A Prolog-based Proof Tool for Type Theory TA_λ and Implicational Intuitionistic-Logic

L. Yohanes Stefanus
University of Indonesia
Depok 16424, Indonesia
yohanes@cs.ui.ac.id
and Ario Santoso*
Technische Universität Dresden
Dresden 01187, Germany
santoso.ario@gmail.com

Abstract

Studies on type theory have brought numerous important contributions to computer science. In this paper we present a GUI-based proof tool that provides assistance in constructing deductions in type theory and validating implicational intuitionistic-logic formulas. As such, this proof tool is a testbed for learning type theory and implicational intuitionistic-logic. This proof tool focuses on an important variant of type theory named TA_λ, especially on its two core algorithms: the principal-type algorithm and the type inhabitant search algorithm. The former algorithm finds a most general type assignable to a given λ-term, while the latter finds inhabitants (closed λ-terms in β-normal form) to which a given type can be assigned. By the Curry–Howard correspondence, the latter algorithm provides provability for implicational formulas in intuitionistic logic. We elaborate on how to implement those two algorithms declaratively in Prolog and the overall GUI-based program architecture. In our implementation, we make some modification to improve the performance of the principal-type algorithm. We have also built a web-based version of the proof tool called λ-Guru.

1 Introduction

In 1902, Russell revealed the inconsistency in naive set theory. This inconsistency challenged the foundation of mathematics at that time. To solve that problem, type theory was introduced [1, 5]. As time goes on, type theory becomes widely used, it becomes a logician's standard tool, especially in proof theory. Since 1970s, the need for robust programming languages has made type theory the center of attention among computer scientists. Many practical programming languages have been developed based on type theory [4, 6].

Type theory has many variants, one of them is called TA_λ (for *type-assignment in λ-calculus*). TA_λ is one of the simplest type-theories. Many complex techniques for analyzing the structures of complex type-theories appear in TA_λ in an easier form and hence easier to understand. Therefore, TA_λ is a good vehicle for learning more complex techniques in type theory. By learning TA_λ, one can obtain a good foundation to study other type-systems [4].

The importance of TA_λ and its relation to type theory in general drive this research project. This work focuses on two most important algorithms in TA_λ, i.e., the principal-type algorithm and the type inhabitant search algorithm. The former algorithm finds a most general type (principal type) assignable to a given λ-term, while the latter finds inhabitants (i.e., closed λ-terms in β-normal form) to which a given type can be assigned. Those two algorithms are implemented using Prolog and exploiting the ability of definite clause grammars (DCG)

*Supported by the European Master's Program in Computational Logic (EMCL)

for processing linguistic expressions. DCG is a Prolog feature for expressing grammar rules [2]. In addition, a graphical user interface (GUI) is developed for helping users try those two algorithms. Therefore, the resulting software can be used as a testbed for learning TA_λ and, by the Curry–Howard correspondence, as a proof assistant for implicational formulas in intuitionistic logic.

2 The Principal-Type Algorithm Implementation

The principal-type algorithm decides whether a λ-term is typable or not. It takes a λ-term as an input. If the given λ-term is typable, this algorithm will produce its principal (most general) type and its TA_λ-deduction tree. If the given λ-term is not typable, this algorithm will give a correct statement that the given λ-term is not typable. Here are some examples of input and output of this algorithm:

1. **Input:** $\boldsymbol{\lambda yz.(yz)z}$
   Output:

$$\dfrac{\dfrac{\dfrac{\boldsymbol{y : a{\to}a{\to}b \mapsto y : a{\to}a{\to}b} \quad \boldsymbol{z : a \mapsto z : a}}{\boldsymbol{y : a{\to}a{\to}b, z : a \mapsto yz : a{\to}b}}\,(\to\!\boldsymbol{E}) \quad \boldsymbol{z : a \mapsto z : a}}{\boldsymbol{y : a{\to}a{\to}b, z : a \mapsto (yz)z : b}}\,(\to\!\boldsymbol{E})}{\boldsymbol{y : a{\to}a{\to}b \mapsto \lambda z.(yz)z : a{\to}b}}\,(\to\!\boldsymbol{I})}{\boldsymbol{\mapsto \lambda yz.(yz)z : (a{\to}a{\to}b){\to}a{\to}b}}\,(\to\!\boldsymbol{I})$$

2. **Input:** $\boldsymbol{\lambda xz.(xz)x}$
   Output: The term is not typable

We do not explain the details of the algorithm in this paper due to space limitation. Please check the details in [3, 4].

In this implementation, we use a definite clause grammar (DCG) in Prolog for parsing the input as a λ-term. While parsing the λ-term, the parser constructs a tree that represents the given λ-term. If the given λ-term is valid, the program will apply the principal-type algorithm to decide whether the given λ-term is typable or not and produce a deduction for the typable case. Generally the process is described by the flowchart in Figure 1.

The principal-type algorithm consists of four cases: (1) when the λ-term is a term variable, (2) when the λ-term is an application, (3) when the λ-term is an abstraction $\boldsymbol{\lambda x.P}$ with $\boldsymbol{x \in FreeVariable(P)}$, and (4) when the $\boldsymbol{\lambda}$-term is an abstraction $\boldsymbol{\lambda x.P}$ with $\boldsymbol{x \notin FreeVariable(P)}$. The four cases are distinguished in the predicate `pta/5` which implements the algorithm by recursively analyzing the structure of the tree which represents the input $\boldsymbol{\lambda}$-term. The five arguments of the predicate are as follows:

1. The $\boldsymbol{\lambda}$-term $\boldsymbol{M}$ which we want to assign a type.
2. The $\mathrm{TA}_{\boldsymbol{\lambda}}$-deduction for the $\boldsymbol{\lambda}$-term $\boldsymbol{M}$.
3. The result of the algorithm, i.e. the statement which says that the $\boldsymbol{\lambda}$-term $\boldsymbol{M}$ is typable or not.

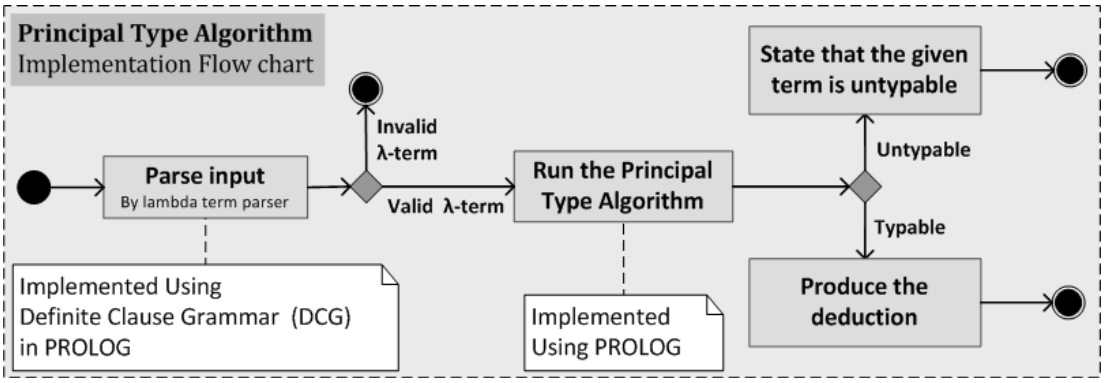

Figure 1: Flowchart for the Implementation of the Principal-Type Algorithm

4. The list of variables used during deduction.
5. The tree which represents the deduction.

The case distinction happens in the first argument. For example, we represent an abstraction $\boldsymbol{\lambda x.M}$ as `abstraction(termVar(X),M)`. Then if the $\boldsymbol{\lambda}$-term in the current recursion has that form, then we apply the algorithm for an abstraction.

Improvement on the Principal-Type Algorithm

In our implementation, we make some modification to improve the performance of the principal-type algorithm given in [4]. The improvement is obtained by combining case II and case III. This modification eliminates the need for checking whether an abstraction variable is free in the abstraction body or not.

3 The Type Inhabitant Search Algorithm Implementation

The type inhabitant search algorithm is a core procedure for answering the question "given a type $\boldsymbol{\tau}$, how many *inhabitants* (closed $\boldsymbol{\lambda}$-terms in $\boldsymbol{\beta}$-normal form) can receive $\boldsymbol{\tau}$ in $\mathrm{TA}_{\boldsymbol{\lambda}}$?" [8, 4]. This algorithm can be used as a tester whether the number of closed $\boldsymbol{\lambda}$-terms that can receive a certain type $\boldsymbol{\tau}$ is zero or not. As a consequence, by the Curry–Howard correspondence, it can be used to test whether a certain implicational formula in intuitionistic logic is provable or not. The algorithm takes a type as input and searches for closed $\boldsymbol{\lambda}$-terms in $\boldsymbol{\beta}$-normal form that can receive the given type in $\mathrm{TA}_{\boldsymbol{\lambda}}$ through several steps. For example, given

$$\boldsymbol{\tau \equiv (((c \to d) \to c \to a) \to b) \to (d \to a) \to b}$$

as input, the algorithm will produce the following sets:

$$\mathscr{A}(\boldsymbol{\tau}, \mathbf{0}) = \{\boldsymbol{V^{\tau}}\},$$

$$\mathscr{A}(\boldsymbol{\tau}, \mathbf{1}) = \{\boldsymbol{\lambda x_1^{((c\to d)\to c\to a)\to b} x_2^{d\to a}.x_1^{((c\to d)\to c\to a)\to b} V_1^{(c\to d)\to c\to a}}\},$$

$$\mathscr{A}(\boldsymbol{\tau}, \mathbf{2}) = \{\boldsymbol{\lambda x_1^{((c\to d)\to c\to a)\to b} x_2^{d\to a}.x_1^{((c\to d)\to c\to a)\to b} (\lambda x_3^{c\to d} x_4^{c}.x_2^{d\to a} V_2^{d})}\},$$

$$\mathscr{A}(\tau, 3) = \{\lambda x_1^{((c\to d)\to c\to a)\to b} x_2^{d\to a}.x_1^{((c\to d)\to c\to a)\to b}(\lambda x_3^{c\to d} x_4^c.x_2^{d\to a}(x_3^{c\to d} V_3^c))\},$$

$$\mathscr{A}(\tau, 4) = \{\lambda x_1^{((c\to d)\to c\to a)\to b} x_2^{d\to a}.x_1^{((c\to d)\to c\to a)\to b}(\lambda x_3^{c\to d} x_4^c.x_2^{d\to a}(x_3^{c\to d} x_4^c))\}$$

as it searches for closed $\boldsymbol{\lambda}$-terms in $\boldsymbol{\beta}$-normal form. The $\boldsymbol{\lambda}$-terms are written here in full typed-notation where each term variable is decorated with its type explicitly at the superscript position. The sequence of $\mathscr{A}$-sets shows the approximation from initially unknown $\boldsymbol{\lambda}$-terms (containing meta-variables $\boldsymbol{V}, \boldsymbol{V_1}, \boldsymbol{V_2}$, etc.) toward closed $\boldsymbol{\lambda}$-terms in $\boldsymbol{\beta}$-normal form. In the example above, type $\boldsymbol{\tau}$ has the $\boldsymbol{\lambda}$-term

$$\lambda x_1^{((c\to d)\to c\to a)\to b} x_2^{d\to a}.x_1^{((c\to d)\to c\to a)\to b}(\lambda x_3^{c\to d} x_4^c.x_2^{d\to a}(x_3^{c\to d} x_4^c))$$

as its inhabitant. By the Curry–Howard correspondence, the existence of this inhabitant means that the formula

$$(((c \to d) \to c \to a) \to b) \to (d \to a) \to b$$

is valid in intuitionistic logic.

In our implementation, we use definite clause grammars in Prolog for parsing the input as a type expression. The parser constructs a tree that represents the given type expression. If it is valid, the program will run the type inhabitant search algorithm to search for its inhabitants. In general, the process is described by the flowchart in Figure 2.

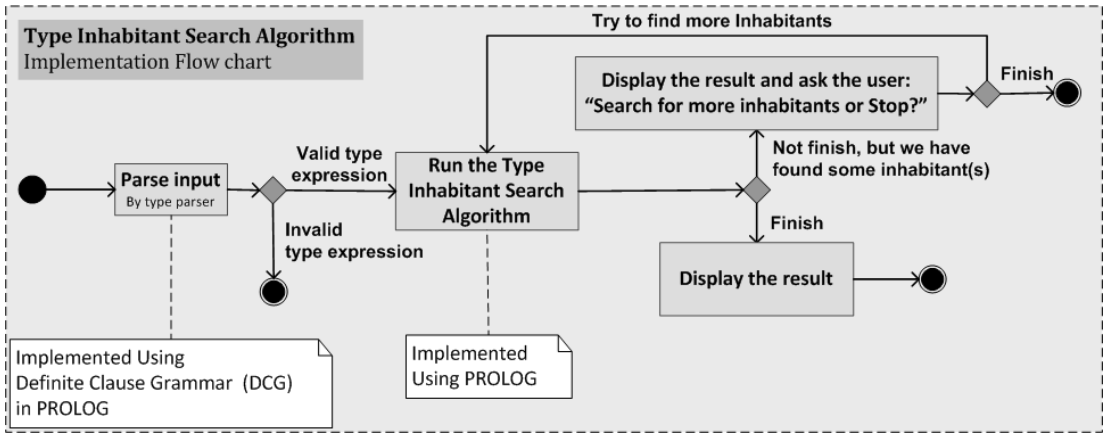

Figure 2: Flowchart for the Implementation of the Type Inhabitant Search Algorithm

The main idea of the type inhabitant search algorithm is to iterate through the approximation of the $\boldsymbol{\lambda}$-term and to produce $\boldsymbol{\mathcal{A}}$-sets with certain depth ($\boldsymbol{\mathcal{A}(\tau, 0)}$, $\boldsymbol{\mathcal{A}(\tau, 1)}$, ..., $\boldsymbol{\mathcal{A}(\tau, n)}$) during the iteration. In this implementation, we model the iteration as recursion which constructs the $\boldsymbol{\mathcal{A}}$-sets. We define a predicate `searchIhbt/3` with the following arguments:

1. an $\boldsymbol{\mathcal{A}}$-set, i.e. $\boldsymbol{\mathcal{A}(\tau, d)}$ for a certain type $\boldsymbol{\tau}$ and depth $\boldsymbol{d}$.
2. the list of previous $\boldsymbol{\mathcal{A}}$-sets, i.e. the list of $\boldsymbol{\mathcal{A}(\tau, t)}$ with $\boldsymbol{t = 0, 1, \ldots, d-1}$.
3. the list of all $\boldsymbol{\mathcal{A}}$-sets that we get during the iteration (searching).

4 User Interface

To help users interact with the implementation of those two algorithms, we provide a user-friendly GUI (graphical user interface). The design of our user interface follows the golden rules in designing user interface [7]. Figure 3 and Figure 4 show an overview of our GUI design.

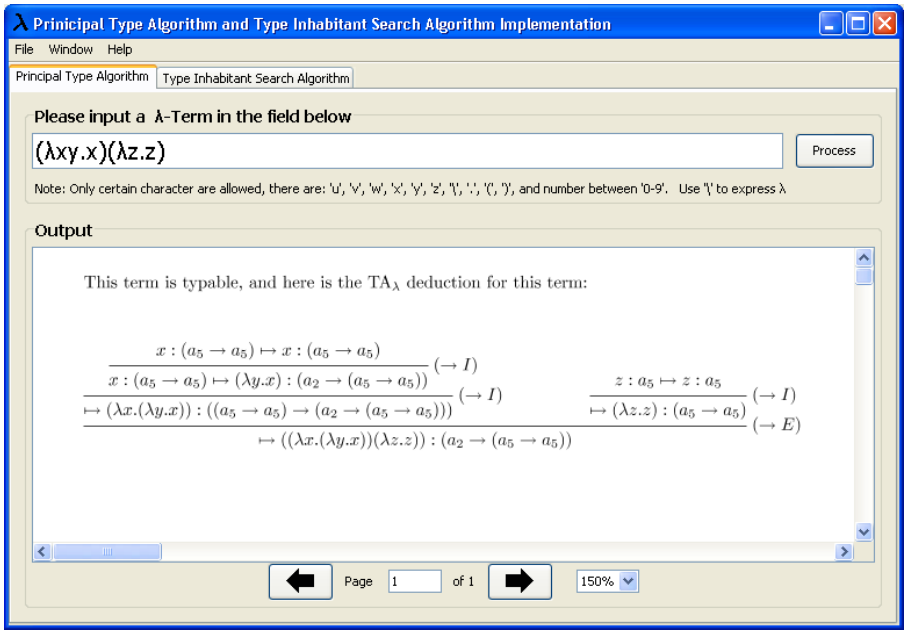

Figure 3: Overview of GUI Design for the Principal-Type Algorithm

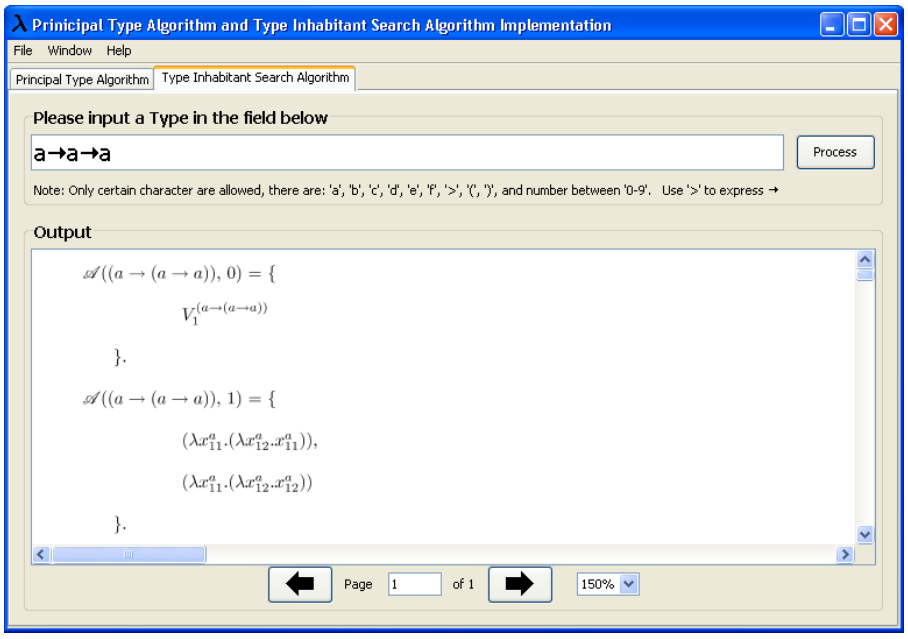

Figure 4: Overview of GUI Design for the Type Inhabitant Search Algorithm

The user interface consists of two main tabs: Principal-Type Algorithm tab and Type Inhabitant Search Algorithm tab. In the Principal-Type Algorithm tab, there is a textbox where user can input a **λ**-term and a panel where the type deduction is shown. In the Type Inhabitant Search Algorithm tab, there is a textbox where user can input a type and a panel where the algorithm result is shown.

This GUI is implemented using Java with Prolog in the back end. We also use LaTeX to help generating tidy output from the algorithms, i.e. deduction trees and steps of type inhabitants searching. In general, the program works as follows. The Java program takes the user input and calls the Prolog programs implementing the algorithms declaratively. The Prolog programs generate the output of the algorithms as LaTeX code. After the algorithms finish execution, the Java program calls LaTeX to compile the algorithm output into a PDF file. Lastly, the Java program reads the generated PDF file and displays it to the user as output.

The flowchart of this architecture is shown in Figure 5 and the explanation is as follows:

1. User writes a **λ**-term into the input textbox.
2. The Java GUI program parses the input **λ**-term using the parser implemented in Prolog.
3. The parser passes the parsed **λ**-term to the program implementing the principal-type algorithm.
4. The Java GUI program gets back the deduction result for the given **λ**-term.
5. The Java GUI program writes the deduction in LaTeX format into a file.
6. The Java GUI program calls the LaTeX program to convert the LaTeX file into a PDF file.

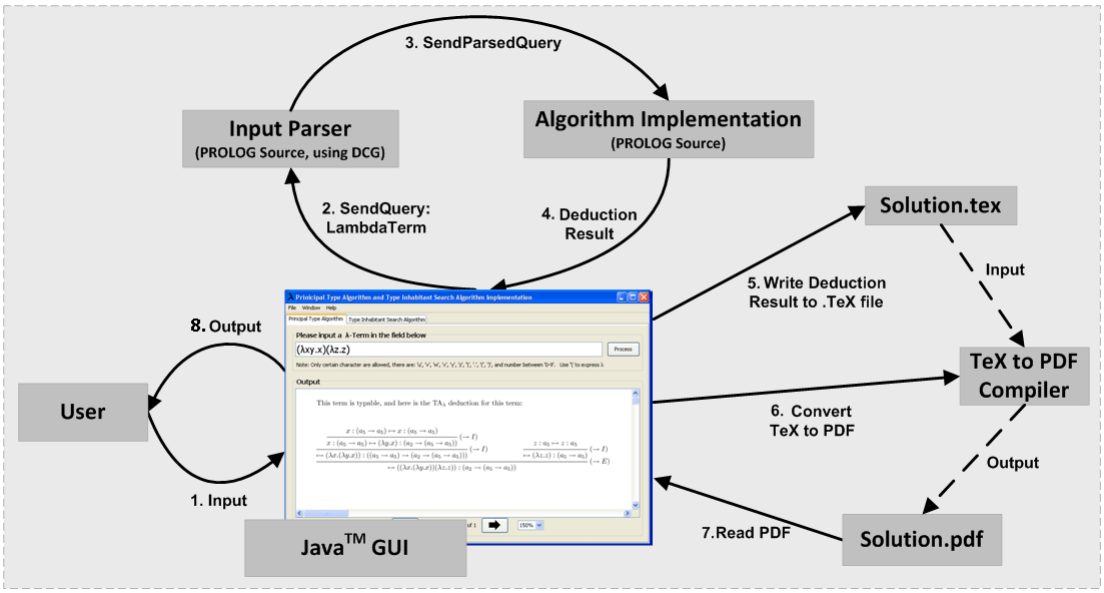

Figure 5: Illustration of the Scenario for the Principal-Type Algorithm Program

7. The Java GUI program reads the PDF file.

8. The Java GUI program displays the solution to the user.

For the program of the type inhabitant search algorithm implementation, the flowchart is similar. The difference is only on the parser and the main Prolog program for computing the final result.

As a package, this GUI-based implementation of the principal-type algorithm and the type inhabitant search algorithm can play the role of a proof tool that provides assistance in constructing deductions in type theory and establishing the validity of implicational intuitionistic-logic formulas. The intuitive interface helps people study and understand the two algorithms and the related theories.

A web-based version of the software, called **λ**-Guru, has also been built. It can be accessed at **http://fmse.cs.ui.ac.id/LambdaGuru**.

Generally, to construct the **λ**-Guru, we convert the Java GUI program, which is developed under the Java Swing library, into a JApplet. We also create a PHP program which accepts HTTP POST requests from the JApplet. This PHP program then calls the Prolog program which parses the input and applies the principal-type algorithm or the type inhabitant search algorithm. After the PHP program gets the result back from the Prolog program, it then forwards the algorithm result to the JApplet program. Finally, the JApplet program displays the result to the user. The flowchart of this process is shown in Figure 6.

5 Conclusion and Future Work

We have presented a Prolog-based implementation of a proof tool that provides assistance in constructing deductions in type theory and establishing the validity of implicational intuitionistic-logic formulas. With this functionality, this proof tool is useful for learning type theory. This

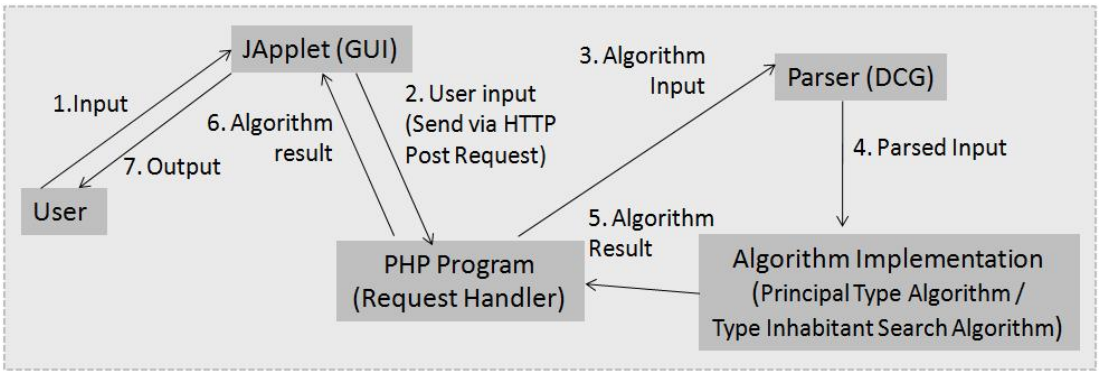

Figure 6: $\boldsymbol{\lambda}$-Guru Web-based Architecture

proof tool focuses on an important variant of type theory named $\mathrm{TA}_{\boldsymbol{\lambda}}$, especially on its two core algorithms: the principal-type algorithm and the type inhabitant search algorithm. We have elaborated on how to implement those two algorithms declaratively in Prolog and the overall GUI-based program architecture. We have also described the graphical user interface that can help users interact with the program. Furthermore, we have built a web-based version of the proof tool called $\boldsymbol{\lambda}$-Guru.

The next step of this work is to implement declaratively the converse principal-type algorithm. It is shown in [4] that if a type $\boldsymbol{\tau}$ is assignable to a closed $\boldsymbol{\lambda}$-term $\boldsymbol{M}$ but is not the principal type of $\boldsymbol{M}$, then it is the principal type of another closed $\boldsymbol{\lambda}$-term $\boldsymbol{M^*}$. The task of the converse principal-type algorithm is to construct $\boldsymbol{M^*}$ when $\boldsymbol{\tau}$ and $\boldsymbol{M}$ are given.

Acknowledgments

The authors would like to thank the anonymous reviewers of this paper for their helpful comments and suggestions.

References

[1] A. N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, Cambridge, revised edition, 1925-1927.

[2] L. Sterling and E. Shapiro. *The Art of PROLOG.* MIT Press, Massachusetts, second edition, 1994.

[3] J. R. Hindley. *The Principal Type-Scheme of an Object in Combinatory Logic.* Trans. American Math. Soc., 1969.

[4] J. R. Hindley. *Basic Simple Type Theory.* Cambridge University Press, Cambridge, first edition, 1997.

[5] C. Munoz. Type theory and its applications to computer science. *Quarterly News Letter of Institute for Computer Application in Science and Engineering (ICASE), Vol 8, No 4*, 2007.

[6] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, Massachusetts, 2002.

[7] B. Shneiderman and C. Plaisant. *Designing The User Interface.* Pearson Education, fourth edition, 2005.

[8] B. Yelles. Type-assignment in the lambda-calculus; syntax and semantic. Technical report, Mathemathics Dept. University of Wales Swansea, Swansea SA2 8PP, UK, 1979.