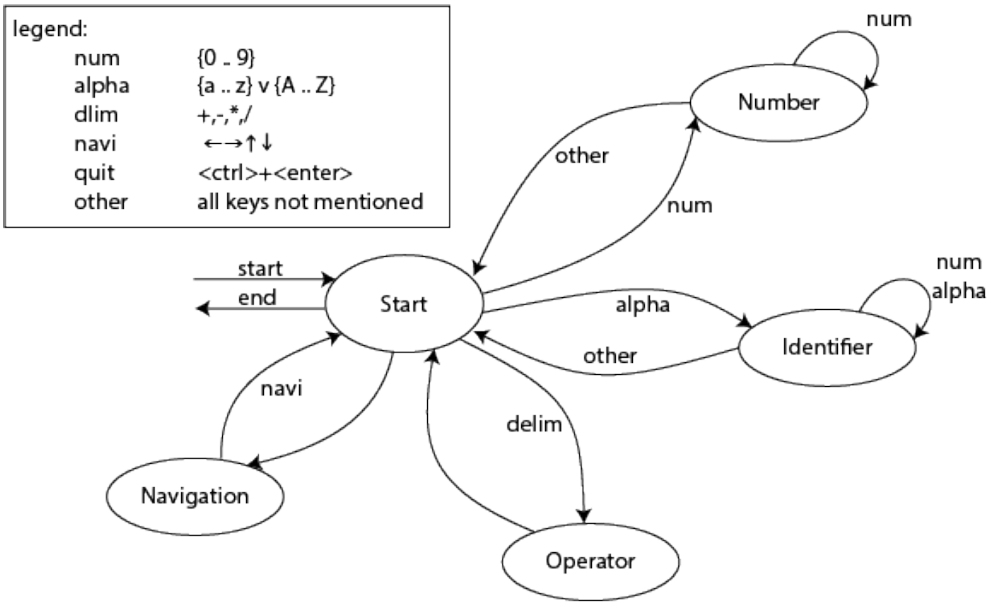

Figure 4.12: Aufbau des State transition diagram.

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

```
Appl(Constant("CURSOR"), Constant("GAP"))
```

or continues input on this kind of AST:

```
Appl(Variable(str), 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.

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 *GAP*s 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.

[3] Replacing a whole sub-term following "selection of boxes" is not yet implemented.

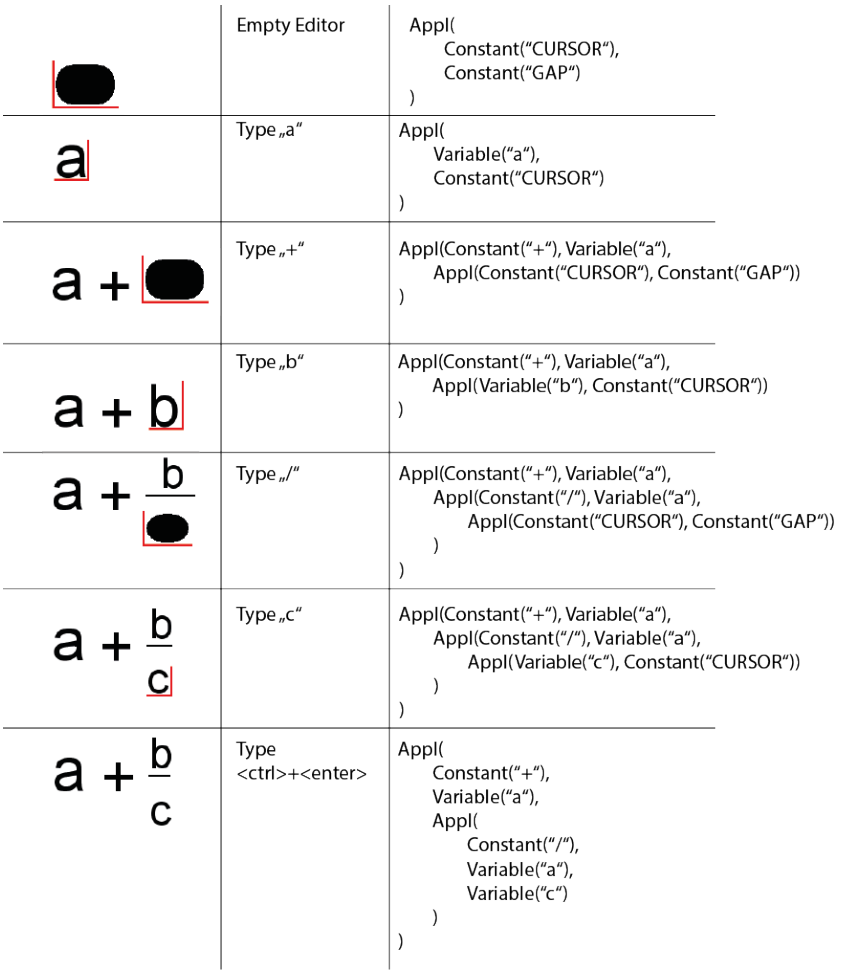

Figure 4.13: create the Formulas by typing "a+b/2".

And Fig.4.14 on p.60 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

```
Appl(Cursor, ast)
```

and

```
Appl(ast, Cursor)
```

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

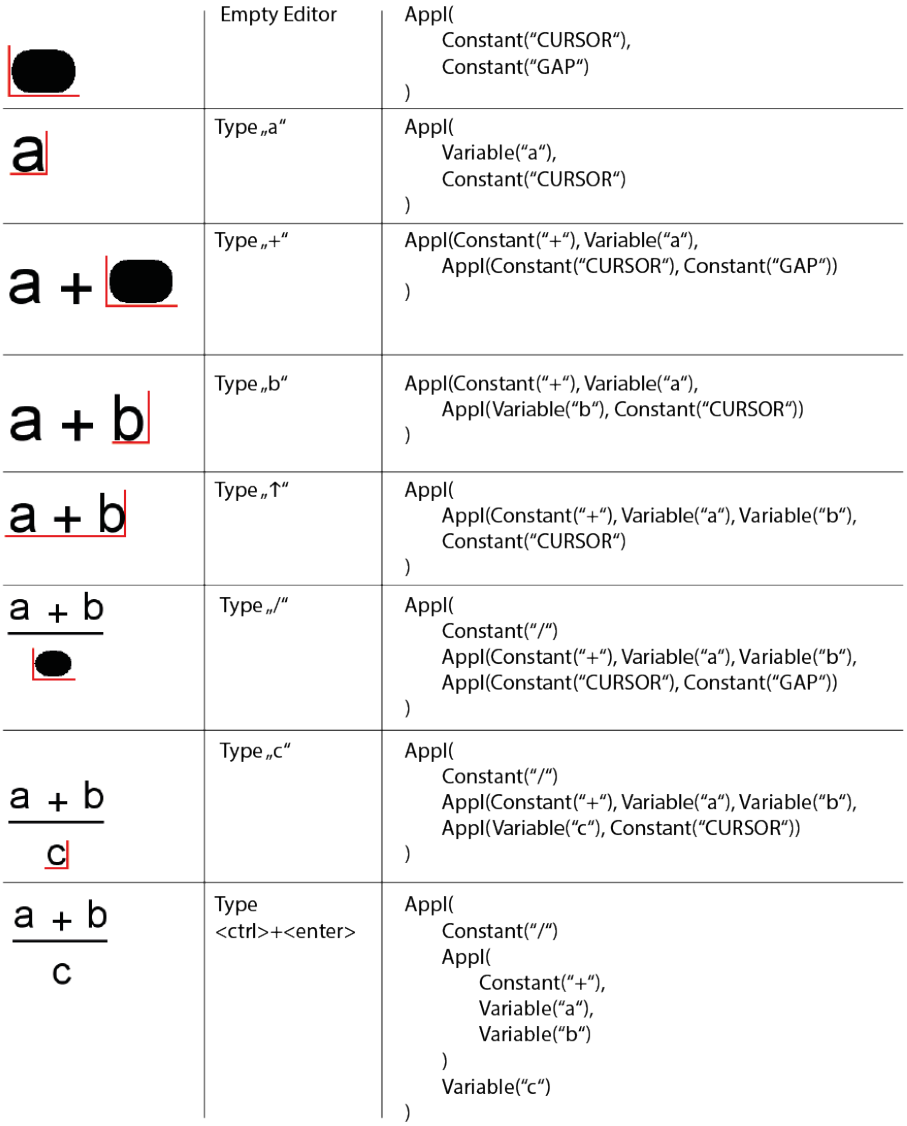

Figure 4.14: Create the Formulas by typing $\frac{a+b}{c}$.

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 `GAP`s 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 §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 for a Single Formula

The ISAC prototype still needs to be started from within the IDE, with separately launching four modules §2.4.1 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 TestCase. 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 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 [6] 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 *prototyping* a formula editor in ISAC, a *prototype* for a new kind of educational software, which builds upon a 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 TP-based formula editor. The study involved much preliminary work on accessibility and inclusion §2.1, on standards in formula presentation (LaTeX and MathML) §2.2, on the state of the art of front-ends in TP §2.3, on the predetermined architecture and code base for embedding the editor §2.4 as well as predetermined user-requirements and software-requirements §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 ISAC, datastructure for rendendering 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 ISAC was challenging but brings the benefits of existing data structures for formulas, respective transformations and an elaborated user interface for calculations.

1. **. . . accomplished**
   (a) design and implementation of an interface for an accessible and inclusive editor
   (b) integration into ISAC 2.4 (UR.2.5) by adopting the *CalcTreeCell-Renderer* and *CalcTreeCellEditor*
2. **. . . partially accomplished**
   (a) two levels of feedback are implemented (UR.2.3): feedbakc by Isabelle is left as is and *LocalCheck* is only a stub
   (b) translation from ASTs (after input) to terms for check by Isabelle/ISAC 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. **. . . 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

Datastructure for rendendering in Isabelle are annotated syntax trees (AST), however the present interface between ISAC and Isabelle only transports terms.

1. **. . . 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)
   (e) parentheses according to operator priority (UR.2.16)

2. **. . . partially accomplished**
   (a) CURSOR has no position
3. **. . . 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-cklick
      i. keep types during translation from Term to AST
      ii. additional mouse features in the editor

Rendering in LaTeX quality establishes the first impression when approaching a formula editor in a software system for mathematics.

1. **. . . 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 respevctive levels
   (d) automated selection of fonts depending on operators or identifiers of variables
2. **. . . partially accomplished**
   (a) rendering of operators 2.8 UR.2.9
      i. integral works only with two arguments
      ii. unknown operators are displayed prefix with the arguments in a horizontal sequence
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.

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 (2.8 UR.2.9)

i. addition, multiplication, subtrakcion

ii. division, exponentiation

(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. **. . . partially accomplished**

(a) input a formula from scratch (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 ware 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 level 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.2.202.21, (UR.2.22)

3. **. . . not yet tackled**

(a) input a formula from scratch

i. unary minus

purpose	*program language*	*lines of code (LoC)*
production code	Java	246
	Scala	2.118
test code	Java & Scala	1.114
total		3.478

Table 5.1: Lines of code added by the prototype editor

(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 – in principle 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 ISAC'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 ISAC plus the code added by this thesis as shown in Tab.5.1 on p.67. (This table does not show the updates required for introducing the interface for the editor into ISAC and adaptions to the test environment §4.4). 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.

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-stuctures like those of formulas. Thus integration into ISAC was straight forward.

This insight paves the way for a mid-term alternative to an ISAC front-end on tablets and handhelds [1] — an editor implemented in Scala & 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 straight forward, but respective layout classes have been identified.

This insight allows to envisage development of ISAC 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 menues. Respective development can follow 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 ISAC 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. The following enumeration collects all points which have been mentioned as not (completely) accomplished in §5.

1. Integration into ISAC
   (a) consider to do simple checks of a formula locally (without calling Isabelle): matching parentheses, etc

[1] (ISAC 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.

   (b) extend AST-AST-translations to the full range of mathematical operators and data structures (matrices, etc)
   (c) implement the interface for the Braille display
   (d) provide user-configuration of mathematics standards for the Braille
   (e) connect Isabelle's user-friendly syntax definitions with the editor for the purpose of extending the language of mathematics (UR,2.8)
   (f) automated generation of rewrite rules for AST-AST-translations from syntax definitions
2. Datastructure for rendendering
   (a) provide the other direction of translation: from AST back to Term and submit the latter to Isabelle (instead of strings to be reparsed)
   (b) record positions for elements of formulas for the purpose of displaying terms on request
   (c) make types visible in the editor by mouse-cklick
      i. keep types during translation from Term to AST
      ii. additional mouse features in the editor
3. Rendering in LaTeX quality
   (a) rendering of operators 2.8 UR.2.9
   (b) unary minus
   (c) keys for navigation cannot be changed (UR.2.17)
   (d) no auditive feedback (UR.2.19)
   (e) selected sub-terms cannot be deleted or replaced
4. User Interactions and Editing

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

- ISAC'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 with exceeds 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

[1] Luis Damas and Robin Milner. "Principal type-schemes for functional programs". In: *9th Symposium on Principles of programming languages (POPL'82)*. Ed. by ACM. 1982, pp. 207–212 (cit. on p. 16).

[2] Amy Fowler. "A Swing architecture overview" (1998) (cit. on pp. 30, 31, 44, 49).

[3] Gerhard Gentzen. "Untersuchungen über das logische Schließen. II". *Mathematische Zeitschrift* 39.1 (1935), pp. 405–431. URL: http://dx.doi.org/10.1007/BF01201363 (cit. on p. 12).

[4] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994 (cit. on p. 9).

[5] R. Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60 (cit. on p. 16).

[6] Natalie Karl. "Developing an Inclusive Approach for Representing Mathematical Formulas". http://www.ist.tugraz.at/projects/isac/publ/Masterthesis_NatalieKarl.pdf. MA thesis. Linz, Austria: Hagenberg University of Applied Sciences, 2016 (cit. on pp. 2, 5–7, 19, 21, 23, 25, 62).

[7] Donald E. Knuth. *Typesetting Concrete Mathematics*. 1989 (cit. on p. 9).

[8] Mathias Lehnfeld. "Verbindung von 'Computation' und 'Deduction' im ISAC-System". Bakkalaureate project. MA thesis. Institut für Computersprachen, Technische Universität Wien, 2011 (cit. on p. 29).

[9] Daniel Marqu'es. *WIRIS OM tools. a semantic formula editor*. 1st ed. Birkbeck Institutional Research, 2006 (cit. on p. 6).

[10] Walther Neuper. "Automated Generation of User Guidance by Combining Computation and Deduction". In: http://eptcs.web.cse.unsw.edu.au/paper.cgi?THedu11.5. 2012, pp. 82–101 (cit. on p. 19).

[11] Walther Neuper. "Lucas-Interpretation from Users' Perspective". In: *Joint Proceedings of the FM4M, MathUI, and ThEdu Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Computer Mathematics*. http://cicm-conference.org/2016/ceur-ws/CICM2016-WIP.pdf. Bialystok, Poland, July 2016, pp. 83–89 (cit. on p. 19).

[12] Walther Neuper. "On the Emergence of TP-based Educational Math Assistants". In: vol. 7. 2. Special Issue "TP-based Systems and Education". Feb. 2013, pp. 110–129. URL: https://php.radford.edu/~ejmt/ContentIndex.php#v7n2 (cit. on p. 17).

[13] Walther Neuper. "Prototyping "Systems that Explain Themselves" for Education". In: http://www.ist.tugraz.at/projects/isac/publ/proto-sys-explain.pdf. Draft of submission to ThEdu at CADE'26, 2017 (cit. on p. 2).

[14] Walther Neuper. "Technology of Deduction for "Systems that Explain Themselves"". In: http://www.ist.tugraz.at/projects/isac/publ/ded-for-sys-explain.pdf. Draft of submission to ThEdu at CADE'26, 2017 (cit. on p. 2).

[15] Walther Neuper and Christian Dürnsteiner. *Angewandte Mathematik und Fachtheorie mithilfe adaptierter Basis-Software*. Tech. rep. 683. https://www.imst.ac.at/imst-wiki/images/f/f9/683_Kurzfassung_Neuper.pdf. University of Klagenfurt, Institute of Instructional and School Development (IUS), 9010 Klagenfurt, Sterneckstrasse 15: IMST – Innovationen Machen Schulen Top!, 2007 (cit. on p. 20).

[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL, a proof assistant for high-order logic*. Springer Verlag, 2008 (cit. on pp. 12, 13).

[17] Martin Odersky and al. *An Overview of the Scala Programming Language*. Tech. rep. IC/2004/64. EPFL Lausanne, Switzerland, 2004 (cit. on p. 26).

[18] Luca Padovani. *On the Roles of LATEX and MathML in Encoding and Processing Mathematical Expressions*. 1st ed. Second International Conference, MKM 2003 Bertinoro, Italy: Springer-Verlag, 2003 (cit. on p. 7).

[19] L. C. Paulson and K. W. Susanto. "Source-level proof reconstruction for interactive theorem proving". In: *TPHOLs 2007*. Vol. 4732. LNCS. Springer Verlag, 2007, pp. 232–245 (cit. on p. 18).

[20] Lawrence C. Paulson and Jasmin C. Blanchette. *Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers*. http://people.mpi-inf.mpg.de/~jblanche/iwil2010-sledgehammer.pdf. 2010 (cit. on p. 18).

[21] Mojzesz Presburger. "Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt". In: *Comptes Rendus du I congrés de Mathématiciens des Pays Slaves*. 1929, pp. 29–60 (cit. on p. 37).

[22] Michael A. Schoonover, John S. Bowie, and William R. Arnold. *GNU Emacs: UNIX Text Editing and Programming*. Hewlett-Packard Press Series. Reading, Menlo Park, New York, Don Mills, Wokingham, Amsterda, Bonn, Paris, Milan, Madrid, Sydney, Singapore, Tokyo, Seoul, Taipei, Mexico City, San Juan: Addison-Wesley Publishing Company, 1992 (cit. on p. 12).

[23] Li Lian Su Wei Paul S.Wang. *An On-line MathML Editing Tool for Web Applications*. 1st ed. IEEE, 2007 (cit. on p. 7).

[24] ISAC Team. *TheRules ofthe ISAC-Developers*. 2016. URL: http://www.ist.tugraz.at/projects/isac/publ/isac-rules.pdf (cit. on p. 49).

[25] Makarius Wenzel. "Asynchronous User Interaction and Tool Integration in Isabelle/PIDE". In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. 2014, pp. 515–530. URL: http://dx.doi.org/10.1007/978-3-319-08970-6_33 (cit. on pp. 13, 19).

[26] Makarius Wenzel. "Parallel Proof Checking in Isabelle/Isar". In: *ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS)*. Ed. by Dos Reis and L. Théry. Munich: ACM Digital library, Aug. 2009 (cit. on p. 13).

[27] Makarius Wenzel. "System description: Isabelle/jEdit in 2014". In: *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014*. 2014, pp. 84–94. URL: http://dx.doi.org/10.4204/EPTCS.167.10 (cit. on pp. 13, 29).

[28] Makarius Wenzel. *The Isabelle/Isar Reference Manual*. 2016. URL: http://isabelle.in.tum.de/dist/Isabelle2016-1/doc/isar-ref.pdf (cit. on p. 49).

[29] Markus Wenzel. "Isar - a Generic Interpretative Approach to Readable Formal Proof Documents". In: *Theorem Proving in Higher Order Logics*. Ed. by G. Dowek et al. LnCS 1690. 12th International Conference TPHOLs'99. Springer, 1999 (cit. on p. 22).

Online sources

[30] R. Ausbrooks. Dec. 2016. URL: http://www.w3.org/TR/MathML3 (cit. on p. 7).