PROOFTOOL: a GUI for the GAPT Framework*

Cvetan Dunchev[1] Alexander Leitsch[1] Tomer Libal[1] Martin Riener[1]
Mikheil Rukhaia[1] Daniel Weller[2]
Bruno Woltzenlogel-Paleo[1]

[1] Institute of Computer Languages (E185),
[2] Institute of Discrete Mathematics and Geometry (E104),
Vienna University of Technology
{cdunchev,leitsch,shaolin,riener,mrukhaia,weller,bruno}@logic.at

This paper introduces PROOFTOOL, the graphical user interface for the General Architecture for Proof Theory (GAPT) framework. Its features are described with a focus not only on the visualization but also on the analysis and transformation of proofs and related tree-like structures, and its implementation is explained. Finally, PROOFTOOL is compared with three other graphical interfaces for proofs.

1 Introduction

GAPT[1] (General Architecture for Proof Theory) is a framework that aims at providing data-structures, algorithms and user interfaces for analyzing and transforming formal proofs. GAPT was conceived to allow general tools for creating, processing, displaying and transforming structured proofs and one of its first goals was to replace the CERES system[2] [8, 12, 13] and expand its scope beyond the original focus on cut-elimination by resolution for first-order logic [4]. Through a more flexible and succint implementation based on basic data structures for simply-typed lambda calculus and for generic sequent calculus style proofs, in the hybrid functional object-oriented language Scala [18], this goal has already been achieved: generalizations of the CERES method (cut-elimination by resolution) to proofs in higher-order logic [9] and to schematic proofs [6], as well as methods for structuring and compressing proofs, such as Herbrand sequent extraction [11, 20] and cut-introduction [10] have been implemented in GAPT.

The GAPT system provides a *command line interface* (CLI) that allows the user to access all functionality of the system, e.g. to create and manipulate proofs and a *graphical user interface* called PROOFTOOL[3]. The CLI is very flexible, because it is built on top of Scala's REPL (Read-Evaluate-Print Loop) Interpreter[4]; it is not suited for the visualization of proofs, though, since proofs usually are large, contain specialized mathematical symbols and have a tree-like structure that is inconvenient to read as pure text.

Convenient proof visualization, PROOFTOOL's main goal, is achieved by means of several features (e.g. tree rendering, scrolling, hiding and highlighting, LaTeX rendering), which are described in Section 5. But PROOFTOOL is more than just a proof viewer. As explained in Section 4, most of GAPT's

*Partially supported by the project I383 of the Austrian Science Fund and the Vienna PhD School of Informatics.

[1] GAPT homepage: `http://www.logic.at/gapt/`

[2] CERES homepage: `http://www.logic.at/ceres/`

[3] Both binaries are available in the Download section on the GAPT homepage; the current download links at the time of publication are `http://www.logic.at/gapt/gapt-cli-1.4` and `http://www.logic.at/gapt/prooftool-1.4`.

[4] Scala's REPL Interpreter: `http://www.scala-lang.org/node/2097`

C. Kaliszyk and C. Lüth (Eds.): 10th International Workshop
on User Interfaces for Theorem Provers
EPTCS 118, 2013, pp. 1–14, doi:10.4204/EPTCS.118.1

features for analyzing and transforming proofs are already available within PROOFTOOL, and we follow the policy of adding new features of GAPT to PROOFTOOL continuously, soon after they reach a stable stage of development.

The fact that PROOFTOOL is more than a proof viewer and needs to support the variety of structural proof theory algorithms allowed in GAPT is the main reason why simply reusing existing proof viewers, such as LOUI [22] and IDV [24], was not a suitable option. Another benefit of the close connection between PROOFTOOL and GAPT is that even GAPT's more exotic objects (e.g. schematic proofs) can be easily displayed and manipulated in PROOFTOOL. It would be hard, if not impossible, to achieve the same functionality on such objects using an external proof viewer. Nevertheless, a comparison between these different systems is available in Section 7. A last but nonetheless important aim is to allow the proof theory community to extend GAPT's algorithms and data structures. This allows immediate access to the relevant PROOFTOOL features.

2 Preliminaries

Most kinds of objects that can be displayed by and manipulated within PROOFTOOL are very specific to the proof mining algorithms implemented in GAPT, and especially to the CERES algorithm of cut-elimination by resolution [4, 5]. Among the data structures used by CERES and displayable by PROOFTOOL, the most prominent are:

- **Struct**: an unsatisfiable formula obtained from a proof with cuts; when seen as a tree, it has the same branching structure as the proof with cuts.
- **Characteristic clause set**: the clause set obtained by transforming the struct into clause form.
- **Projection**: a cut-free part of the proof with cuts; each projection corresponds to one clause of the characteristic clause set.
- **Resolution Refutation**: a refutation of the characteristic clause set; when combined with the projections, an essentially cut-free proof is obtained.
- **Herbrand sequent**: a propositionally valid and quantifier-free sequent made of formulas that are instances of the formulas present in the end-sequent of a cut-free proof; it serves as a compact representation of the first-order content of a proof.

2.1 Proof Schemata

Formula schemata were introduced and investigated in [1, 3]. A subclass called *regular schemata* was identified and shown decidable, and a tableau calculus STAB was defined and implemented [2]. A *sequent calculus proof schema* is a primitive recursively specified infinite sequence of sequent calculus proofs, where sequents are multisets of formula schemata. PROOFTOOL is able to display a proof schema as a first-class object, with no need to instantiate it to a particular element of the infinite sequence of proofs it represents. This is made possible with *proof links*: a special axiom rule that is responsible for the recursion. A schematic CERES method for eliminating cuts from proof schemata has been developed [6] and all its required data structures (e.g. schematic structs, schematic characteristic clause sets, schematic refutations and schematic projections) can be displayed in PROOFTOOL as well.

3 Supported Input/Output Formats

There are several input formats parsed by the GAPT system but the most important ones are the *Handy LKS* language, the *Ceres XML* format and a less restrictive, so called *Simple XML* format. They will be described below.

The GAPT system supports the `cnf` and `fof` subset of the TPTP format [23] and the IVY proof checker format [17]. TPTP exporting is used to communicate with external theorem provers, the IVY format is used to import proofs created by PROVER9.

In practical applications, the input files quickly reach sizes over one MB. Therefore compression was implemented in the form of the *.gz* file reader. PROOFTOOL supports parsers for *.lks*, *.xml* and *.ivy* files in gzipped form as well.

In PROOFTOOL there are several exporters for objects of the GAPT system, most importantly LaTeX and *.pdf* exports. All objects from PROOFTOOL can be exported directly into *.pdf* files and additionally, the proofs and clause sets can be exported into *.tex* files as well.

3.1 Ceres XML

Ceres XML is a file format used by the CERES system. It can be used to encode DAG-like proofs in a second-order language. Since the GAPT system can deal with proofs which fall outside of this language (namely proofs in full higher-order logic), *Ceres XML* is intended to be replaced by a more general format in the future. Still, the format has been successfully used in our experiments with first- and second-order proofs, and for this reason we include its description here. The *Ceres XML* format is mainly described by a Document Type Definition (DTD), which fixes the set of XML files which are considered as *Ceres XML* . The DTD of the format is quite detailed and restrictive; a part of it is shown in Figure 1.

The DTD of the *Ceres XML* format allows a partial correctness check of proofs in XML format: For example, if an XML file conforms to the DTD, then all the formulas occuring in it are well-formed. On the other hand, since the actual calculus rules are not specified, checking for DTD conformity does not suffice for proof-checking.

GAPT's capabilities with respect to the *Ceres XML* format are asymmetric: GAPT can read sequent calculus proofs containing explicit permutation rules represented in the format, while it outputs (for technical reasons) sequent calculus proofs containing implicit permutations. This asymmetry is due to backwards-compatibility with the CERES system and will be removed in the near future.

3.2 Simple XML

Simple XML is a simplified version of *Ceres XML* (i.e. its DTD has a much simpler structure and less restrictions, see Figure 2) and is used to represent arbitrary tree-like proofs (e.g. natural deduction, tableaux, etc). While *Ceres XML* fixes a certain language for the formulas occuring in the proofs it encodes, the language of formulas is left open in *Simple XML* . This allows the implementation of the proof parser (which follows the structure of the DTD) to be separated naturally from the implementation of the formula parser (which is unrestricted in principle). This makes it very easy to exchange proofs if proof checking is not desired, since then a generic prooftree-parser can be combined with a parser for the language that is used in the proof. The *Simple XML* format was exploited to get a straightforward and easy implementation of the parsing of proofs produced by REGSTAB [2], which is a STAB prover, producing tableau refutations for formula schemata.

```
<!ELEMENT proofdatabase (definitionlist,axiomset,proof*,
  sequentlist*,variabledefinitions)>
<!ELEMENT proof (rule)>
<!ATTLIST proof
          symbol CDATA #REQUIRED
          calculus CDATA #IMPLIED >
<!ELEMENT rule (sequent,(rule|prooflink)*,
 substitution?,lambdasubstitution?)>
<!ATTLIST rule
          symbol CDATA #IMPLIED
          type   CDATA #REQUIRED
          param  CDATA #IMPLIED >
<!ELEMENT prooflink EMPTY>
<!ATTLIST prooflink
          symbol CDATA #REQUIRED >
<!ELEMENT sequentlist (sequent+)>
<!ATTLIST sequentlist
          symbol CDATA #REQUIRED >
<!ELEMENT sequent (formulalist,formulalist)>
<!ATTLIST sequent
          projection CDATA #IMPLIED >
<!ENTITY % atomformula '(constantatomformula|variableatomformula)'>
<!ENTITY % formula
  '(formulavariable|conjunctiveformula|quantifiedformula|
    secondorderquantifiedformula|%atomformula;)'>
```

Figure 1: A piece of the DTD file of *Ceres XML*

There are disadvantages of using the *Simple XML* format: since it is a very general format, structural details about the nodes of the proof are absent. Therefore they are displayed simply as the strings which occur in the XML file instead of a more easily readable LaTeX rendering. This also means some advanced features of PROOFTOOL might not work. Basic features zooming and scrolling are not affected but certain views might not be available.

3.3 Handy LKS

Since proofs are the main input for the GAPT system, a comfortable proof input language is important. In the CERES system, the Handy LK language[5] was used as that language, and it allowed the input of proof schemata (by this we mean infinite sequences of proofs specified in an inductive way) in a second-order language.

Unfortunately the Handy LK language and its implementation had several shortcomings. First and foremost, it did not support full higher-order logic, and its implementation of proof schemata was not based on a formal theory. In particular, there was no logical notion of a "proof schema" object —

[5] http://www.logic.at/hlk/

```
<!ELEMENT prooftrees (proof*)>
<!ELEMENT proof (rule)>
<!ATTLIST proof
          symbol CDATA #REQUIRED
          calculus CDATA #IMPLIED >
<!ELEMENT rule
          (conclusion,(rule|prooflink)*)>
<!ATTLIST rule
          symbol CDATA #IMPLIED
          type CDATA #REQUIRED
          param CDATA #IMPLIED >
<!ELEMENT conclusion (#PCDATA)>
<!ELEMENT prooflink EMPTY>
<!ATTLIST prooflink
          symbol CDATA #REQUIRED >
```

Figure 2: The full DTD file of *Simple XML*

only the instances of the general schema were treated in a logical way. Furthermore, the Handy LK compiler, which generated a proof in *Ceres XML* from a proof written in Handy LK, was implemented separately from other parts of the CERES system, causing several practical problems. Finally, the Handy LK compiler did not allow for a very fine-grained control over the resulting sequent calculus proofs, but such control, while unimportant for the general task of formalizing proofs, can be important when experimenting with proofs in the sense of structural proof-theory.

The replacement of Handy LK in GAPT is the *Handy LKS* language. This language is an implementation of the schematic proofs defined and investigated in [7]. The format is simple enough that an input proof can be written in any text editor. To differentiate it from other text files, the extension *.lks* is used. The full description of the grammar of the language as well as of the formal calculus, can be found in [6].

An *.lks* file must contain at least one proof definition, an example of which is given in Figure 3. For an inductive proof definition, the `base` block describes the base case and the `step` block describes the recursive case. The IDs are arbitrary labels that are unique within the scope of $\{ \ldots \}$ blocks (i.e. the same labels can be used in the definition of base and step cases) and rules are tuples consisting of the rule's name, the IDs of the premises and of the auxiliary formulas. The `autoprop` keyword is used to prove propositional sequents automatically if the user is not interested in and does not want to give the exact proof of that sequent.

At the moment, *Handy LKS* is geared towards writing inductively defined sequences of proofs. Of course, non-inductively defined proofs (e.g. just a single LK proof) can be trivially represented as inductively-defined proofs. We plan to add syntactic sugar to the *Handy LKS* format to allow encoding of such simpler proofs in a natural way.

As the language would profit from syntax highlighting, having an editor for this language would be convenient. One solution is to use XText[6] and create such an editor using the grammar of the *Handy LKS* language. An advantage of the grammar is that it is easy to give the exact line numbers where the

[6]XText homepage: `http://www.eclipse.org/Xtext/`

```
proof \psi proves A(0), BigAnd(i=0..k) (~A(i) \/ A(i+1)) |- A(k+1)
base {
  1: autoprop(A(0), (~A(0) \/ A(1)) |- A(1))
  root: andEqL3(1, (~A(0) \/ A(1)),  BigAnd(i=0..0) (~A(i) \/ A(i+1)))
}
step {
  1: pLink((\psi, k) A(0), BigAnd(i=0..k) (~A(i) \/ A(i+1)) |- A(k+1))
  2: ax(A(k+1) |- A(k+1))
  3: negL(2, A(k+1))
  4: ax(A(k+2) |- A(k+2))
  5: orL(3, 4, ~A(k+1), A(k+2))
  6: cut(1, 5, A(k+1))
  7: andL(6, BigAnd(i=0..k) (~A(i) \/ A(i+1)), (~A(k+1) \/ A(k+2)))
  root: andEqL1(7, (BigAnd(i=0..k) (~ A(i) \/ A(i+1)) /\
          (~ A(k+1) \/ A(k+2))), BigAnd(i=0..k+1) (~ A(i) \/ A(i+1)))
}
```

Figure 3: An example of a proof schema in *Handy LKS* .

parsing of a file fails because of a syntactic error.

4 Features for Proof Mining

At the moment, GAPT distinguishes two kinds of sequent-like proofs: first- and higher-order sequent calculus (LK) proofs and schematic first-order sequent calculus (LKS) proofs. In PROOFTOOL there are separate menus for LK- and LKS-proofs, containing the possible operations for these proofs respectively. The available functionality from these menus are the following:

- For LK-proofs:
  - Compute the data-structures of the CERES method, such as struct and characteristic clause set.
  - Apply reductive cut-elimination (Gentzen's method). This option is available as well in the context menus of sequents occurring in the proof and applies cut-elimination to the proof ending in that sequent.
  - Extract Herbrand sequent from a cut-free proof.
  - Skolemize and/or regularize a proof.
- For the LKS-proofs:
  - Compute the data-structures of the schematic CERES method, such as schematic struct, schematic characteristic clause set (see Figure 7) and schematic projection term (see Figure 5).
  - Compute the instance of a schematic proof, of a schematic struct or of a projection term for a particular number given by the user.

Sometimes it is useful to get a list of the lemmas which are used in the proof. Therefore, there is a menu item extracting cut-formulas from LK and dedicated to LKS proofs.

The menus currently support only a subset of the capabilities of the GAPT system. The other functionalities will be added to PROOFTOOL on a by-need basis.

5 Visualization Features

PROOFTOOL is a graphical user interface, used to display objects generated by the GAPT system. These objects are: trees, proofs, sequents, formulas and the like. For example, the proof given in Figure 3 is displayed in PROOFTOOL as shown in Figure 4.

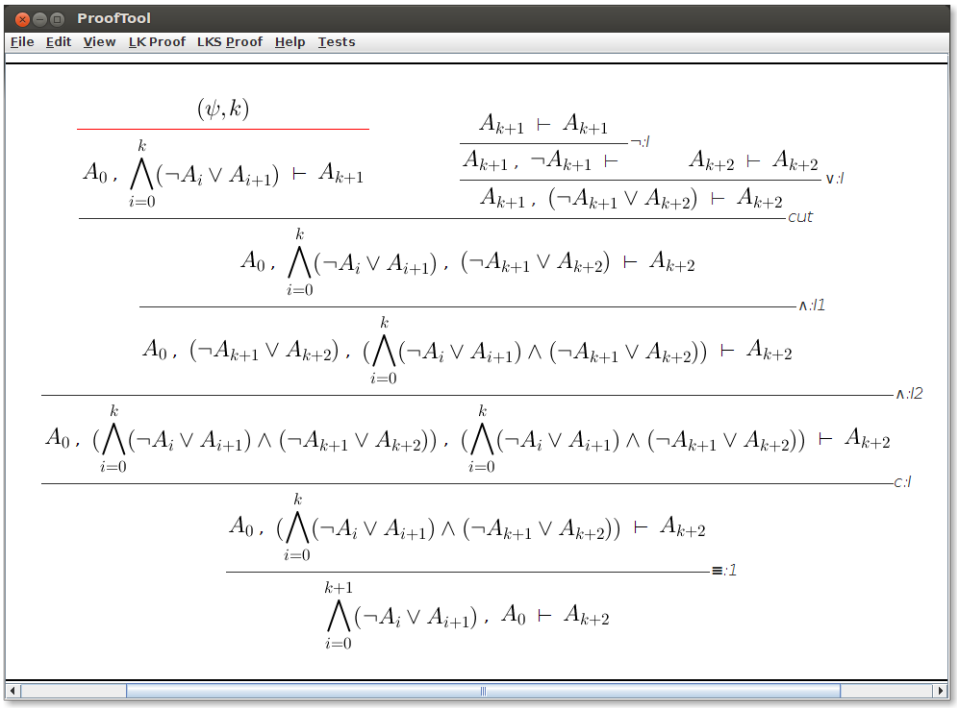

Figure 4: Proof in PROOFTOOL.

A TREEPROOF is a binary tree in the GAPT system which represents a tree-like proof. It is a TREE which is also a PROOF. This means that we can expect that nodes are labeled with sequents. In general, proofs in the GAPT system are sequent-like proofs. The reasons for the decision to display TREEPROOFs instead of PROOFs are as follows: A PROOF is only a directed acyclic graph, so additional arrows for shared structures would be necessary. Apart from sequent calculus proofs, at the moment only resolution proofs need to be displayed; The tree representations of these proofs can easily be obtained from the DAG-form.

The following features of PROOFTOOL provide a better visualization of proofs:

Hide structural rules has the purpose to shorten the size of proofs. In most of the cases structural rules do not contain any valuable information and they can be hidden for the user. The Edit>Hide Structural Rules menu item allows access to this feature.

Hide sequent context is used when the sequents occurring in a proof are very large. In such cases, for each inference in the proof, only its active formulas are shown.

Mark cut-ancestors highlights all ancestors of all cut-formulas occurring in the proof. It allows the user to have a glimpse of how cut-formulas are structured in the derivation.

Split proof allows focusing on a subproof of the proof. To shorten the size of the proof, it is also possible to hide/unhide some subproofs of it. These manipulations can be done from the context menus of the sequents.

Unlike proofs, TREEs are binary trees in the system and they are displayed upside-down (see Figure 5). The user can manipulate the size of the tree by hiding/showing some branches or leaves. This is done by calling the corresponding menu items of the Edit menu, or by clicking on the vertex that should be hidden/shown (see Figure 6).

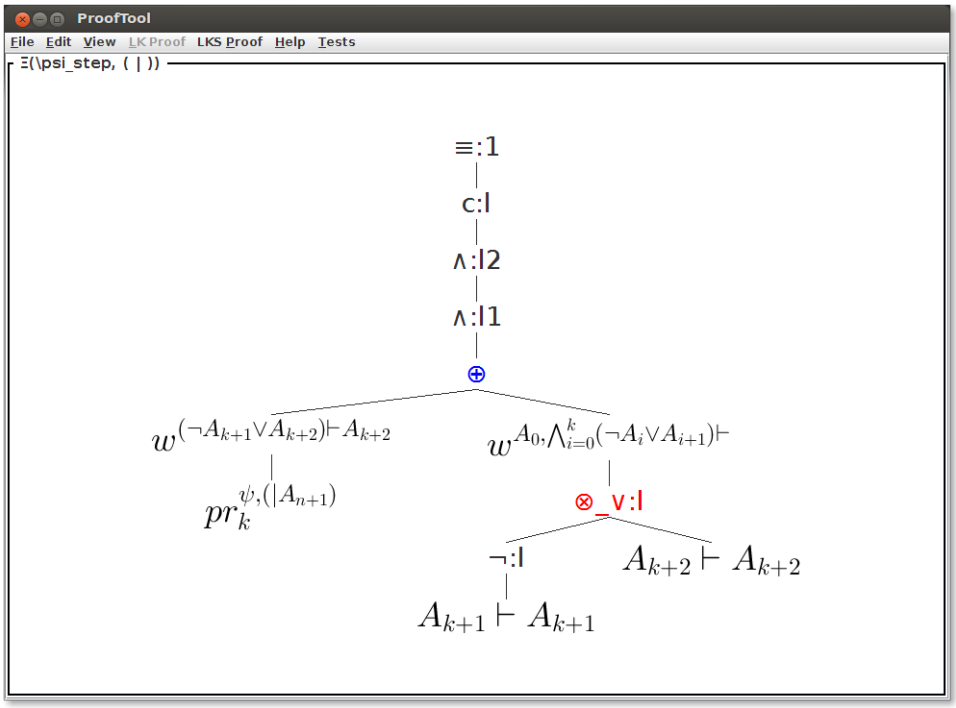

Figure 5: Tree in PROOFTOOL.

For tree-like objects like clause terms or projection terms, the system contains transformations to trees. A simple example is a directed acyclic graph where the corresponding tree just has duplicates of the shared structures.

Lists are very important data-structures and it is worthy to have specialized handling for them. In PROOFTOOL each element of a list is displayed in a single line. Lines are separated with semicolons. The most commonly occurring lists in the GAPT system are *sequent lists* and *definition lists*. For an example of a sequent list displayed using PROOFTOOL see Figure 7.

For the output to the user to be more readable, sometimes we abbreviate long expressions with shorter ones. These abbreviations are stored in the GAPT system as a list of definitions. An example of the definition list for the clause set from Figure 7 is shown in Figure 8.

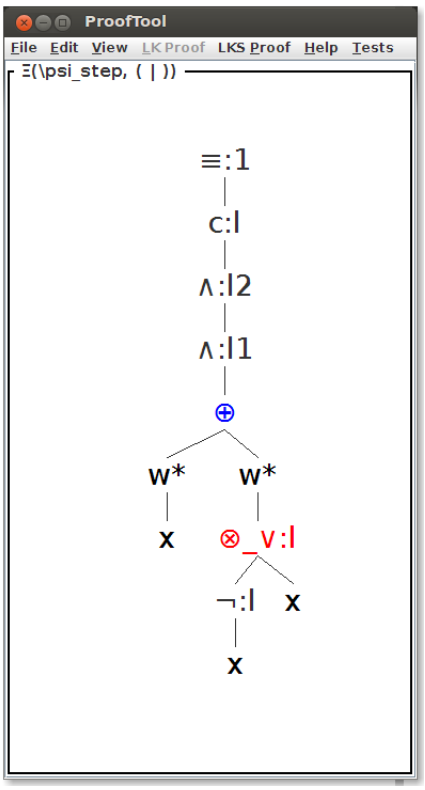

Figure 6: The same tree in PROOFTOOL (leaves are hidden).

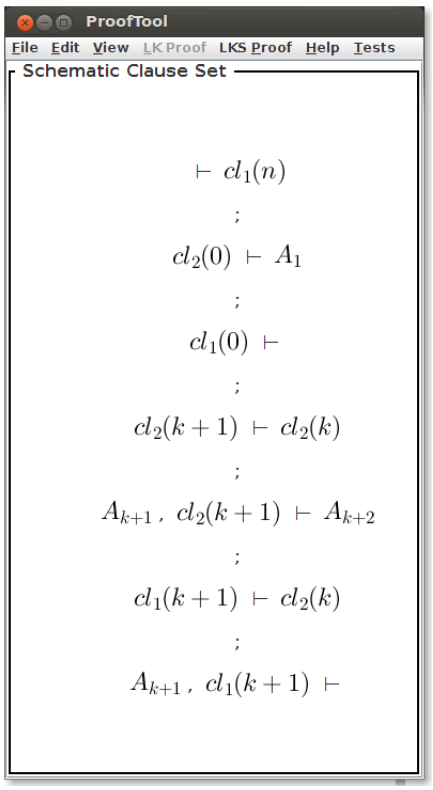

Figure 7: Sequent list in PROOFTOOL.

Thanks to a general design, which will be discussed in the next section, the basic features – zooming and scrolling, can be applied to all objects easily. One more feature that is applicable to all the objects discussed above is searching.

Search is very useful when one has to deal with huge proofs or other objects, which is often the case in our research. In PROOFTOOL, it is currently done in the following way: a user calls the Search dialog from the Edit menu and types the desired term (in LaTeX). The desired term is then searched in the string representation of the displayed object and any occurrences found are highlighted in the displayed object. For example, when searching for `cut` in the proof displayed in Figure 4, PROOFTOOL finds the inference name and colors it green.

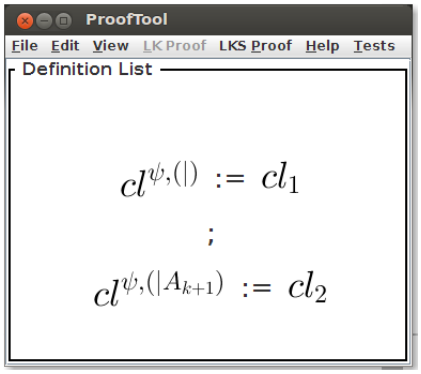

Figure 8: Definition list in PROOFTOOL.

A problem occurring during search is that the string representation of a node might be quite different from its rendering. When searching for the formula $\neg P_{k+1}$, the exact LaTeX representation `\neg␣P_{k+1}` needs to be entered. Counter-intuitively, the search for `\neg␣␣P_{k+1}` fails even if the rendered formula looks the same. To solve this problem, PROOFTOOL allows the user to see the correct LaTeX

representation `\neg␣P_{k+1}` of (one of) the formulas $\neg P_{k+1}$ by right double clicking on it. The LaTeX string can then be directly copied to the search dialog and all occurrences of $\neg P_{k+1}$ are highlighted. A drawback of this solution is that the user has to manually find an occurrence of the object he looks for. Avoiding this drawback would require approximate string matching [19], but for our current needs, this is not necessary.

6 Implementation Details

The GAPT system is implemented in the programming language Scala. We rely heavily on functional features of Scala such as pattern matching. The parts of Scala's library which are of most used are the XML library, the combinator parser library and the Scala frontend for Java's SWING library. JLATEXMATH[7] and ITEXT[8] are the external libraries PROOFTOOL depends on.

The domain specific language for XML that Scala provides reduces the parsing of our XML based input formats to pattern matching of XML tags and taking care of the differences in proof representations between *Ceres XML* and GAPT's internal datastructures. The remaining formats are read by combinator parsers which are also seamlessly integrated into the Scala language. Since most parts of our grammars are LL(1), we can benefit from annotations of non-backtracking rules but don't get much improvement of memoizing techniques as implemented in Scala's packrat parsers [14].

Usage of the combinator parsers is shared with the GAPT's interactive theorem prover TAP and the libraries connecting external theorem provers like VAMPIRE [21] and PROVER9 [16]. In the case of PROVER9 the proof is first converted to the IVY proof checker format by PROVER9's PROOFTRANS utility. IVY proofs are represented as Lisp S-expressions for which GAPT utilizes combinator parsers for reading arbitrary S-expressions. Afterwards the resolution refutation is created from the IVY specific structure by pattern matching S-expressions.

The PDF exporter in PROOFTOOL is written using the ITEXT library. It is a Java library for creating and manipulating the PDF files. We use its implementation of the abstract GRAPHICS2D class to generate a *.pdf* file of the object drawn on the screen.

PROOFTOOL is implemented using the SCALA.SWING library [15]. It is a SIMPLESWINGAPPLICATION and consists of one frame, which contains a MENUBAR and a SCROLLPANE. A brief description of the architecture of PROOFTOOL is shown in Figure 9.

SCROLLPANE has a component called LAUNCHER which extends GRIDBAGPANEL and takes the following parameters:

- A pair (STRING, ANYREF), where ANYREF is an object that should be displayed and STRING is a name of the object. The object has TITLEDBORDER around it and the name is the title.
- An INT, which is a size of font that is used to display an object.

When an object is passed to LAUNCHER it uses Scala's matching mechanism to recognize its structure and instantiates the corresponding class responsible for the drawing of the object. Basically, LAUNCHER differentiates between three kinds of objects: trees, proofs and lists. The rendering classes are DRAWTREE, DRAWPROOF and DRAWLIST respectively.

DRAWTREE extends the BORDERPANEL class and displays a tree in the following way: A leaf node just renders the vertex, whereas an inner node draws the root node at the top and then creates new

[7] JLATEXMATH homepage: `http://forge.scilab.org/index.php/p/jlatexmath/`

[8] ITEXT homepage: `http://www.itextpdf.com`

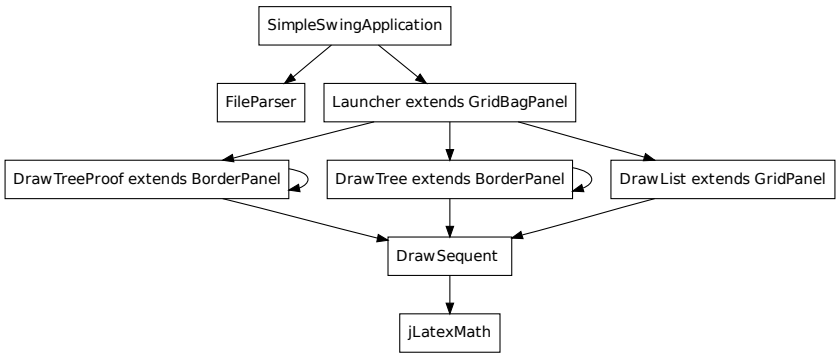

Figure 9: Architecture of PROOFTOOL.

DRAWTREE instances for the child trees. For a binary node, the branches of the tree are rendered side-by-side on a frame and, for a unary node, the branch is rendered in the center. The root is connected to its children by straight lines.

DRAWPROOF also extends the BORDERPANEL class and behaves similar to DRAWTREE. The difference is that the desired output looks like a sequent calculus proof. This means that while DRAWTREE puts the vertex at the top and the branches below, DRAWPROOF puts the vertex at the bottom and the branches on top of it. Then it draws horizontal lines between vertices and puts the rule names next to the lines.

DRAWLIST extends the GRIDPANEL class having only one column and puts each element of the list in a separate cell.

In PROOFTOOL a sequent is displayed by a FLOWPANEL which contains the representation of each formula as a separate LABEL.

To handle formulas it was decided to use JLATEXMATH. It is a Java API which displays mathematical formulas written in LaTeX as images, which was also used in the GUI of Scilab[9]. The drawback of this library is that having an image for every formula is quite memory consuming.

For each formula displayed a label is created. Then the formula is transformed into a LaTeX string and rendered using JLATEXMATH. The resulting image is assigned as an icon to the label. Since this rendering is expensive, we use it only when necessary and display simple string representations otherwise. For example, any vertex containing a higher-order expression of type HOLEXPRESSION (which also includes first-order and schema formulas) is rendered by JLATEXMATH. Other types of vertices are displayed simply as their Scala string representation which often suffices since the Unicode character-set used includes Greek letters as well as other logical symbols.

PROOFTOOL strongly profits from the SCALA.SWING wrapper library which simplifies event handling significantly. Since there are only PUBLISHER and REACTOR elements instead of Java's complicated event handling mechanism, for instance menu items only need to listen to the PROOFTOOLPUBLISHER to adjust their activation status accordingly. In the case a new file is loaded or the proof database is dynamically changed the PROOFDBCHANGED event is issued. Since also the View>View Proof, View>View Clause List and View>View Term Tree menus listen to this event, they can refresh their list of MENUITEMs.

[9]Scilab homepage: `http://www.scilab.org/`

7 Conclusions and Future Work

In the previous sections, several features of PROOFTOOL were described. These features are summarized in Figure 10. PROOFTOOL's predecessor, known as `prooftool`, is also shown. In comparison, it is noticeable that PROOFTOOL's feature set is significantly broader than `prooftool`'s. While PROOFTOOL can display general tree-like structures, such as structs and projection terms, `prooftool` can only display sequent calculus proofs. Even when `prooftool` displays a list of sequents, for instance, what is displayed is actually a fake proof with dummy unary inferences connecting the sequents. Moreover, PROOFTOOL accepts more input and output proof formats than `prooftool`. Most importantly, proof schemata can now be handled and displayed directly, while `prooftool` could only display traditional sequent calculus proofs, and hence only particular instances of proof schemata. Finally, while `prooftool` was just a proof viewer, PROOFTOOL gives access to GAPT's proof mining algorithms, such as cut-elimination and Herbrand sequent extraction.

Features		PROOFTOOL	`prooftool`	LOUI	IDV
Input/Output	Parse ceres XML	✓	✓		
	Parse simple XML	✓			
	Parse TPTP/TSTP	*		✓	✓
	Export to TSTP	*			
	Export to .tex	✓	✓		
	Export to .tptp	✓		✓	✓
	Export to .pdf	✓			
Visualization	Zooming, scrolling	✓	✓	✓	✓
	Sequent calculus proofs	✓	✓		
	Sequent lists	✓	✓		
	Definition lists	✓			
	Proof schemata	✓			
	Trees and related features	✓		✓	
	DAGs and related features	*		✓	✓
Proof Mining	Marking (cut-)formula ancestors	✓			✓
	Extracting cut-formulas	✓			
	Extracting Herbrand sequent	✓			
	Hiding sequent context	✓			
	Hiding structural rules	✓			
	Search	✓			
	Split/unsplit	✓	✓	✓	✓
	Substitute/unsubstitute		✓		
	Proofs in natural language			✓	
	Cut-elimination (Gentzen, CERES)	✓			
	Skolemization, regularization	✓			

Figure 10: Features implemented by PROOFTOOL and other similar systems.

LOUI and IDV are also shown, but care should be taken when using this table for comparison. As the table is focused on features available in PROOFTOOL and relevant for GAPT's needs, it is most probably the case that this table is lacking features implemented by LOUI or IDV, but not by PROOFTOOL. Moreover, the fact that IDV and LOUI do not implement some listed features implemented by PROOFTOOL

should not be considered as a flaw of IDV and LOUI, since most of these features are very specific to GAPT's needs. Nevertheless, the table does include a few of IDV's or LOUI's features that are relevant for GAPT but not yet implemented by PROOFTOOL. Those marked with "*" will be the focus of imminent future work.

Besides the implementation of features marked with "*" in the table, another task that remains for the future is the improvement of the *Handy LKS* language aiming at unifying parsing of schematic and traditional sequent calculus proofs. Furthermore, there is plenty of room for decreasing the amount of detail that *Handy LKS* requires in the proof specifications, beyond what is currently possible with the "autoprop" feature. In particular, the next version of *Handy LKS* intends to additionally allow the omission of structural inferences such as contraction and weakening, which are poor in mathematical content.

Acknowledgements

We would like to thank the anonymous referees and all the participants of the 10th Workshop on User Interfaces for Theorem Provers for their useful comments and suggestions about this work. They helped us to improve not only the contents and presentation of this paper, but also the PROOFTOOL program.

References

[1] Vincent Aravantinos, Ricardo Caferra & Nicolas Peltier (2009): *A Schemata Calculus for Propositional Logic*. In: *Automated Reasoning with Analytic Tableaux and Related Methods*, *Lecture Notes in Computer Science* 5607, Springer, pp. 32–46, doi:10.1007/978-3-642-02716-1_4.

[2] Vincent Aravantinos, Ricardo Caferra & Nicolas Peltier (2010): *RegSTAB: A SAT-Solver for Propositional Iterated Schemata*. In: *International Joint Conference on Automated Reasoning*, pp. 309–315, doi:10.1007/978-3-642-14203-1_26.

[3] Vincent Aravantinos, Ricardo Caferra & Nicolas Peltier (2011): *Decidability and Undecidability Results for Propositional Schemata*. *Journal of Artificial Intelligence Research* 40, pp. 599–656, doi:10.1613/jair.3351.

[4] Matthias Baaz & Alexander Leitsch (2000): *Cut-elimination and Redundancy-elimination by Resolution*. *Journal of Symbolic Computation* 29(2), pp. 149–176, doi:10.1006/jsco.1999.0359.

[5] Matthias Baaz & Alexander Leitsch (2006): *Towards a clausal analysis of cut-elimination*. *Journal of Symbolic Computation* 41(3-4), pp. 381–410, doi:10.1016/j.jsc.2003.10.005.

[6] Cvetan Dunchev, Alexander Leitsch, Mikheil Rukhaia & Daniel Weller (2010–2012): *About Schemata And Proofs web page*. `http://www.logic.at/asap`.

[7] Cvetan Dunchev, Alexander Leitsch, Mikheil Rukhaia & Daniel Weller (2012): *CERES for First-Order Schemata*. *submitted to the Journal of Logic and Computation*.

[8] Tsvetan Dunchev, Alexander Leitsch, Tomer Libal, Daniel Weller & Bruno Woltzenlogel Paleo (2010): *System Description: The Proof Transformation System CERES*. In Jürgen Giesl & Reiner Hähnle, editors: *Automated Reasoning*, *Lecture Notes in Computer Science* 6173, Springer Berlin Heidelberg, pp. 427–433, doi:10.1007/978-3-642-14203-1_36.

[9] Stefan Hetzl, Alexander Leitsch & Daniel Weller (2011): *CERES in higher-order Logic*. *Annals of Pure and Applied Logic* 162(12), pp. 1001–1034, doi:10.1016/j.apal.2011.06.005.

[10] Stefan Hetzl, Alexander Leitsch & Daniel Weller (2012): *Towards Algorithmic Cut-Introduction*. In Nikolaj Bjørner & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 228–242, doi:10.1007/978-3-642-28717-6_19.

[11] Stefan Hetzl, Alexander Leitsch, Daniel Weller & Bruno Woltzenlogel Paleo (2008): *Herbrand Sequent Extraction*. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki & Freek Wiedijk, editors: *Intelligent Computer Mathematics*, *Lecture Notes in Computer Science* 5144, Springer Berlin, pp. 462–477, doi:10.1007/978-3-540-85110-3_38.

[12] Stefan Hetzl, Alexander Leitsch, Daniel Weller & Bruno Woltzenlogel Paleo (2008): *Proof Analysis with HLK, CERES and ProofTool - Current Status and Future Directions*. In Geoff Sutcliffe, Simon Colton & Stephan Schulz, editors: *Proceedings of the CICM Workshop on Empirically Successful Automated Reasoning in Mathematics*, pp. 23–41.

[13] Stefan Hetzl, Alexander Leitsch, Daniel Weller & Bruno Woltzenlogel Paleo (2008): *Transforming and Analyzing Proofs in the CERES-system*. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate Schmidt & Stephan Schulz, editors: *Proceedings of the LPAR 2008 Workshops Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, pp. 77–91.

[14] Manohar Jonnalagedda (2009): *Packrat Parsing in Scala.* Technical Report, École Polytechnique Fédérale de Lausanne. Available at `http://scala-programming-language.1934581.n4.nabble.com/attachment/1956909/0/packrat_parsers.pdf`.

[15] Ingo Maier (2009): *The scala.swing package*. `http://www.scala-lang.org/sid/8`.

[16] William McCune (2005–2010): *Prover9 and Mace4*. `http://www.cs.unm.edu/~mccune/prover9/`.

[17] William McCune & Olga Shumsky (2000): *IVY: A Preprocessor and Proof Checker for First-order Logic*. In Matt Kaufmann, Panagiotis Manolios & J. Strother Moore, editors: *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, doi:10.1007/978-1-4757-3188-0_16.

[18] Martin Odersky, Lex Spoon & Bill Venners (2010): *Programming in Scala: A Comprehensive Step-by-step Guide*, 2nd edition. Artima, Inc.

[19] Bruno Woltzenlogel Paleo (2007): *An Approximate Gazetteer for GATE based on Levenshtein Edit Distance*. In: *Twelfth ESSLLI (European Summer School in Logic, Language and Information) Student Section*, pp. 197–208.

[20] Bruno Woltzenlogel Paleo (2007): *Herbrand Sequent Extraction*. Master's thesis, Vienna University of Technology.

[21] Alexandre Riazanov & Andrei Voronkov (2002): *The design and implementation of VAMPIRE*. *AI Commun.* 15(2,3), pp. 91–110.

[22] Jörg Siekmann, Stephan Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet & Volker Sorge (1999): *LOUI: Lovely OMEGA User Interface*. In: *Formal Aspects of Computing*, pp. 326–342.

[23] Geoff Sutcliffe (2009): *The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0*. *Journal of Automated Reasoning* 43(4), pp. 337–362, doi:10.1007/s10817-009-9143-8.

[24] Steven Trac, Yury Puzis & Geoff Sutcliffe (2006): *An Interactive Derivation Viewer*. In: *Proceedings of the 7th Workshop on Workshop on User Interfaces for Theorem Provers, 3rd International Joint Conference on Automated Reasoning, volume 174 of Electronic Notes in Theoretical Computer Science*, pp. 109–123.