A Theorem Prover for Scientific and Educational Purposes

Mario Frank* Christoph Kreitz
University of Potsdam, Institute for Computer Science, Potsdam, Germany
{mafrank,kreitz}@uni-potsdam.de

We present a prototype of an integrated reasoning environment for educational purposes. The presented tool is a fragment of a proof assistant and automated theorem prover. We describe the existing and planned functionality of the theorem prover and especially the functionality of the educational fragment. This currently supports working with terms of the untyped lambda calculus and addresses both undergraduate students and researchers. We show how the tool can be used to support the students' understanding of functional programming and discuss general problems related to the process of building theorem proving software that aims at supporting both research and education.

1 Introduction

Interactive and automated deduction are important concepts in university education when teaching students how to reason about the correctness of software and hardware. But usually, the teaching has to be limited to the theoretical aspects of theorem proving as there are no proof tools suitable for undergraduates. Well-known proof assistants like Coq [1], NuPRL [2], Isabelle [3], Agda [4] and Idris [5] are implemented specifically for scientific purposes and in our experience have a learning curve too steep for beginning students.

Before students can be exposed to advanced deduction topics relevant for reasoning about hardware and software, such as higher-order logics or type theory, they first need to understand the basics of computability, functional programming, and constructive logic. Among others, they need to understand the computational semantics of the (untyped) lambda calculus [6]. While in principle existing proof assistants could be used for this purpose, solving the issue of the steep learning curve would require a lot of effort. Thus there is need for proof assistants specifically suited for the needs of undergraduate students.

In this article we present the educational fragment of a proof assistant that supports the (untyped) lambda calculus and specifically addresses students. After a brief overview of the existing tools and approaches we first discuss the broader context of our tool by presenting the existing and planned functionality of the larger theorem proving environment. After that, we review the essential basics of the untyped lambda calculus and show the various features of our tool and how they can be used. Finally, we discuss the current state and limitation of our tool as well as possible future extensions.

2 Related Work

There are already some efforts to create educational tools for teaching the basics of deduction. Peter Sestoft [7] created a tool that can demonstrate different reduction strategies for the untyped lambda calculus. It supports multiple strategies and can display all intermediate steps executed by the strategies. Named terms (abbreviations) are also supported, which introduces the possibility of modularisation. But

*http://orcid.org/0000-0001-8888-7475

P. Quaresma and W. Neuper (Eds.): 6th International Workshop
on Theorem proving components for Educational software (ThEdu'17)
EPTCS 267, 2018, pp. 59–69, doi:10.4204/EPTCS.267.4

users do not apply individual reduction steps themselves and they do not have to apply α-conversions explicitly as the system does it automatically for them. So the learning outcome is limited. Furthermore, the tool is not stand-alone since a web server is needed to use it.

The most elaborated lambda simulator is the Penn Lambda Calculator, described in [8]. It supports the typed lambda calculus and is specifically designed as educational environment for learning natural language semantics. It also supports the creation of exercises and a conference mode for e-Learning. Furthermore, it includes a feature to grade solutions of students. The implementation in Java makes it platform-independent. The main difference between the Penn Lambda Calculator and our approach is that our approach is built on top of a theorem prover and aims at getting students used to formal reasoning tools.

The lambda calculus tracer TILC [9] is able to display lambda terms as trees. It can highlight bound and free variables, the most inner redex and also mark subterms of a selected lambda term. Where possible, it can reduce lambda terms to normal form and show all steps of the reduction. But it does not terminate if a non-normalisable term like $(\lambda x.xx)(\lambda x.xx)$ is given as input. Furthermore, the tool seems to be out of maintenance since after 2009 and there is only an executable for Windows. There are also some online tools[1] [2] for lambda term evaluation. While the first does not support named lambda terms, the latter does. Both are capable of evaluating lambda terms automatically but students do not have the chance to perform the reductions or α-conversions themselves. There also is no offline version of these tools. The lambda calculators implemented by Joerg Endrullis[3] and a student of the University of Sidney[4] are Java based tools. The former shows the lambda terms as graphs and allows β-reduction via clicking and term manipulation via drag and drop. But the source code does not contain a main routine, so the tool cannot be started successfully. The latter one is text based and does the β-reduction automatically. Carl Burch implemented multiple tools for educational purposes. One of the tools[5] is a lambda simulator both as online JavaScript version and offline Java version. It supports named terms and reduces the given term to normal form automatically, if possible. Another non-graphical implementation[6] of a lambda simulator was written in Haskell. The typed terms are evaluated automatically, too.

3 The Complete Framework for Theorem Proving

As already pointed out in the introduction, our educational tool for the lambda calculus is part of a larger project that aims at supporting both interactive and automated theorem proving. Current interactive theorem provers like Isabelle, Coq, NuPRL, Agda and Idris are successfully used in both scientific and industrial contexts. Although some interactive theorem provers contain subsystems to delegate proof obligations to automated theorem provers like Vampire[10], E[11], iProver[12] and leanCoP[13], those subsystems have various problems. Sledgehammer[14] for example is used for many years as interface between Isabelle and theorem provers like Vampire. But the interfaces to the automated theorem provers are occasionally unstable and the results from the automated theorem provers are in some cases not usable. The main problem here is a missing exchange format that is also enforced. While a de facto standard exists with the TPTP (Thousands of Problems for Theorem Provers)[15] language that also contains the solution language TSTP (Thousands of Solutions from Theorem Provers), this standards

[1] https://people.eecs.berkeley.edu/~gongliang13/lambda/
[2] https://www.easycalculation.com/analytical/lambda-calculus.php
[3] http://joerg.endrullis.de/lambdaCalculator.html
[4] https://github.com/scyptnex/lambda-calculator
[5] http://www.cburch.com/proj/lambda/
[6] https://github.com/sgillespie/lambda-calculus

do not require the automated theorem provers to return complete proofs. In many cases, the automated theorem provers, especially the ones based on resolution or equational reasoning, do not store all proof steps during proof search, which makes it hard or even impossible to communicate all details. For example, the use of the axioms of equalities, i.e. symmetry, reflexivity and transitivity, is subsumed by the inference that the theory of equalities was used. But this information may not be helpful for interactive theorem provers as the proof cannot be reconstructed easily[14]. In some cases this even leads to loss of termination during reconstruction like in the case of the subsystem Metis[16] of Isabelle. Another problem is the variable replacement by skolemisation that is not communicated. And even if an automated theorem prover communicates a complete proof, that proof may not be useful for interactive theorem provers if the proof is not valid in the underlying logic of these provers.

The theorem prover presented here explicitly aims at enforcing complete proofs. Also, it is intended to provide a framework for prototyping automated proof calculi while offering standard decision procedures like unification (e.g. [17, 18] and standard methods for optimisation like term indexing[19]). In a broader sense, the framework can be seen as a logical framework, but in detail it is more than just that. While logical frameworks usually provide the most atomic layer of logic, i.e. the standard logical connectives (conjunction, disjunction, negation, and implication) together with a calculus like the sequent calculus[20], our framework enriches the standard with multiple layers for specialisation to the proof domain. In our framework, connectives like equivalence, NAND and NOR are not only defined in terms of the basic ones but can also be used as atomic connectives without a need for unfolding their definition. This saves time during proof search and still preserves the possibility of unfolding for the complete proof.

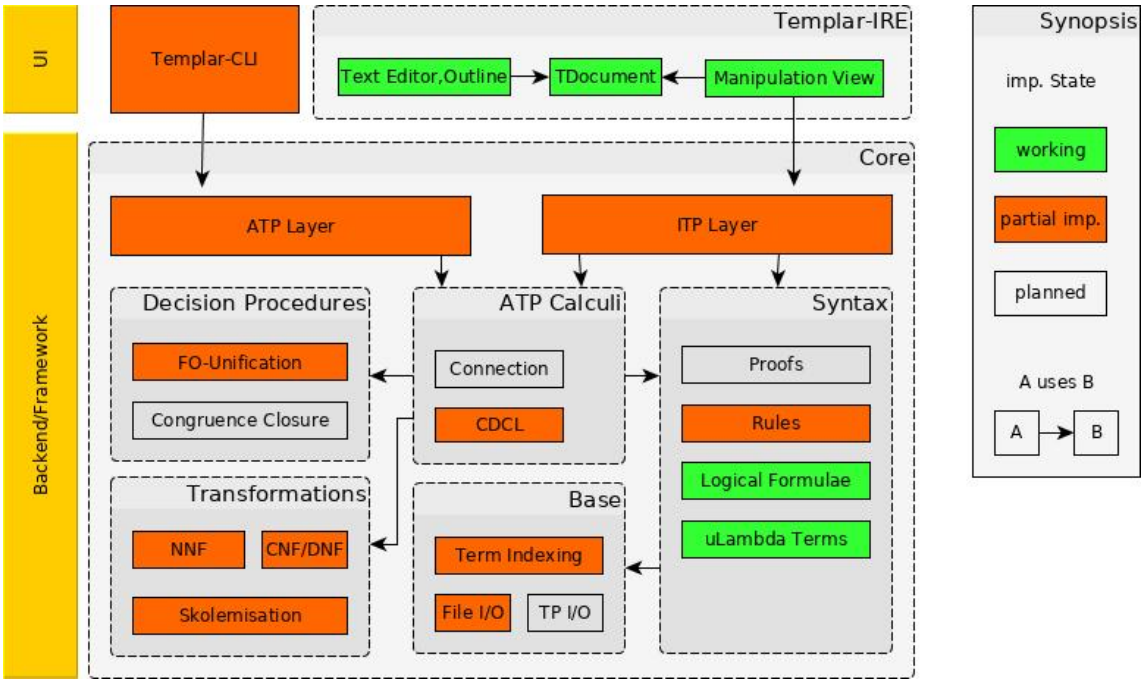


Figure 1: Architecture of the complete theorem proving environment

Those specialisations will enable our framework to select optimised calculi and decision procedures for the problem or domain of interest. To allow specialisation while preserving completeness of information, a special architecture is needed. A draft architecture is presented in figure 1. The architecture of the

system consists of the framework and the UI layer. The framework contains a set of decision procedures, logical transformation procedures, syntax definitions for computational terms, logical formulae along with inference rule and proof schemes. Also, it has a base component with common utility methods for term indexing and file I/O and also already existing prototype implementations of ATP calculi. There are two interfaces that can be accessed externally, i.e. the interactive theorem proving (ITP) and the automated theorem proving (ATP) layer. Both layers have direct access to the ATP calculi component to be able providing its functionality. Moreover, the ITP layer also has direct access to the syntax layer in order to provide it's functionality while the ATP layer gains access to the syntax component via the ATP calculi component. The main reason for the distinction in the syntax component access is that the ITP layer must be able to handle all syntactical definitions directly as it communicates directly with the (graphic) user interface and is not accessed from external systems. The ATP layer on the other hand is planned to being accessible via foreign function interfaces. The communication between the ATP component and external systems is planned to be handled by the base component, especially the theorem prover I/O module.

The most complex part is the syntax component as it needs to enable specialisations. This component contains the untyped lambda syntax and rule definitions that are currently used especially for the educational purpose. Extending these definitions to type theoretical structures, such as the typed lambda calculus would make a communication with ITP systems possible. Moreover, there are implementations for logical connectives that shall be used for automated theorem proving. Between those two definitions, a Curry-Howard-Style transformation is planned, which would make the results from the ATP system reusable for interactive theorem proving.

We plan to provide access to our framework by strictly defined interfaces, like foreign function interfaces. The main advantage of using foreign function interfaces is that the communicated data do not need to be cached in files or in streams but directly forward the syntactical structures to the target system if the target system itself has some reasonably good interface. But as the interfaces are not necessarily present, we focus on systems, where a foreign function interface is easily establishable.

The candidates for interfacing are selected by specific criteria. First of all, there must be a reasonable way to communicate with the theorem provers without passing queries and results as strings via the shell. So, either foreign function interfaces between the theorem, i.e. the programming languages must exist, or the theorem provers must have some structured network based communication possibility, like webservices. This already makes NuPRL and Coq good candidates as the former has network based interfaces and the latter one has foreign function interfaces between OCaml, it's implementation language, and C/C++. Moreover, those candidates are favoured above Isabelle as Isabelle already has ATP connectivity, although it is text based. For ATP systems, the criterion would make E and Vampire good candidates as they are implemented in C and C++ respectively which would make a communication even more easy. OCaml based candidates would also be iProver and Leo[21]. As already the architecture shows, many parts of the system are currently in planning phase or only exist as prototypes. Thus, we focus on the currently usable parts, i.e. the lambda calculus fragment and the IDE.

4 Supported Syntax

The syntax supported by our tool is an extension of the syntax of the untyped lambda calculus.

Definition 1 (Lambda Terms)**.** *The inductive definition of lambda terms is as follows:*

- *A variable* $\boldsymbol{v}$ *is a term.*
- *If* $\boldsymbol{s}$ *and* $\boldsymbol{t}$ *are terms, the application* $\boldsymbol{s\ t}$ *is a term.*

- *If* $\boldsymbol{x}$ *is a variable and* $\boldsymbol{t}$ *a term, then the abstraction* $\lambda x.t$ *is a term.*
- *If* $\boldsymbol{t}$ *is a term, then* (t) *is a term.*

Applications are left associative and abstractions bind as far as possible. α-conversion substitutes a bound variable with a variable with a new name, i.e. $\lambda x.t \xrightarrow{\alpha} \lambda x'.t[x/x']$ and β-reduction reduces an application of the form $(\lambda x.t)s$ to $t[x/s]$. In order to leverage the understanding of modularisation, we support named terms, i.e. abbreviations for complex lambda terms. Thus, we add a rule $t \xrightarrow{\equiv} def_t$ where t must be a named term and def_t is the definition of the named term.

We also extend the syntax by support for multi-bindings, which, for instance, allows to use $\lambda x,y,z$ as abbreviation for $\lambda x.\lambda y.\lambda z$. Both the multi-binding delimiter "," and the normal binding delimiter "." are configurable. For instance, it is possible to define a whitespace as multi-binding delimiter. This way the syntax can be made more conformant to the syntax of other proof assistants like Coq where the whitespace delimits binding parameters. Moreover, the grammar can support multiple variants to write a λ, e.g. "\" and "\lambda", which is also configurable. The tool automatically transforms both variants into the λ symbol. There is also automatical renaming of the rules "\alpha", "\beta" and "\equiv" to α, β and $\equiv$ respectively.

It is possible to define both unnamed and named terms where the former have the syntax $\{Term(\rightarrow(\alpha|\beta|\equiv)Term)*\}$ and the latter extend the unnamed term syntax with a leading $Name :=$ where the name must begin with an upper case symbol. An example for a named term definition would be $True := \{\lambda x.\lambda y.x\}$. So, named and unnamed lambda terms are represented as list of terms, delimited by the rule that was applied in order to generate them.

5 Different Aspects of the Tool

The graphical user interface consists of three parts as shown in figure 2. The left sidebar, i.e. the outline, shows the names or ids of successfully parsed terms. The center contains the source code editor where the lambda terms can be written. The right window, i.e. the manipulation view, shows the currently selected lambda term and its derived terms in a more interactive variant.

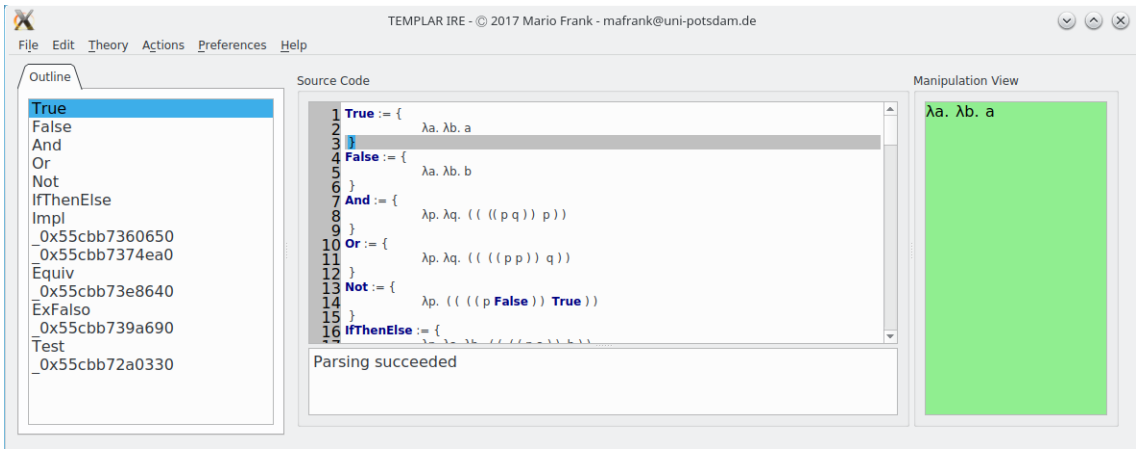


Figure 2: The user interface

Selecting the name or id of a term in the outline sets the lambda term as content of the manipulation

view and signals the code editor to move the cursor to the end of the source code of the term. We will focus on the functionality and use of the source code editor and the manipulation view.

5.1 The Source Code Editor

The source code editor is the main part of the integrated development resoning environment and supports usual the functionality common to most integrated development environments. For example, it supports syntax highlighting, i.e. showing the closing parenthesis for an opening parenthesis on the current cursor position and highlighting the used named terms in blue if they are defined and in red if they are not.

Also, it is able to show the definition and arity of the named term as tool tip when hovering over the name (see figure 3). Moreover, the source code editor supports dynamic code completion for named terms and applicable rules (α-conversion, β-reduction and named term expansion $\equiv$). If a new named term is defined, it is automatically supported by the code completion. Writing in the text editor is more or less like in any other text editor except for the automatic rewriting of special keywords. For example, when the user writes "->\beta \lambda x. \lambda y. f x y", this is automatically rewritten to ->β $\lambda x.\lambda y.fxy$ while the text is written. But this rewriting is only used for the presentation in the editor. The file itself contains the written text in plain format, i.e. \lambda. Although the set of rewriting rules is currently fixed, it is in principle possible to extend it. Enabling users to extend the rules is a part of the future work. The source code editor does not yet support checking the correctness of β-reduction, α-conversion and $\equiv$-expansion that is written. This is currently only supported by the manipulation view.

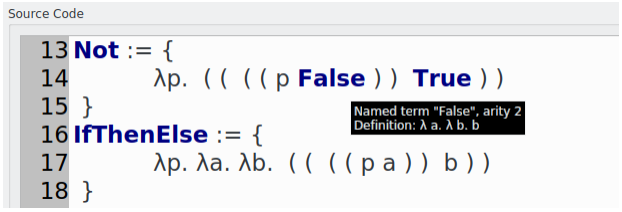


Figure 3: Tooltip with detailed information about the named term.

Below the source code editor, the parser output is shown. On error, the line and column of the erroneous part of the source code is shown.

5.2 The Manipulation View

The manipulation view shows the selected lambda term and its derived terms and supports both informational actions and rule applications on the terms.

5.2.1 Informational Actions

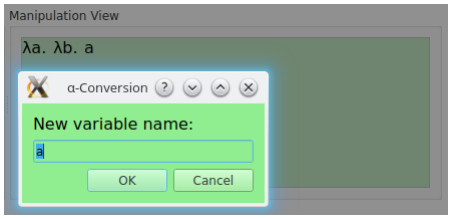
Informational actions are unlocked by holding the control button pressed. The most relevant informational actions are the syntax highlighting. Here, clicking on a parenthesis highlights the corresponding one, and clicking on the name of a variable bound by an abstraction (λx) highlights all occurrences of this variable in the subterm. Also, clicking on a bound variable somewhere in the term highlights all occurrences of it and the location where it is bound, i.e. the abstraction.

Hovering over a named term reference shows its original definition of the named term together with its arity as it is done in the source code editor. Here, holding the control button pressed is not necessary.

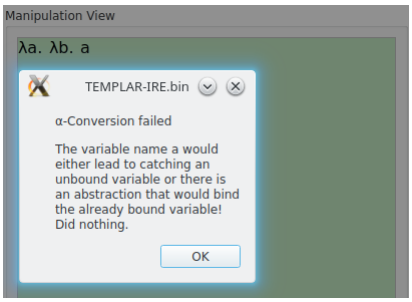
5.2.2 Rule Application

The rule applications are the most important part of the manipulation view for the students. Rules can be applied only on the last term shown in the manipulation view.

5.2.2.1 α-conversion The α-conversion is triggered by double-clicking either on the variable name of an abstraction or on the λ-symbol itself.



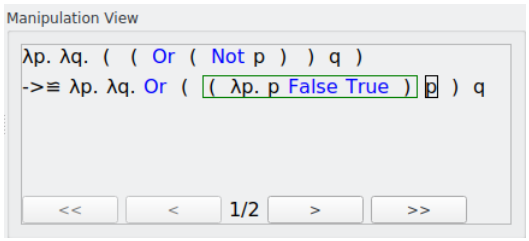
(a) α-conversion input



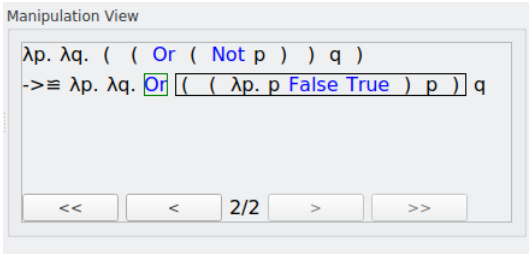
(b) α-conversion failure

Figure 4: α-conversion

This action opens opens an input field (see figure 4a where the new name can be given for the variable. If the new name does not bind a previously free variable or catches another bound variable, α-conversion is applied. Both the manipulation view and the source code are updated. Otherwise, a warning message is shown (see figure 4b). The input field also has to make sure that the variable naming conventions are preserved, i.e. no upper case variable names are allowed. A better way of applying the α-conversion whould be to make the double clicking on the variable switching to edit mode and confirming the change with the return button. This is subject for improvements.



(a) β-reduction choice 1

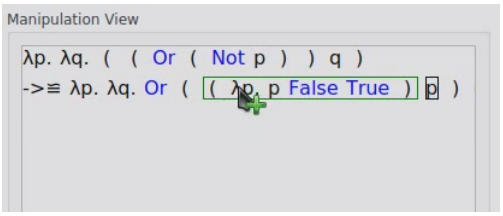


(b) β-reduction choice 2

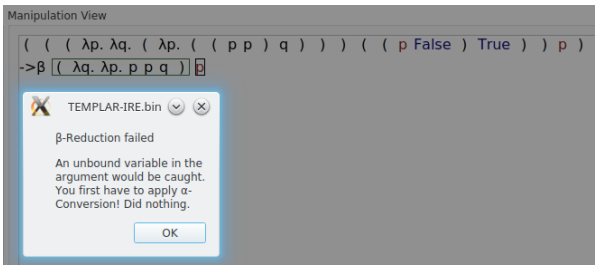
Figure 5: β-reduction choices

5.2.2.2 β-reduction If the last term is not in β-normal form, the β-reduction can be applied to it. The manipulation view shows how many possible β-reductions are possible in a navigation bar at the bottom. A β-reduction option is always shown by highlighting the argument in a black box and the function where the argument can be inserted in a green box (see 5a). If there are multiple possible reductions, the focus on the application to be reduced can be changed with the shortcuts CTRL+P for previous and CTRL+N for next or with the navigation buttons on the bottoms of the manipulation view (see figure 5a and 5b).

Currently, applying the β-reduction can be performed either by the keyboard shortcut CTRL+A or dragging the function of the application and dropping it on the function (see figure 6a).



(a) Dropping the argument on the function



(b) Failing β-reduction

Figure 6: β-reduction application

The β-reduction will fail with a warning if it would lead to free variables becoming bound. Figure 6b for example shows that the beta reduction fails as the free variable p (red) would become bound. This way, the students' awareness for binding scopes shall be increased. If the term is in normal form, i.e. irreducible, the background of the manipulation view is coloured green (see 7).

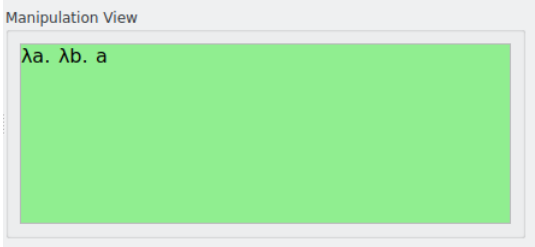


Figure 7: λ-term in normal form

5.2.2.3 $\equiv$-expansion The $\equiv$-expansion can be applied on all named term references that have a valid definition. It is triggered by double-clicking on the named term reference. Also, the $\equiv$-expansion is applied automatically to the function of a reducible application if the β-reduction is triggered and the function is a named term reference.

The manipulation view also supports the undo and redo operations, although there are still improvements needed in this functionality as the undo/redo operations need to be synchronised with the editor view.

6 Challenges

Concepting a theorem prover as educational tool leads to some problems which do not have to be considered for pure theorem provers. One of the main problems evolving from didactic considerations is the necessity of α-conversion. In ordinary theorem proving, no α-conversion is needed if a De Bruijn index like structure is used, as the variable names are not relevant at all.

But when teaching theory of programming, naming clashes are an important aspect students need to understand. Thus, the α-conversion should be supported from the didactic point of view. Moreover, even

if the framework itself does not induce or have to consider naming clashes, the graphic user interface still may do this. In the context of our tool, the lambda terms are stored internally with a De Bruijn like representation. Thus, the β-conversion and $\equiv$-expansion can be completely agnostic about variable names. But for readibility reasons, the variables become relevant as they have to be shown in the text editor panel and are also parsed from the raw text. A solution would be to rename variables on clashes on the fly but then again the didactical gain for students would be smaller as they can just expect the framework to correct their mistakes. Thus, the user interface needs to address α-conversion and the framework itself must handle naming collisions.

Another challenge in the conception of the tool was the usability, especially the speed of reaction. As waiting for the result of a β-conversion is not what students expect, the rendering speed of the newly generated lambda term needed to be improved. The main problem of complexity was that on applying a rule to the current lambda term, the complete term with all reduction, conversion and expansion steps was rendered. With big terms, this could take several seconds. Thus, the manipulation view was adopted to only render the newly created term. But as both the text editor and the manipulation view support undo/redo, and must be held consistent, they had to share some information. Thus, the lambda terms were embedded in a document structure that is used both by the editor and manipulation view.

7 Supported Platforms and Future Work

Our tool is implemented in C++ and is available as binary distribution for Linux. An AppImage for Linux is also available. This bundle contains all dependencies to work with the tool also on older Linux distributions. A binary distribution for Microsoft Windows is currently under development and could already be tested successfully in preliminary versions. All versions of the tool are portable, i.e. no installation is needed. A distribution as online tool is under consideration as the emscripten[7] plugin for the C++ compiler supports the translation from C++ to JavaScript. The tool in all its variants is available at `http://www.cs.uni-potsdam.de/~mafrank/` .

There are many possible small improvements. First of all, a stronger decoupling of the graphic user interface from the framework is planned. The framework should just apply the rules without checking for potentially caught variables. The graphic user interface itself should check for those collisions. Also, the application of the alpha conversion should be improved by replacing the interactive renaming window with an inline edit mode. Furthermore, the visualisation of the drag and drop mechanism for β-reduction will be improved. There are still some inconsistencies in the undo/redo functionality of the manipulation view.

Also, extensions of the tool are planned. This includes a graph view for lambda terms and improvements on the manipulation view. Lifting the tool to the typed lambda calculus is also work in progress. Moreover, the current representation of the source code as file will be replaced by a database with plain text export.

8 Conclusion

We presented an educational tool for undergraduate students that has the aim to improve their understanding of functional programming while giving them at least some of the features of state of the art developement environments. Though this tool is work in progress it was successfully used by multiple

[7] `https://github.com/kripken/emscripten`

students in first year undergraduate studies. A first release where application was not yet possible with drag and drop received positive feedback but also some proposals for improvement. Much of the functionality was motivated by this feedback as is also the future work. We hope that this kind of educational tools will enable the students to get used to formal tools like proof assistants more easily.

References

[1] The Coq development team (2004): *The Coq proof assistant reference manual*. LogiCal Project. Available at `http://coq.inria.fr`. Version 8.0.

[2] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki & Scott F. Smith (1986): *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ.

[3] Tobias Nipkow, Markus Wenzel & Lawrence C. Paulson (2002): *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, doi:10.1007/3-540-45949-9.

[4] Ulf Norell (2009): *Dependently Typed Programming in Agda*. In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, ACM, New York, NY, USA, pp. 1–2, doi:10.1145/1481861.1481862.

[5] Edwin Brady (2013): *Idris, a general-purpose dependently typed programming language: Design and implementation*. *Journal of Functional Programming* 23, pp. 552–593, doi:10.1017/S095679681300018X.

[6] Henk P. Barendregt (1984): *The Lambda Calculus Its Syntax and Semantics*, revised edition. 103, North Holland.

[7] Peter Sestoft (2001): *Demonstrating Lambda Calculus Reduction*. *Electr. Notes Theor. Comput. Sci.* 45, pp. 424–432, doi:10.1016/S1571-0661(04)80973-3. `http://www.itu.dk/people/sestoft/lamreduce/`.

[8] Lucas Champollion, Joshua Tauberer & Maribel Romero (2007): *The Penn Lambda Calculator: Pedagogical Software for Natural Language Semantics*. Technical Report, University of Konstanz, Germany, KOPS. Available at `http://kops.ub.uni-konstanz.de/volltexte/2009/9605/`. `http://lambdacalculator.com/`.

[9] David Ruiz & Mateu Villaret (2009): *TILC: The Interactive Lambda-Calculus Tracer*. *Electron. Notes Theor. Comput. Sci.* 248, pp. 173–183, doi:10.1016/j.entcs.2009.07.067.

[10] Alexandre Riazanov & Andrei Voronkov (1999): *Vampire*. In: *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, pp. 292–296, doi:10.1007/3-540-48660-7_26.

[11] Stephan Schulz (2002): *E - a brainiac theorem prover*. *AI Commun.* 15(2,3), pp. 111–126.

[12] Konstantin Korovin (2008): *iProver – An Instantiation-Based Theorem Prover for First-Order Logic (System Description)*. In A. Armando, P. Baumgartner & G. Dowek, editors: *Proceedings of the 4th International Joint Conference on Automated Reasoning, (IJCAR 2008)*, *Lecture Notes in Computer Science* 5195, Springer, pp. 292–298, doi:10.1007/978-3-540-71070-7_24.

[13] Jens Otten (2008): *leanCoP 2.0 and ileanCoP 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic (System Descriptions)*. In: *Proceedings of the 4th international joint conference on Automated Reasoning*, IJCAR '08, Springer-Verlag, Berlin, Heidelberg, pp. 283–291, doi:10.1007/978-3-540-71070-7_23.

[14] Lawrence C. Paulson & Jasmin C. Blanchette (2012): *Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers*. In Geoff Sutcliffe, Stephan Schulz & Eugenia Ternovska, editors: *IWIL 2010. The 8th International Workshop on the Implementation of Logics*, *EPiC Series in Computing* 2, EasyChair, pp. 1–11, doi:10.29007/36dt. Available at `https://easychair.org/publications/paper/wV`.

[15] 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.

[16] Joe Hurd (2003): *First-order proof tactics in higher-order logic theorem provers*. In: Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports, pp. 56–68.

[17] John A. Robinson (1965): *A Machine-Oriented Logic Based on the Resolution Principle*. J. ACM 12(1), pp. 23–41, doi:10.1145/321250.321253.

[18] Alberto Martelli & Ugo Montanari (1982): *An Efficient Unification Algorithm*. ACM Trans. Program. Lang. Syst. 4(2), pp. 258–282, doi:10.1145/357162.357169.

[19] Robert M. Colomb (1991): *Enhancing unification in PROLOG through clause indexing*. The Journal of Logic Programming 10(1), pp. 23–44, doi:10.1016/0743-1066(91)90004-9.

[20] Gerhard Gentzen (1935): *Untersuchungen über das logische Schließen. I*. Mathematische Zeitschrift 39(1), pp. 176–210, doi:10.1007/BF01201353.

[21] Max Wisniewski, Alexander Steen & Christoph Benzmüller (2014): *The Leo-III Project*. In Alexander Bolotov & Manfred Kerber, editors: Joint Automated Reasoning Workshop and Deduktionstreffen, p. 38. Available at `http://christoph-benzmueller.de/papers/W53.pdf`.