Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 248 (2009) 173–183
www.elsevier.com/locate/entcs

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`, and the tools `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 archetypical minimal functional programming language and therefore the computational model for this paradigm. As in many other courses where

[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`

doi:10.1016/j.entcs.2009.07.067

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 build the factorial function and hence, to illustrate that any "recursive" function can be encoded within this formalism. Nevertheless, when one shows these encodings, students feel as if there were a kind of "*black magic*"... because, although it works and they can follow the proof of the soundness of the definitions, to really *see* why these encodings work, students have to practice.

TILC is motivated by the convincement that to appreciate the syntax, the notational assumptions, like left-associative of application, and the operational aspects of pure untyped lambda-calculus, students must experiment with it. Using a tool that deals with all these aspects in a friendly and graphical manner incentives this experimentation. 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 λ-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. This user-friendly nature of the tool encourages practicing.

Several works exist [4] dealing with the practice of lambda-calculus but none of them fits precisely with our educational purpose. In [6] *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 just works textually and does not deal with parse-tree representation. In [3] we find the graphical application *The Penn Lambda Calculator*. It focusses on teaching and practicing with lambda-calculus but it is applied to natural language semantics. Another graphical web-based tool is the *Lambda-Animator* [8] which goes one step further dealing 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 [9] 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.

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

TILC has been developed by David Ruiz as a diploma thesis and supervised by Mateu Villaret. It has been fully developed using `HASKELL`, and the tools `wxHaskell` [1] for the graphical interface and `Happy` [4] to build the parsers. Its home page is `http://ima.udg.edu/~villaret/tilc` from where Windows binaries can be freely downloaded. Other platform binaries and source code are under preparation.

The paper proceeds as follows: in Section 2 we briefly define the language that is considered in TILC by recalling the basic concepts of pure untyped lambda-calculus. In Section 3 we explain the main features of the tool and illustrate them by means of examples, we also overview the modular architecture of the tool. Finally, in Section 4 we conclude and explain the forthcoming extensions of the tool suggested from the first impressions collected from its usage in Girona and by some other inputs from other places where the tool is also being used.

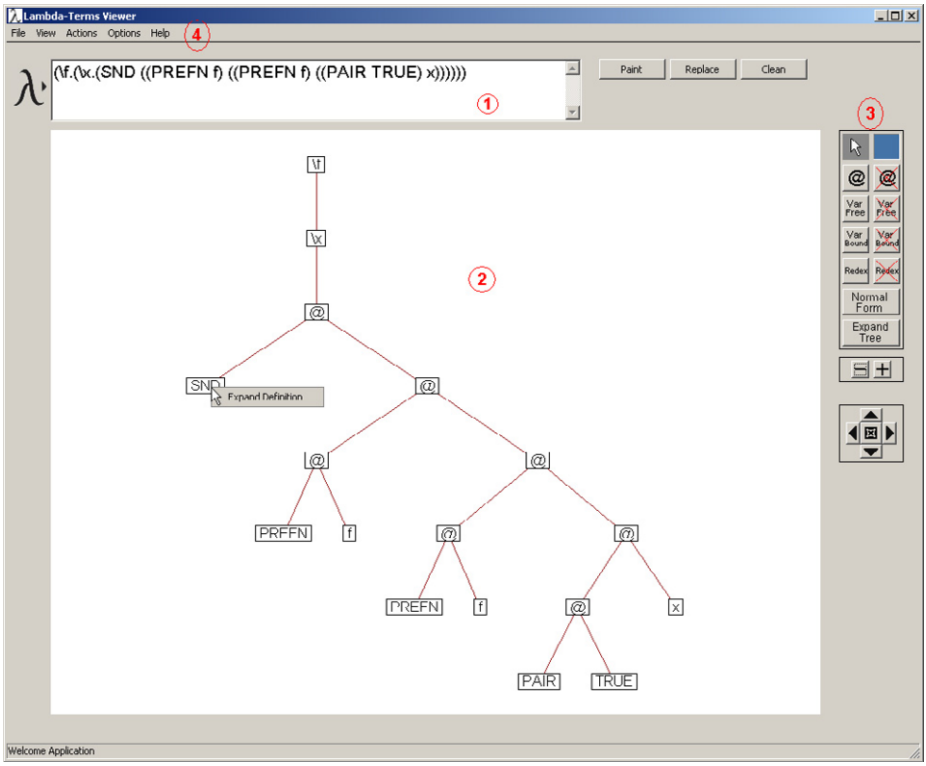

Fig. 1. Main parts of the application with the example of the partially expanded and normalized `PREC 2` lambda-term.

2 Recalling Pure Untyped Lambda-Calculus for TILC

Pure untyped lambda-calculus used in our framework relies on [2] as the standard reference, here we only informally recall the basic concepts directly concerning our application.

The object of study of lambda-calculus are *lambda expressions* or *lambda-terms*.

The syntax of these lambda-terms may be defined with a BNF grammar. First of all we assume that we have some infinite set V of *variables*. We use $x, y, z, ...$ to stand for arbitrary variables. The grammar for lambda-terms is:

$$t \ ::= \ x \quad | \, (\lambda \, x \, . \, t) \quad | \, (\, t \ t \,)$$

where $x \in V$. Expressions of the form $(\lambda x \, . \, t)$ are called *lambda-abstractions* and intuitively they refer to functions that, given an argument x, return the value t; we say that the *scope* of the λ-*binder* λx extends to t. Expressions of the form $(t_1 \ t_2)$ are called *applications* and intuitively correspond to the application of a function t_1 to the argument t_2.

There are some syntactic conventions that are generally convenient to avoid the extensive use of parentheses but that are confusing to learn: application is left-associative, hence when we write $t_1 \ t_2 \ t_3 \ \ldots t_n$ we mean $(\ldots((t_1 \ t_2) \ t_3) \ldots t_n)$. The scope of a λ-binder binds as much to the right as possible, hence when we write $\lambda x. \, \lambda y. \, \lambda z. \, t$ we mean $(\lambda x. \, (\lambda y. \, (\lambda z. \, t)))$. Finally, we can avoid the repetition of λs in consecutive λ-binders. When we write $\lambda x, y, z \, . \, t$ we mean $\lambda x. \, \lambda y. \, \lambda z. \, t$.

Variables in lambda-terms may occur *free* or *bound*: we say that a variable x occurs free in a term if it is not within any scope of a λ-binder λx, otherwise we say that x is bound by the closer λx binder. For instance consider the folowing term:

$$(\lambda x. \, (\lambda y. \, x)(\lambda x. \, (x \ y)))$$

variable x occurs bound twice: the leftmost occurrence of variable x is bound by the first λx binder whilst the rightmost one, although being in the scope of the first λx binder too, is bound by the closest λx binder, i.e. by the second one. Variable y occurs free because it is not in the scope of any λy binder.

Concerning the operational semantics of lambda-calculus *substitution* plays a critical role. By $t_1[x \mapsto t_2]$ we denote the *substitution* (or capture avoiding variable replacement) of free occurrences of variable x in t_1 by t_2; this substitution must not capture variables occurring free in t_2 hence, when necessary, λ-binders and corresponding bound variables in t_1 are properly renamed preserving bindings. For instance:

$$(\underline{\lambda x_1}. \, \mathbf{x_2} \, \underline{x_1} \, (\lambda x_2 . x_2)) \, [\mathbf{x_2} \rightarrow \mathbf{x_1}(\lambda \mathbf{x_3} . \mathbf{x_3})] \ \Rightarrow \ (\underline{\lambda y}. \, (\mathbf{x_1}(\lambda \mathbf{x_3} . \mathbf{x_3})) \, \underline{y} \, (\lambda x_2 . x_2))$$

Lambda-calculus is based on three main equivalence rules, α, β and η. The α-equivalence rule allows us to rename λ-binders and corresponding bound variables whenever we preserve the bindings and the freedom of the occurrences of the variables. For instance, $\lambda x. \, x \, y$ is α-equivalent to $\lambda z. \, z \, y$, while it is not to $\lambda z. \, x \, y$ neither to $\lambda y. \, y \, y$. We skip the η-equivalence rule because it is not considered at all in TILC. The β-equivalence rule defines the mechanism of applying a function to its argument. We look at it as a reduction rule because we are in a programming course. The *β-redexes* (redexes for short) are subterms of the form $((\lambda x. \, t_1) \ t_2)$, *$\beta$-reducing* a redex like this results into $t_1[x \mapsto t_2]$. 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 lambda-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. For instance, consider the following term that is not in normal form because it has two redexes:

$$\underline{(\lambda x.y)(\overline{(\lambda z.\, z\, z\, z)(\lambda z.\, z\, z\, z)})}$$

if we follow the normal reduction order, we first reduce the outermost redex, the underlined one, and hence we obtain the term y which is obviously in normal form. But, if we reduce the innermost redex, the one that is overlined, we obtain the term

$$\underline{(\lambda x.y)(\overline{((\lambda z.\, z\, z\, z)(\lambda z.\, z\, z\, z))}(\lambda z.\, z\, z\, z))}$$

that is not in normal form, moreover, if we continue reducing innermost redexes, it will never be. Fundamental results on lambda-calculus show that the normal form of a term, if it exists, is unique, moreover the normal reduction order would allow us to find it.

3 Description of TILC

As we have already said in the introduction, TILC is a graphical application that has an area where lambda-terms are textually introduced (1 in Figure 1), and a panel (2 in Figure 1) where the parse-tree of the term is represented and can be manipulated using the buttons at its right (3 in Figure 1). These manipulations are automatically reproduced in the textual part (1 in Figure 1). The menu (4 in Figure 1) allows the user to deal with lambda-terms definitions and with textual normalization of terms.

The syntax required for introducing lambda-terms is as usual: the λ symbol is the backslash symbol \, variables are words starting with lower-case letters, and names of defined lambda-terms are words in capital letters. We can also use typical conventions like left-associativeness of application, scope of λ-binder and λ-binders repetition avoidance.

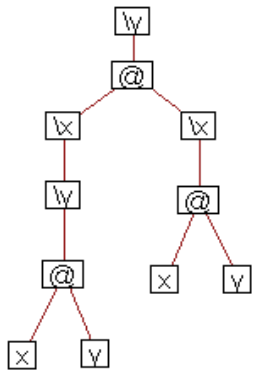

Fig. 2. Tree representation for $\lambda y.\,((\lambda x.\,(\lambda y.\,(x\,y)))(\lambda x.\,(x\,y)))$.

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 x that is being abstracted `\x`. This transformation can be obtained by means of this recursive rule:

$$\begin{array}{lll} \mathcal{F}(\, x\,) & = \texttt{x} & \text{where } x \text{ is a variable translated into } \texttt{x} \\ \mathcal{F}(\, \lambda x\,.\, t\,) & = \texttt{\textbackslash x}(\, \mathcal{F}(t)\,) & \text{where } \texttt{\textbackslash x} \text{ is the corresponding unary function symbol corresponding to the binder } \lambda x \\ \mathcal{F}(\, t_1\ t_2\,) & = @(\, \mathcal{F}(t_1), \mathcal{F}(t_2)\,) & \text{where } \texttt{@} \text{ is the binary function symbol denoting application} \end{array}$$

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 the tree representation of Figure 2.

The advantages of providing a tree representation for the lambda-terms are quite obvious, for instance, β-redexes are easily identified because we just need to identify subtrees with the `@` symbol as the root and having, as the left child, a lambda-abstraction, i.e. a subtree with root `\x`. In other words, β-redexes are subtrees of the form `@(\x(_),_)`. Other concepts like *scope* for variables or the *leftmost ourtermost* redex, have also direct visual interpretations on the tree.

3.1 Main Features

To make of TILC the user-friendly visual experimentation platform for untyped lambda-calculus 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 λ-binders by selecting a node with the bound-variable, or its λ-binder.
  - β-redexes identification: users can highlight all β-redexes of the tree and see the corresponding subterm highlighted with the same color.
- 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.

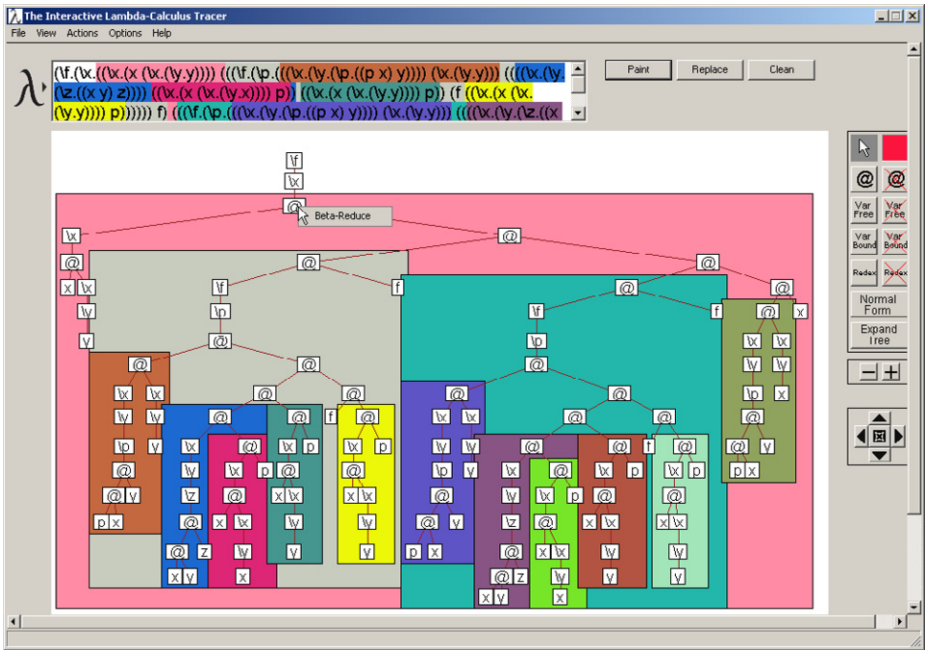

Fig. 3. Lots of β-redexes of fully expanded `PREC 2`.

- 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, where the selected β-redex of each step is underlined.
- `let` 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.

3.2 TILC Throught Examples

We illustrate with some screenshots these main features.

Apart from manipulating the parse-tree of the lambda-terms, users can introduce some lambda-terms definitions and save and load them whenever they want. The names of the defined terms are treated by default as free variables and they are typically written in capital letters. Users may expand them on demand (using the right-button of the mouse over the names box in the tree) or all at once. In figure 4, we show the lambda-terms definition editor and in Figure 1 we can see 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 Figure 3, 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.

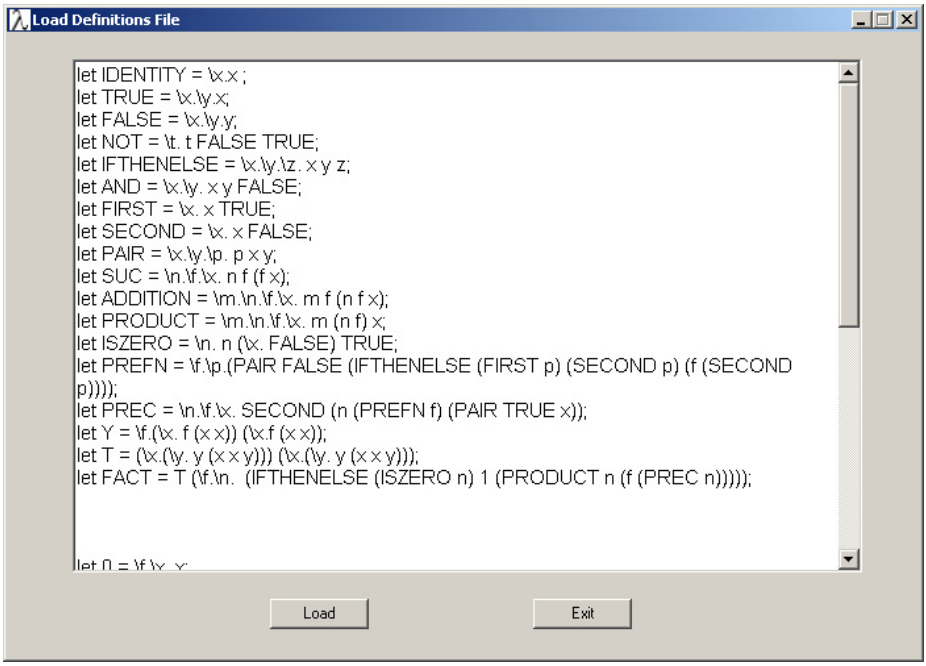

Fig. 4. Terms definition editor.

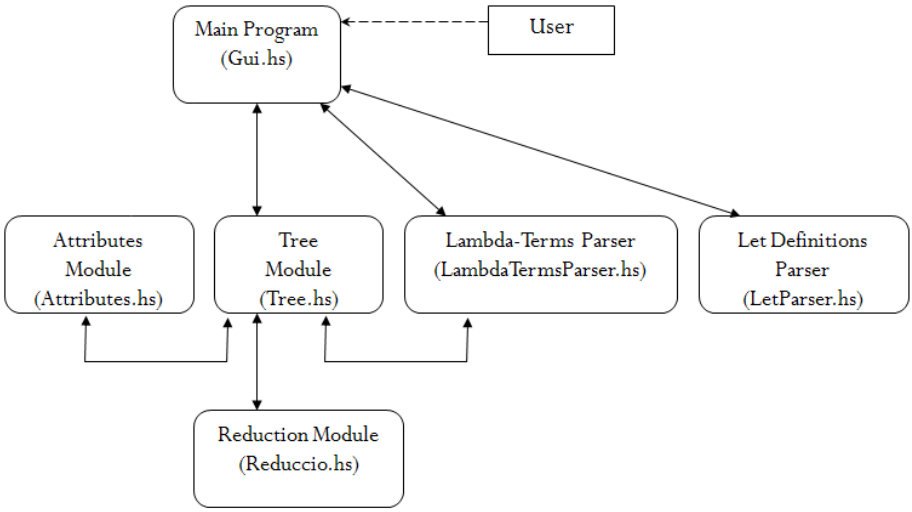

Fig. 5. Modules schema.

3.3 The TILC Modular Structure

The modular structure of TILC (see Figure 5) relies on a main module `Gui.hs` that uses the `wxHaskell` library to deal with the graphical interaction of the application. `Gui.hs` has the functions used to create the interaction between the text of the lambda-term that we are working with, and its tree representation. Nevertheless, the most important module is `Tree.hs`, this is the module in charge of the tree representation and manipulation of the lambda-terms according to the demands of the

user through the `Gui.hs` module (e.g. drag&drop the nodes, mark subtrees, identify free and bound variables, mark and identify β-redexes, ...). The lambda-term introduced textually is parsed with the parser of module `LambdaTermParser.hs` obtaining firstly, an intermediate tree-structure that is an instance of type

```
data LExt = LEmpty
          | LNode String [LExt]
```

and secondly the definitive tree-structure that adds a list of `Attributes` to the nodes

```
data Tree = Empty
          | Node [Attributes] [Tree]
```

The list of attributes contain necessary information for several manipulations of the trees like for instance the label of the node, the position of the nodes and its area, the status of dragging with respect to the node, etc. The function for obtaining the balanced drawing of the tree is based on [5].

```
data  Attributes = Name String
                 | Position Point
                 | DragState Bool
                 | Area Size
                 | ColorNode Color
                 | Iden Int
                 | VarType Int
                 | ...
```

There is also another parsing module to deal with `let` definitions (`LetParser.hs`) and translate them into a pair of `(String, Tree)`, where `String` is the name of the definition and `Tree` is the parsed tree-structure obtained from the definition.

To deal with the β-reductions and normalization using the normal order reduction, we have implemented the module `Reduction.hs`. This module interacts with the module `Tree.hs` and obtains its corresponding lambda-term of type

```
data Lterm = Var String
           | Abst String Lterm
           | Appli Lterm Lterm
```

In Figure 6 we can see the process followed by a lambda-term introduced textually then drawn, then normalized, drawn again and textually printed.

4 Conclusion

We have introduced TILC, the interactive lambda-calculus tracer. We have argued why we believe that this tool helps in teaching/learning main basic pure untyped lambda-calculus concepts by showing its main features. We have also presented its modular structure. Now, we want to point out the main extensions that we are

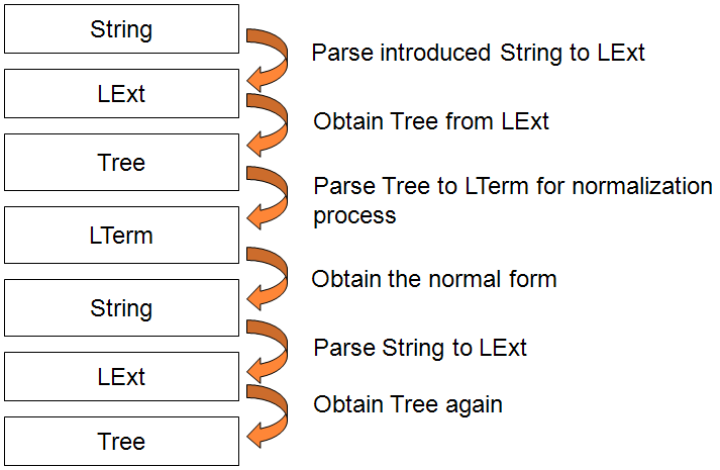

Fig. 6. Normalization process throughout data types.

currently developing.

The good experience of using this tool in the course lectures 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 other reduction strategies and sorts of normal forms, use and representation of *de Bruijn* style, adding types and type inference algorithm explanations as in [7], defined combinators recognition, etc. We are currently working in some of these extensions.

It is quite feasible that lambda-calculus is taught in a course where also `HASKELL` (or other functional programming languages) is taught. 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.

We would also like to acknowledge the suggestions for improving the tool that we have received from people using it in their lectures like Salvador Lucas from Universitat Politècnica de València, Temur Kutsia from RISC and Manfred Schmidt-Schauß in Johann Wolfgang Goethe-Universität.

References

[1] `WxHaskell`, `http://haskell.org/haskellwiki/WxHaskell`.

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

[3] 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.

[4] Gill, A. and S. Marlow, *Happy: The parser generator for haskell*, http://www.haskell.org/happy.

[5] Kennedy, A. J., *Functional pearls: Drawing trees*, Journal of Functional Programming **6** (1995).

[6] 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.

[7] 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.

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

[9] 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.