A Visual Lambda-Calculator Using Typed Mind-Maps

Li-Ren Chien and Daniel J. Buehre
Department of Computer Science and Information Engineering, Chung Cheng University
#168, University Rd, Min-Hsing, Chia-Yi, Taiwan, R.O.C
clj@cs.ccu.edu.tw, dan@cs.ccu.edu.tw

Abstract

Lambda calculus is an influential and extensively-used notation for describing computable functions, and Mind-Mapping is widely used as an expression of radiant thinking via a powerful graphical technique. In this paper, we introduce a completely visual representation based on typed Mind Maps to represent steps of calculation for a pure untyped lambda calculator, VLM. This visual representation has several advantages over traditional textual and visual calculators. VLM uses typed Mind Maps for both the lambda calculator input and output. Although VLM is designed as a computable typed Mind Maps node of our Typed Mind Maps API project [1], it can also be applied to learning and teaching the concepts of lambda calculus as a visualization of traditional textual rewrite steps. Moreover, the lambda calculus queries and the results of queries are both represented as FreeMind files, and that allows them to be organized and deployed easily. However, the Mind-Mapping of lambda calculus is interesting and elegant in its own right.

1. Introduction

After developing DICE, which is a parse-tree-based automatic assessment system with a TDD (test-driven development) model [2], we have defined a systematic and symbolic model for typed Mind Maps that was used as a uniform data model in an e-learning system [1]. So now the typed Mind Map is not only a content repository (i.e. ‘organizing information via hierarchies and categories [3]’) but also a computational graph, showing steps of computations.

At the beginning, we proposed a computable node in our typed Mind Map API as a general mechanism for dynamic transitions between different Mind Maps or as a conversion description from different data models to typed Mind Maps. It was natural to integrate typed Mind Maps with lambda calculus since lambda calculus is a successful model of the computable functions, and there are also typed versions [4]. We found mind maps to be very suitable for visualizing the processing of lambda calculus expressions. It can be applied to learning and teaching the concepts of lambda calculus as a visualization of traditional textual rewrite steps. The visual lambda calculator with typed Mind Maps was named VLM.

It is quite difficult to teach and learn lambda calculus in a programming language course. For example, think about evaluating the successor of Curry number 0. The execution of the lambda expression SUCC 0 involves combinators SUCC and 0, the definition of the Curry number 0, the rewrite steps of $\beta$ reduction and the equivalence of the evaluated result with Curry number 1. For a learner, it is challenging to decipher the nested lambda expressions by using useful visual cues. For both of these problems, however, a visual lambda calculator is apparently helpful.

Although graphs have been used to express lambda expressions [5, 6] since Wadworth [7], textual lambda calculators have been widely used in different areas. We were amazed to discover that there does not seem to be any educational tool suited for generating standalone materials of pure visual lambda calculus evaluation. The Penny lambda calculator [8] might be the closest project to this task. But we are interested in visualizing a lambda calculus evaluation itself, while Penny was concerned with computing the compositional semantics of natural language syntax trees.

VLM is a standalone Java-based application with executable functions from the OS level. A reader can run VLM by OS command under OS level operations. The unit of data in the OS level operation is stored on files. VLM takes a FreeMind compatible file, which contains lambda calculus query information, and generates a file with same format as the output of the query result. A programmer can take the VLM API,
which consists of typed Mind Map algebra, query system, transformation system and delivery system, as a Java API in their own project.

This paper describes a visual lambda calculator with typed Mind Maps. In Section 2, we present a motivating example, comparing the evaluation of a complex lambda expression (the Y combinator) by using either VEX or VLM. Section 3 describes the implementation of VLM from a user’s viewpoint. Section 4 concludes with areas for future work.

2. Motivating Example

Figure 1 shows a typical VLM query with a typed Mind Mapping. It expresses that a lambda expression \((Y \text{ e})\) needs to be calculated. The root of a VLM query mind mapping has type ‘LambdaQuery’. That consists of a ‘Query’ child node, a ‘Setting’ child node and a ‘Combinators’ node. A setting node rearranges the VLM evaluation environment dynamically at runtime. The ‘Combinators’ node integrates combinator groups that are distributed across the network by URLs.

The Y combinator, defined by \(\lambda f. (\lambda x. (f (x x))) (\lambda x. (f (x x)))\), is also a lambda expression. In this example, and in the concrete syntax definition of VLM in the next section, we rely on the survey by W. Citrin [6] and M. Erwig [9].

The recursive behaviour of the Y combinator is a good paradigm to expose the faults of a textual version calculator compared with a graphical version. The readers should refer to the survey by W. Citrin [6] to contrast the “surprising and non-intuitive behavior of the textual version with the more intuitive behavior of the graphical version”. We take this as our opening example to compare query and calculus processing in VLM and VEX.

2.1. VLM representation of the Y Combinator

The Y combinator was defined as \((\lambda f. ((\lambda x. (f (x x))) (\lambda x. (f (x x))))\) in text while, as Figure 3 shows, VEX represented it by the combination of a vertex-labeled graph and Euler diagrams. The arcs in the VEX representation indicate the binding relation between lambda variables, and the circles of the Euler diagram show the hierarchical relationships between lambda expression terms (i.e. variables, applications and abstractions).

VLM makes use of pure graph theory to represent the lambda calculus expression by dividing it into data parts, through a vertex-labeled directed rooted graph,
VLM classifies the representation of visual lambda calculus into two types, as indicated in the following two subsections.

### 2.1.1. Lambda Expression Terms.

We aim to develop a visual language using Mind Mappings. Consequently, VLM visualizes the $\lambda$-terms as a vertex-labeled directed rooted graph, for 'a Mind Mapping is a vertex-labeled directed rooted graph' [2] while VEX represents $\lambda$-terms with an Euler diagram.

Actually, a lambda expression term is defined to be a binary tree since the number of child nodes of a lambda expression term is not more than two. In our view, a tree is more natural and suitable than an Euler diagram for the visual illustration of a hierarchical definition such as $\lambda$-terms. In the meanwhile, we obeyed the intuitive view of the definition of $\lambda$-terms by Herk [4].

As Figure 4 shows, the green node labeled by ‘Comb.(Y)’ denotes a combinator named Y. Its only child node is labeled with ‘Abs.’ It shows a lambda abstraction term ($\lambda$x.M), where x is a lambda variable term and M is a $\lambda$-term that is represented by the ‘Abs.’ node’s ordered children. The first child labeled by ‘Var(\lambda).’ with a light blue background represents a $\lambda$-variable with its child ‘f’ as a value. The secondary child of ‘Abs.’ labeled by ‘App.’ shows a lambda application term form (M N) where M and N are both $\lambda$-terms, and so on.

### 2.2.2. Lambda Calculus Operations

Some lambda calculus operation definitions are distinct from $\lambda$-terms in VLM. For example, the lambda variable terms are divided into bound variables (i.e. Var(B)), free variables (i.e. Var(F)) and lambda variables (i.e. Var(\lambda)).

As Figure 4 shows, the purple arc links a lambda variable to the bound variable, indicating the binding relation between them. In VEX, the relation is shown in dependent trees.

Moreover, we present a lambda application form ($\lambda$x.M) N that needs to substitute N for the x’s in M, as indicated by red and blue arrows. The learner can easily trace the rewrite processing by following the red/blue arcs.

### 2.2 The Lambda Calculus Flow in VLM

Figure 5 shows the how VEX visualizes the substitution processing of applying the Y combinator to a variable e. The upper left sub-figure shows the lambda expression (Y e) and the upper right diagram gives the result after substituting the variable e for f. Finally, the lower picture shows Y e = e (Y e) in VEX by comparing it with the upper right one.

Figure 6 shows the partial result of expanding node labeled ‘1’ in Figure 1, which displays the VLM version of (Y e). The red/blue arrows indicate the steps of $\beta$-reduction. The learner can take the free variable e and follow the red arrow reaching the lambda variable f. Again, we follow the blue line leading to the bound variable f and replace bound variable f by the free variable e.
Fig. 7. VLM version of \((\lambda x. e(x \ x))(\lambda x. e(x \ x))\)

There are two red arrows in Figure 6 that stand for two steps of substitution to be done in this phase. A child node named 'processing' in the next black node labeled '2' in Figure 7 shows the result of these two substitutions. Although two main methods (i.e. call by value and call by name) were introduced, VLM exhibits the normal-order reduction, for it is guaranteed to reach a normal form if one exists.

Fig. 8. VLM version of \(e(Y \ e) = (\lambda x. e(x \ x))(\lambda x. e(x \ x)) = (Y \ e)\)

Figure 8 is analogous to the lower part in Figure 5, the ‘App.’ node conjoined after the ‘Result’ node shows a free variable \(e\) is rewritten out from Figure 5, as well as keeping same lambda expression as Figure 5 in its ‘App.’ sub-node.

2.3 The Automatic Combinator Generator in VLM

The evaluation of the equation \((Y \ e) = e(Y \ e)\) can be illustrated more clearly with the handling of an automatic combinator generator in VLM. Referring to Figure 9, the nodes with a green background color indicate combinators that can be used in a lambda expression evaluation. The text in the combinator node uses brackets to identify it as a combinator. The underscore means the combinator is generated by the system for each step of the evaluation, while the combinators without underscores are defined by the user.

VLM applies an \(\alpha\) -reduction to rename the bound variable in a lambda combinator. Then it finds out the equivalent combinator from known combinators by tree pattern matching, since we have defined the lambda expression terms of a tree in Section 2.1.1.

Fig. 9. Combinators in VLM

2.4 The \(\alpha\)-conversion and \(\eta\)-reduction

In the last few subsections, we showed the \(\alpha\) -reduction in VLM by comparing it to VEX. In this subsection, we describe how VLM represents two main rewrite rules; \(\alpha\)-conversion and \(\eta\)-reduction.

The \(\alpha\)-conversion can also be called “re-naming”, since \(\lambda x. M = \lambda z. \{z/x\} \ M\) where \(z\) is free in \(M\). It shows the bound variables may be consistently re-named. The \(\alpha\)-conversion renames the bound variables of a lambda expression by following the bound links (purple arrows), marked by clouds with different colors in VLM.

Figure 10 shows the \(\alpha\) -conversion of lambda expression \((\lambda m. (\lambda n. (\lambda f. (m (n f))))))\). VLM denotes the original value and the value after conversion in a ‘Textual’ sub-node through \(\alpha\)-conversion.
Automatic $\alpha$-conversion of MULT combinator in VLM

The $\eta$-reduction relates an abstraction to its underlying expression body in certain cases: $\lambda x. (M x) = M$ if $x$ is free in $M$. Again, VLM shows the $\eta$-reduction by a dark cloud with a 'Textual' sub-node denoting the original and converted value. Figure 11 shows a $\eta$-reduction from $(\lambda x. ((\lambda x. (e x)) x))$ to $e$.

In Section 2, we have given a brief description of VLM through a motivating example and contrast it with a VEX representation. The basic idea is to divide lambda calculus into data structures (i.e. lambda expression terms) and algorithms (i.e. lambda evaluation) and integrating them into a single colorful graph.

3. Implementation

In this section, we describe the implementation of VLM from the end-user viewpoint. The VLM API which based on formal definition of VLM visual lambda expression (see Appendix A) is packaged as a classical Java package which can be used as a Java class as well as be executed by a sequence of OS batch commands.

We relied on a widely-used freeware named FreeMind, written in Java, for the creation of mind maps. [11] The electronic mind maps with extensions .mm that are created by FreeMind obey the XML specifications. A tree-based node collection named ‘node’ with a ‘TEXT’ attribute denotes the text in a node that is enclosed by a root node named ‘map’. As a knowledge representation, FreeMind-like XML schema simplifies the Mind Map directly from a visual image into XML nodes named ‘node’ with a ‘Text’ attribute. The interesting terms in a domain are denoted by the ‘TEXT’ attribute in the ‘NODE’ node. The inductive definition of a node provides a tree-based collection to organize nodes. The ‘LINK’ attribute in a node extends the tree into a directed rooted graph. By restricting the link to have no self-references, the graph becomes a Directed Acyclic Graph (DAG).

The implementation of our typed Mind Map API is based on FreeMind-like XML schema. A FreeMind-like Mind Map file can be read into the API as a typed Mind Map node. A typed Mind Map algebra supports rich operations to operate on Mind Maps. A tree-based query function retrieves suitable nodes from Mind Maps.

VLM takes the typed Mind Map API as a basis of input and output. This makes VLM intrinsically an XML-based data format. The core of VLM receives a query typed Mind Map which includes a setting node to set the VLM evaluation environment. The setting will influence the type of result Mind Maps for flexibility of execution.

As Figure 12 shows, the query can be distributed, deployed and organized easily via an IP network by typed Mind Map links. One can categorize combinators on different hosts as a combinatory group, while the other people can use the combinator group in their queries. In VLM, built-in combinators consist of arithmetic, logic, pair and Curry numbers that can be used in a beginning lambda calculus teaching job. The query result is a persistent standalone FreeMind compatible file which can be easily used for learning and teaching materials.

For the advanced user, the query results are treated as typed Mind Map nodes which are formed as Java tree nodes in our typed Mind Maps API. The typed
Mind Map API has a set of operations for a typed Mind Map algebra, query system, transformation system, and delivery system. Those operations support a wide use of VLM based on typed Mind Maps. For example, one can use the typed Mind Map delivery system to transfer a student's query from a Web-based server to his teacher’s PC. A lambda expression parser is established by using SableCC (a LALR parser generator) [12]. One can use the lower level parser result for his own evaluating processes as well as the typed Mind Map result.

We have given a brief description of the implementation of VLM from the end-user and the advanced-user viewpoints in this section. VLM is a Free-Mind compatible lambda expression calculator from the end-user viewpoint, while the advanced users treat VLM as a Java API with typed Mind Maps, a lambda parser and lambda calculator support.

4. Conclusions and Future work

Integrating everything that happens in a lambda calculus evaluation is a highly sophisticated and interesting challenge. The evaluation rules seem to be simple, but are in fact not simple. As instructors argue, ‘A sentence of just ten words can easily fill an entire blackboard and take half an hour to draw’ [8], it is obviously difficult to teach beginners. The first benefit of this paper supports a light-weight visual lambda calculator which generates the steps of the evaluation process of a lambda expression. It is a useful tool for learning and experimenting with the lambda calculus. Secondly, the instructor can use VLM to build and organize a distributed lambda combinator collection with typed Mind Maps. VLM will be published as freeware soon. (Download the beta-version of VLM from http://www.cs.ccu.edu.tw/~clj/LambdaMM.zip)

6. References


