To appear in PROLE 2008

TILC: The Interactive Lambda-Calculus Tracer [1]

David Ruiz and Mateu Villaret [2,3]

Informàtica i Matemàtica Aplicada
Universitat de Girona
Girona, Spain

Abstract

This paper introduces TILC: the interactive lambda-calculus tracer. TILC aims to be a friendly user graphical application that helps teaching/studying the main basic concepts of pure untyped lambda-calculus. This is achieved by allowing users to graphically interact with a sort of parse-tree of the lambda-terms and automatically reproducing these interactions in the lambda-term. This graphical interaction encourages students to practice with lambda-terms easing the learning of the syntax and of the operational semantics of lambda-calculus.
TILC has been built using `HASKELL`, `wxHaskell` and `Happy`, it can be freely downloaded from `http://ima.udg.edu/~villaret/tilc`.

Keywords: lambda-calculus, tracing tool, teaching/learning

1 Introduction

Teaching (studying) lambda-calculus for the first time to undergraduate students, not used to this kind of formalisms, has some difficulties. Take the grammar of lambda-calculus with just names of variables, lambda-abstractions and the curried application, mix it with the corresponding lot of parentheses, finally shake it with the notational convention, and that's it, you get the more appropriate cocktail to produce in the students the feeling of "*Oh my god!!! what a hard day...*".

In the Universitat de Girona, pure untyped lambda-calculus is taught in a fourth year mandatory *programming paradigms* course in the computer science curriculum, as the archetypal minimal functional programming language and therefore the computational model for this paradigm. As in many other courses where lambda-calculus is taught, we follow this process: presentation of syntax, definitions of bound and free variable occurrences, definition of capture-avoiding substitution, definition of the operational semantics of lambda-calculus with α, β and η-transformations, and in the end, normalization strategies and corresponding main theorems. Then we try to convince the students that this formalism is, in fact, the computational formalism that underlies functional programming. Hence, we define lambda-terms for Church numerals, boolean, conditional, tuples, lists and finally, the Y and the T fixed-point combinators. These terms allow us to see that any "recursive" function, like the factorial function, can be encoded within this formalism.

Nevertheless, when one shows these encodings, students feel as if there were a kind of "*black magic*"...To *see* why these encodings work, students have to practice.

[1] The work has been partially founded by Escola Politècnica Superior of Universitat de Girona

[2] Email: u1046809@correu.udg.edu

[3] Email: villaret@ima.udg.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

TILC is a graphical application that mainly consists of an area where lambda-terms are textually introduced, and a panel where the parse-tree of the term is represented and can be manipulated. The effects of these manipulations are graphically and textually reproduced: sub-term identification, bound-variables and corresponding lambda-binders highlighting, β-reduction, ... Moreover, the application allows the user to define alias for lambda-terms via `let`-expressions and these can be naturally used in subsequent lambda-terms. Using a tool dealing with these features in a friendly and graphical manner encourages students experimentation, and therefore, students comprehension of lambda-calculus.

Several works exist [4] dealing with the practice of lambda-calculus but none of them fits precisely with our educational purpose. In [3] *lambreduce* is described. It is a web-based tool written using Moscow ML which allows users to write pure untyped lambda-terms and ask for different normal forms using distinct strategies. Nevertheless it works textually and does not deal with parse-tree representation. In [2] we find the graphical application *The Penn Lambda Calculator*. It is focussed on teaching and practicing with lambda-calculus but applied to natural language semantics. Another graphical web-based tool is the *lambda-animator* [5]. This application goes one step further than ours because it deals with more advanced features as: graph reduction with sharings, laziness, δ-reductions, etc. Nevertheless, it does not assist basic syntax comprehension like subterm or binding, nor direct manipulation of β-redexes, etc. Some of the features of this application could be a perfect continuation to ours. In fact, the use of δ-reductions, sharing and so on, links with many other tools that deal with visualizations for the functional programming paradigm as: *CIDER*, *WinHIPE*, *TERSE*, ... The survey in [6] provides a brief description of these and other tools that also serve for tracing functional programs. These could be the natural subsequent tools in a functional programming course.

TILC has been developed by David Ruiz as a diploma thesis and was proposed and supervised by Mateu Villaret. It has been fully developed using `HASKELL`, `wxHaskell` for the graphical interface and `Happy` to build the parsers. Its home page is `http://ima.udg.edu/~villaret/tilc` from where documentation and binary files for Windows can be freely downloaded. Binaries for other platforms and source code are under preparation.

2 Recalling Pure Untyped Lambda-Calculus for TILC

Pure untyped lambda-calculus used in our framework rely on [1] as the standard reference. A *lambda-term* is a *variable* x from some denumerable set of variables, a *lambda-abstraction* $(\lambda x \,.\, \Lambda)$ which binds x in Λ or an *application* $(\Lambda_1\ \Lambda_2)$ of a lambda-term Λ_1 (typically a "function") to a lambda-term Λ_2 (the "argument"). Let *var* be any variable, the recursive grammar of lambda-terms is: $\Lambda ::= \mathit{var} \mid (\lambda\ \mathit{var} \,.\, \Lambda) \mid (\ \Lambda\ \Lambda\ )$. As usual we assume that application is left-associative, hence when we write $\Lambda_1\ \Lambda_2\ \Lambda_3\ \ldots \Lambda_n$ we mean $(\ldots((\Lambda_1\ \Lambda_2)\ \Lambda_3)\ldots\Lambda_n)$. We also assume that the scope of a λ-abstraction binds as much to right as possible, hence when we write $\lambda x_1 \,.\, \lambda x_2 \,.\, \lambda x_3 \,.\, \Lambda$ we mean $(\lambda x_1 \,.\, (\lambda x_2 \,.\, (\lambda x_3 \,.\, \Lambda)))$. Finally, we

[4] For an extense list of web-pages related with lamda-calculus, several of them containing lambda-calculus interpreter implementations, visit `http://okmij.org/ftp/Computation/lambda-calc.html`.

can avoid the repetition of λs in consecutive λ-abstractions, hence when we write $\lambda x_1, x_2, x_3 \,.\, \Lambda$, we mean $\lambda x_1 \,.\, \lambda x_2 \,.\, \lambda x_3 \,.\, \Lambda$.

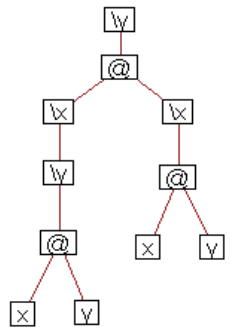
Variables in lambda-terms may occur *free* when they are not bound by any λ-abstraction. Terms are considered identical modulo renaming of bound variables. By $\Lambda_1[x \mapsto \Lambda_2]$ we denote the *substitution* of variable x occurring free in Λ_1 by Λ_2; this substitution can not capture variables occurring free in Λ_2 hence, bound variables in Λ_1 are renamed if necessary. *Redexes* are subterms of the form $((\lambda x \,.\, \Lambda_1)\ \Lambda_2)$, *$\beta$-reducing* a redex like this results in $\Lambda_1[x \mapsto \Lambda_2]$. When a term does not have any redex, it is said to be in *normal-form*. A redex occurs at the left of another if its first λ-abstraction appears further to the left. The *leftmost outermost* redex is the leftmost redex not contained in any other redex. The *normal reduction order* is the one that consists of reducing firstly the leftmost outermost redex.

3 Description of TILC

TILC parser for lambda-terms is as usual: the λ symbol is \, variables are words starting with lower-case letters, and names of defined lambda-terms are words in capital letters. The lambda-terms parser also allows us to use typical assumptions like left-associativeness of application, scope of lambda-binder and lambda-abstraction repetition. Roughly speaking, the graphical representation for the lambda-terms is its parse-tree where the non-terminal production for application is made explicit with the binary symbol `@`. In other words, it is the tree representation of the translation of the lambda-term to a first-order syntax where application is the binary function symbol `@` and lambda-abstractions are unary function symbols labelled by the variable that is being abstracted `\x`. This transformation can be obtained by means of this recursive rule:

$$\begin{aligned}
\mathcal{F}(\, x \,) &= \texttt{x} && \text{where } x \text{ is a variable translated into } \texttt{x} \\
\mathcal{F}(\, \lambda x \,.\, \Lambda \,) &= \texttt{\textbackslash x}(\, \mathcal{F}(\Lambda) \,) && \text{where } \texttt{\textbackslash x} \text{ is the corresponding unary function symbol} \\
\mathcal{F}(\, \Lambda_1\ \Lambda_2 \,) &= @(\, \mathcal{F}(\Lambda_1), \mathcal{F}(\Lambda_2) \,)
\end{aligned}$$

For instance, $\mathcal{F}(\ \lambda y \,.\, ((\lambda x \,.\, (\lambda y \,.\, (x\, y)))(\lambda x \,.\, (x\, y)))\ )$ results into the following *first-order* term:
`\y( @( \x( \y( @(x, y) ) ), \x( @(x, y) ) ) )`
which has this tree representation.



As we have already said, TILC aims to be a user-friendly visual experimentation platform for untyped lambda-calculus. Therefore, we provide the tools to help the user understand basic syntactical and operational sematic aspects. We enumerate some of them:

- Syntactical aspects:
  - Notational conventions: users can write terms according to convention. Marking subtrees and getting the sub-term highlighted with the same color is useful for students to get rid of the initial doubts with respect to syntax convention.

· Free and bound variable occurrences: users can highlight free-variable occurrences and bound-variable occurrences with their corresponding lambda-binders by selecting a node with the bound-variable, or its lambda-binder.
· β-redexes identification: users can highlight all β-redexes of the tree and see the corresponding subterm highlighted with the same color.

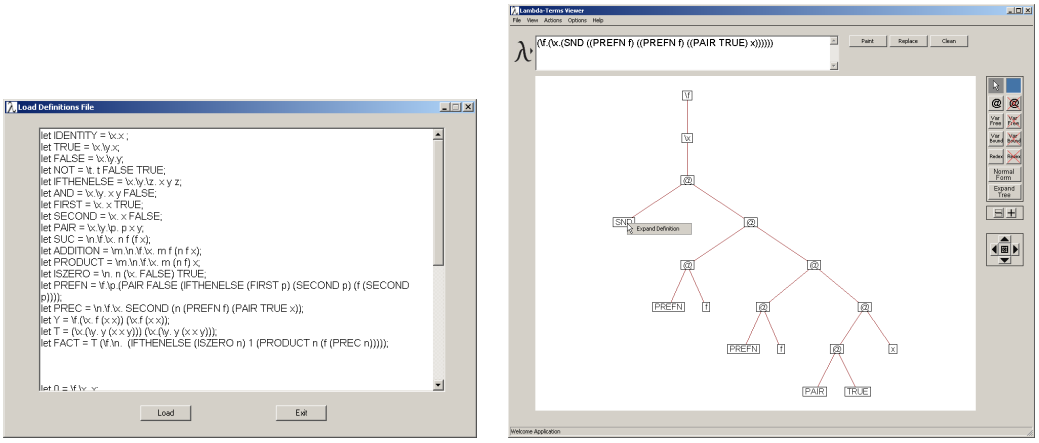


Fig. 1. Terms definition editor and partially expanded and normalized `PREC 2` lambda-term.

- Operational Semantics:
  · β-reduction: users can choose the β-redex they want to reduce by selecting it on the tree. Then they can see its effect on the tree and on the term.
  · normal-form: users can obtain the normal-form of a term (if it exists).
  · normal-order reduction sequence: users can obtain the normal-order reduction sequence textually, and the selected β-redex is underlined.
  · combinators definitions: users can define, save and load, any lambda-term, like the classical ones for church numerals. Once these are loaded, they can be freely used in the terms and graphically expanded in the tree.

In figure 1, we show the λ-terms definition editor and the partially expanded and reduced `PREC 2` term (predecessor of 2). But identification of β-redexes and the possibility of choosing the redex to reduce is the most attractive feature. In 2, we select a redex among the lot of redexes that has been highlighted at this stage of normalization of the fully expanded `PREC 2` term. Other more advanced aspects as *sharing* for laziness and high-performance in β-reductions are not currently considered because of our original pedagogical goal.

As a brief overview to the modular structure of TILC we can say that it relies on a main module `gui.hs` that uses the `wxHaskell` library to deal with the graphical interaction of the application. Nevertheless, the most important module is `Tree.hs`, this is the module in charge of the tree representation and manipulation of the lambda-terms; this module uses the module `Attributes.hs` that defines the required properties of the tree nodes. Apart from these modules, there are also a couple of parsing modules written using `Happy`: one of them translates lambda-terms into trees and the other one deals with `let` definition. There is also the module `Reduction.hs` that has all functions for performing β-reductions and normalization.

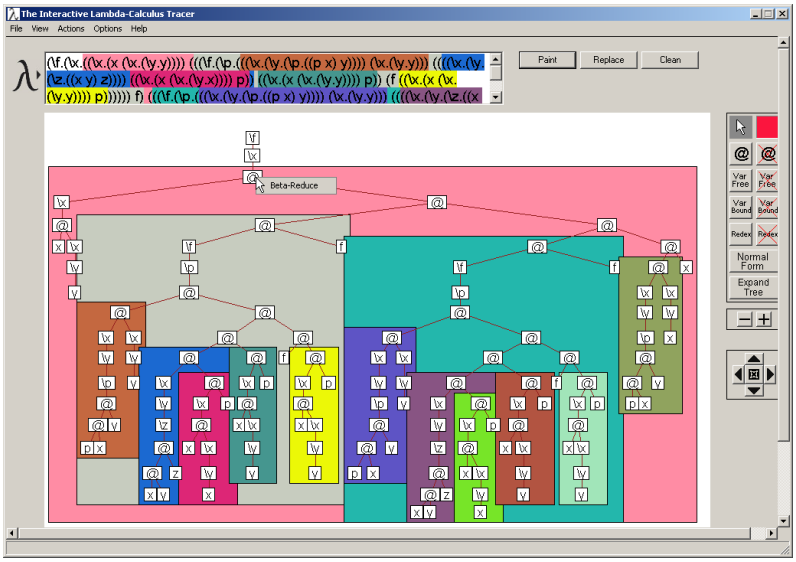


Fig. 2. Lots of β-redexes of fully expanded `PREC 2`.

4 Further Work

The good experience of using this tool in the classroom and as a downloadable tool for the students suggested us to think of extending it for dealing with other basic, and not so basic, features as: α-conversion and η-reduction, tracing substitution, including many other reduction strategies and sorts of normal forms, use and representation of *de Bruijn* style, adding types and type inference algorithm explanations as in [4], defined combinators recognition, etc.

The fact that the tool is written in `HASKELL`, apart from showing to the students that functional programming serves for making *cool applications* too, provides to TILC another potential interesting pedagogical value: allowing teachers to use some of its modules as a platform to ask the students to develop more features. Namely, one could remove the β-reducer module and ask the students to do it using the desired reduction strategy. Therefore, we are considering the possibility of providing free-access to a bounded version of the code where this β-reduction part is missing, and restricting access to the full code to teachers on-demand.

References

[1] Barendregt, H., "The Lambda Calculus. Its Syntax and Semantics." North-Holland, 1984.

[2] Champollion, L., J. Tauberer and M. Romero, *The penn lambda calculator: Pedagogical software for natural language semantics*, in: T. H. King and E. M. Bender, editors, *Proceedings of the GEAF07 Workshop* (2007), pp. 106–127, http://csli-publications.stanford.edu/GEAF/2007/geaf07-toc.html.

[3] Sestoft, P., *Demonstrating lambda calculus reduction*, in: *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, Lecture Notes in Computer Science **2566** (2002), pp. 420–435.

[4] Simoes, H. and M. Florido, *Typetool - a type inference visualization tool.*, in: *Proc. 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP'04)* (2004), pp. 48–61, technical Report AIB-2004-05, Department of Computer Science, RWTH Aachen, Germany.

[5] Thyer, M., *The lambda animator*, http://thyer.name/lambda-animator/.

[6] Urquiza-Fuentes, J. and J. A. Velázquez-Iturbide, *A survey of program visualizations for the functional paradigm*, in: *Proc. 3rd Program Visualization Workshop* (2004), pp. 2–9, research Recport CS-RR-407, Department of Computer Science, University of Warwick, UK.