

UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science Speciality

Viktor Massalõgin

Visual Lambda Calculus

Master thesis (20 AP)

Supervisor: Prof. Varmo Vene

Author:	Viktor Massalõgin	“....” May 2008
Supervisor:	Prof. Varmo Vene	“....” May 2008
Allowed to defense		
Professor		“....” May 2008

TARTU 2008

Contents

Introduction.....	3
1. Lambda calculus basics.....	4
2. Existing visualizations of lambda calculus.....	9
2.1. Programming with Visual Expressions (VEX) by Wayne Citrin.....	10
2.2. A Graphical Notation for the Lambda Calculus by Dave Keenan.....	11
2.3. Lambda Animator by Mike Thyer.....	11
2.4. Puzzle game Alligator Eggs by Bret Victor.....	13
2.5. Motivation for a new notation.....	13
3. Bubble notation.....	14
4. Interface.....	19
5. Implementation.....	22
6. Student experiments.....	26
7. Conclusion.....	27
7.1. Summary.....	27
7.2. Future work.....	27
Resümee (in Estonian).....	30
References.....	32
Appendix.....	33

Introduction

The history of visual programming languages (VPL) starts in 1963 with the Sketchpad system by Ivan Sutherland [1]. Over time VPL gained in importance in human-computer interaction study, especially for education. This thesis doesn't aim to prove the importance of VPL. Just note that the main argument of such proofs is the primacy of the perception of images over the perception of text [1], thus a graphical representation may successfully supplement or even substitute a textual statement.

This thesis is devoted to the development of a visual notation for lambda calculus expressions and a VPL environment based on this notation. We believe that lambda calculus is especially suited for visualization purposes. The simplicity of lambda calculus allows defining simple transparent semantics for notations. It also allows keeping notations purely graphical (meaning that no literal labeling is needed). At the same time such a visual language (along with lambda calculus) is powerful enough to denote Boolean operations, numbers, lists, recursion etc [2].

The main requirements for the developed notation and the visual programming environment are an exact and clear representation of lambda expressions and an intuitive interface for their manipulation respectively. Such an environment may be used as a learning aid for teaching lambda calculus, as a mental game for children or as an assistant tool for research.

This thesis is organized as follows. Section 1 introduces and reviews some basic definitions of lambda calculus with let-expressions and lazy evaluation. This section may be skipped by an experienced reader. Section 2 gives an overview of four existing graphical notations for lambda calculus and describes their benefits and drawbacks. Section 3 specifies the *bubble notation* proposed by the author. This notation combines the benefits of the notations cited above. Section 4 describes the user interface of the *Visual Lambda* environment prototype based on the bubble notation. Section 5 gives the main aspects of the implementation of the environment in the Python programming language. Section 6 reports the results of lessons with two students. The lessons consisted of an introduction to the lambda calculus and the solving of tasks (both using the Visual Lambda environment). Section 7 presents some conclusions and thoughts on future work. Visual Lambda is the open source project licensed under GPLv3 license. A copy of the project source (as of 27.05.2008) is available online and on the attached CD.

1. Lambda calculus basics

This section is a short and informal description of lambda calculus. For more complete description we recommend any of the introductions to lambda calculus such as [2] or [3].

Lambda calculus theory is tightly bound to the notion of function. Consider a definition of a function. Traditionally we specify function arguments by the side of a name of a function being defined. As an example, the definition of the two-argument function `LENGTH` can be given as:

$$\text{LENGTH}(x,y) := \sqrt{x^2+y^2}$$

The lambda syntax allows uniting arguments with the function body. For example,

$$\lambda x y . \sqrt{x^2+y^2}$$

defines a vector length function without being bound to any name. The following notation

$$\text{LENGTH} \equiv \lambda x y . \sqrt{x^2+y^2}$$

denotes that `LENGTH` and $\lambda x y . \sqrt{x^2+y^2}$ are synonyms.

An important point is that every function in lambda calculus takes its arguments one by one. As an example, consider the evaluation of the function `LENGTH`:

$$\begin{aligned} \text{LENGTH } 5 \ 12 &\equiv (\lambda x y . \sqrt{x^2+y^2}) \ 5 \ 12 \rightarrow \\ &(\lambda y . \sqrt{5^2+y^2}) \ 12 \rightarrow \\ &\sqrt{5^2+12^2} = 13 \end{aligned}$$

This implies that a result of an evaluation of a function application may be another function. Thus we may define the one-argument function

$$\text{DOUBLE} \equiv \text{MULT } 2$$

where `MULT` is the two-argument function $\lambda x y . x \cdot y$.

A function may also take another function as an argument. For example, we use the function $\lambda f x . f x x$ to define one-argument functions `SQR` and `DOUBLE`:

$$\text{SQR} \equiv (\lambda f x. f x x) \text{ MULT}$$

$$\text{DOUBLE} \equiv (\lambda f x. f x x) \text{ PLUS}$$

Indeed:

$$\begin{aligned} (\lambda f x. f x x) \text{ PLUS} &\rightarrow \\ \lambda x. \text{ PLUS } x x \end{aligned}$$

Now a bit more formally. The *lambda expressions* (or *lambda terms*) are defined by the following grammar:

$$E ::= V \mid \lambda V. E \mid E_1 E_2$$

where V is any identifier. An expression $\lambda V. E$ is called an *abstraction* and corresponds to a function, where identifier V is an argument and E is called a *body* of abstraction. An expression $E_1 E_2$ is called an *application* of argument E_2 to the function E_1 .

The following abbreviation is accepted as syntactic sugar:

$$\lambda f. \lambda x. f x x \equiv \lambda f x. f x x$$

Application is left associative, that is:

$$((f x) y) z \equiv f x y z$$

A variable in a lambda expression can be either *free* or *bound*. A bound variable is associated with some lambda within the expression. For example, x is bound and f is free in $\lambda x. f x x$.

It is allowed to change name of bound variables in an expression if the renaming does not make a collision of variables. Such a renaming is called *α -conversion*. For example, expressions $\lambda a. b(a c)$ and $\lambda d. b(d c)$ are equivalent.

An application FA is a *redex* if F is some abstraction. In this case, a *β -reduction* corresponds to calling the function F with the argument A . A *β -reduction* of a redex $(\lambda x. E) A$ involves a substitution of all occurrences of x in E by A . *α -conversions* may be needed before *β -reduction* for avoiding collision of variables. For example, the expression $\lambda b. (\lambda a b. a(a b))(\lambda a. b a)$ is reduced as follows (redexes being reduced are underlined):

$$\begin{aligned}
& \lambda b . (\lambda a b . a (a b)) (\lambda a . b a) \xrightarrow{\alpha} \\
& \lambda b . (\lambda c d . c (c d)) (\lambda a . b a) \xrightarrow{\beta} \\
& \lambda b . \lambda d . (\lambda a . b a) ((\lambda a . b a) d) \xrightarrow{\beta} \dots
\end{aligned}$$

The reduction of an expression may be continued as long as the expression contains redexes. A *normal form* is an expression containing no redexes. An order of reduction of redexes is determined by a *reduction strategy*. *Normal order reduction* reduces the leftmost outermost (not contained in any other redex) redex first. *Applicative order reduction* reduces the leftmost innermost (not containing any other redex) redex first. Either of strategies has a weakness. Normal order strategy may cause repeated evaluations:

$$\begin{aligned}
& (\lambda x . \text{PLUS } x x) (\text{FACTORIAL } 4) \xrightarrow{\beta} \\
& \text{PLUS } (\text{FACTORIAL } 4) (\text{FACTORIAL } 4) \xrightarrow{\beta} \dots \xrightarrow{\beta} \\
& \text{PLUS } 24 (\text{FACTORIAL } 4) \xrightarrow{\beta} \dots \xrightarrow{\beta} \\
& \text{PLUS } 24 \text{ } 24 \xrightarrow{\beta} \dots
\end{aligned}$$

Applicative order strategy may cause a long (or even infinite) unnecessary evaluation:

$$\begin{aligned}
& (\lambda x . y) (\text{FACTORIAL } 4) \xrightarrow{\beta} \dots \xrightarrow{\beta} \\
& (\lambda x . y) 24 \xrightarrow{\beta} \\
& y
\end{aligned}$$

Lazy evaluation eliminates these weaknesses. We supplement pure lambda calculus with a *let-expression* $\text{let } x = A \text{ in } E$ and redefine reduction rule as follows:

$$(\lambda x . E) A \xrightarrow{\beta} \text{let } x = A \text{ in } E$$

It allows to not evaluate A if E does not depend on x , or to use a value of A for all occurrences of x in E . A substitution of one occurrence of x by a value of A is called a *dereferencing*. As an example, consider the following reduction:

$$\begin{aligned}
& \text{let } x = \lambda z . z \text{ in } \underline{x} x \xrightarrow{\text{deref}} \\
& \text{let } x = \lambda z . z \text{ in } (\lambda z . z) \underline{x} \xrightarrow{\text{deref}}
\end{aligned}$$

$$\text{let } x = \lambda z. z \text{ in } (\lambda z. z)(\lambda z. z) \xrightarrow{\text{garbage}} (\lambda z. z)(\lambda z. z)$$

Note that a let-expression may be dropped as *garbage* if x no longer appears in E . Lazy evaluation is described in more details in [3]. As an example, we consider the reduction of the expression $(\lambda x. xx)((\lambda yz. yz)(\lambda w. w))$ without going into details. Figure 1 illustrates the reduction in standard notation and in the usual graph form.

Obviously, a core of each lambda expression is a graph structure. Each textual notation representing lambda expressions has a weakness. For instance, using of named variables in standard notation is redundant and causes intermediate α -conversions. Expressions written using de Bruijn indexes [4] are invariant with respect to α -conversion but the same bound variables may be represented by different numbers, which makes the rules of reduction more complicated. In the next section we describe some graphical notations for lambda expressions and determine their benefits and drawbacks.

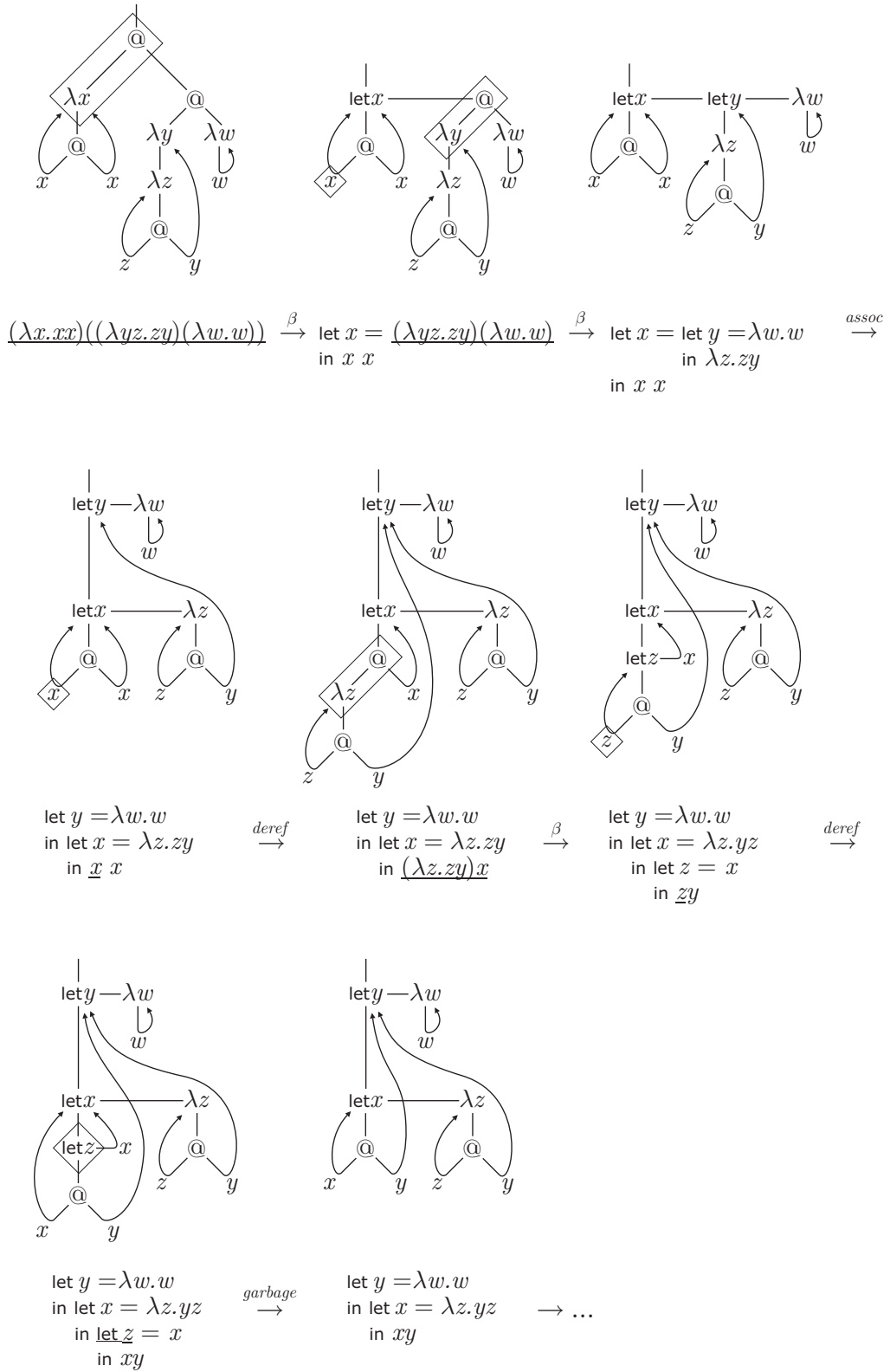


Figure 1. Lazy evaluation.

2. Existing visualizations of lambda calculus

Martin Erwig [5] proposes a general approach to visual languages. He presents a framework based on a rather general notion of *abstract visual syntax* and defines an abstract visual syntax of a visual language notation as a set of directed labeled graphs, where nodes represent visual objects and edges represent relationships between objects. In this section we describe four existing approaches to visualize lambda expressions, compare them in the framework of abstract visual syntax and note their benefits and drawbacks.

First consider the expression $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. Figure 2 shows its structure. We can see that there are three types of nodes (*application node*, *lambda node* and *variable node*) and two types of edges such as *tree edges* (parent-child) and *binding edges* (variable-lambda). Erwig considers such a structure as the abstract syntax graph of a lambda calculus visual notation.

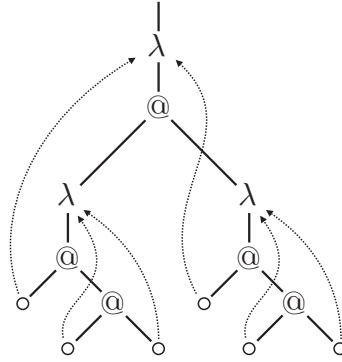
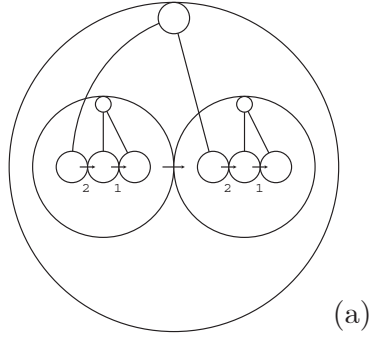
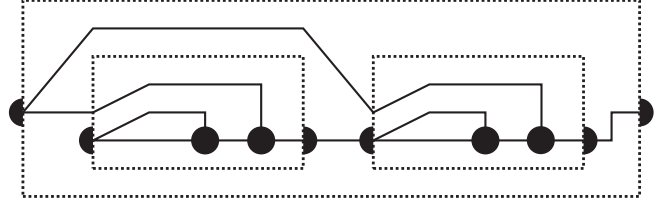


Figure 2. The structure of $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

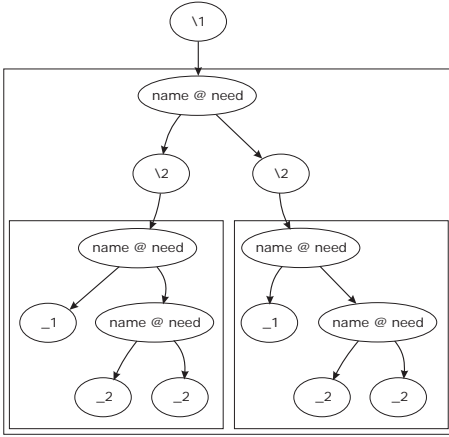
There exist a lot of approaches to visualize lambda expressions. We have chosen the following four as the most mature and interesting: VEX by Wayne Citrin [6], a Graphical Notation by Dave Keenan [7], Lambda Animator by Mike Thyer [8] and a puzzle game Alligator Eggs by Bret Victor [9]. Corresponding representations of the expression above are shown in Figure 3.



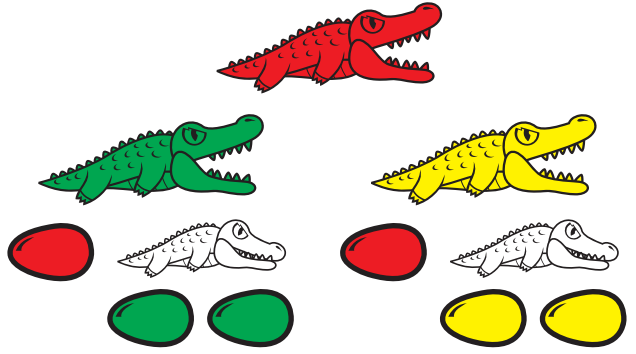
(a)



(b)



(c)



(d)

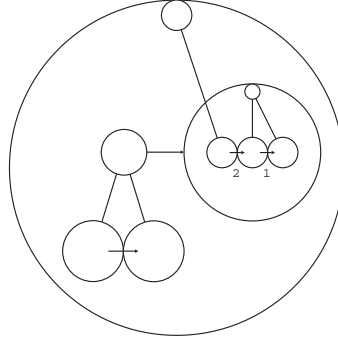
Figure 3. $\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))$ in notations of
(a) Wayne Citrin, (b) Dave Keenan, (c) Mike Thyer and (d) Bret Victor.

2.1. Programming with Visual Expressions (VEX) by Wayne Citrin

This notation represents a variable node as an empty circle. A lambda node is represented as a circle and in addition an internally tangent circle (a parameter of an abstraction). An application node is represented as an arrow between two externally tangent circles (a function circle and an argument circle). Note that the notation does not differentiate application order ($f(xx)$ or $(fx)x$). So application arrows have to be numbered according to priority. Hence the notation is not purely visual.

The notation uses an appropriate nested syntax (a body of abstraction is inside a lambda circle), which allows not to encumber a representation with tree edges. Binding edges are drawn as lines that connect a variable circle and an abstraction parameter circle. Note also that the notation provides a representation

of let-expressions. For example the expression $\lambda f.\text{let } y = \lambda x.f(xx) \text{ in } yy$ is drawn as



2.2. A Graphical Notation for the Lambda Calculus by Dave Keenan

This purely visual notation also uses a nested syntax. However tree edges of application are mixed up with binding edges, which rather encumbers the representation. The most important advantage of this notation is that it allows for smooth animations representing the reduction of an expression. The smoothly animated reduction allows easier following the transformation of an expression. Figure 4 shows the reduction

$$(\lambda xy.x)(\lambda z.z) \xrightarrow{\beta} \lambda yz.z$$

Regrettably, this notation does not provide a representation of let-expressions.

2.3. Lambda Animator by Mike Thyer

The Lambda Animator represents the expressions in tree form. So the tree edges are drawn. Binding edges are represented by numbering the lambda and variable nodes. The notation very nearly reproduces the traditional tree notation of lambda expressions. Only this notation is implemented (in Java) and available online [8], which is the main advantage. The notation also provides a representation of let-expressions. Unfortunately, the “animation” of β -reductions is not smooth and thus it is not easy to follow the transformations.

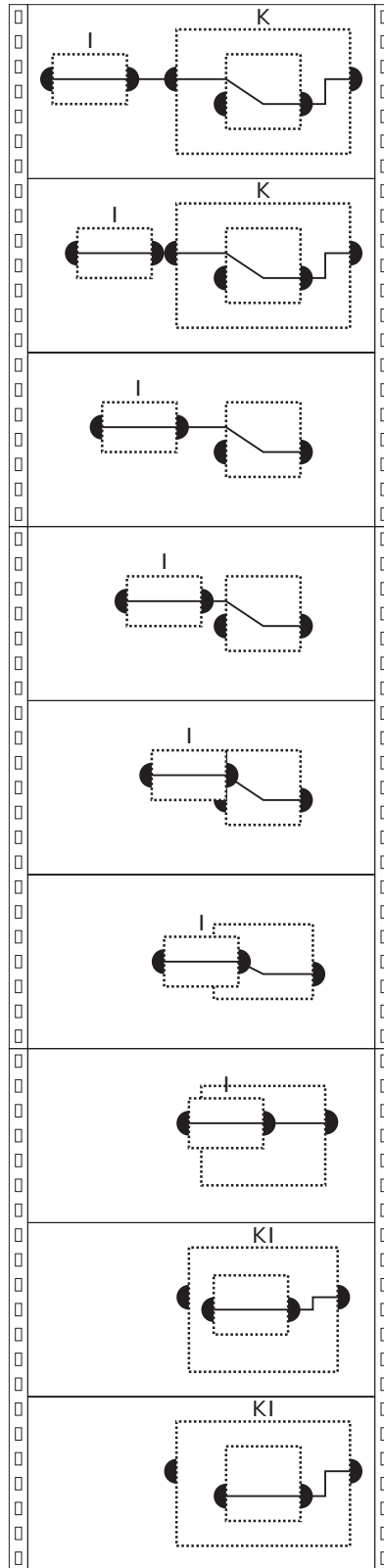


Figure 4. Smoothly animated reduction by Dave Keenan.

2.4. Puzzle game Alligator Eggs by Bret Victor

This is a paper game, which is designed for children. A hungry (colored) alligator represents a lambda node, an egg represents a variable node and an application node is represented as two alligator families next to each other. Old (white) alligators stand for parenthesis and determine the order of applications. This notation allows rather concise representations, which are suited to children. Expressions are represented in tree form but tree edges are not drawn. The felicitous feature of this notation is the using of color: a binding is represented as the same colored egg and alligator, which avoids unnecessary lines and simplifies a representation. β -reduction is denoted by eating and hatching out. Let-expressions are not provided.

2.5. Motivation for a new notation

We saw that none of the considered notations combines all of the advantages determined above (such as the providing of let-expressions, smooth animated reductions, nested and colored syntax etc). It motivated us to develop the bubble notation, which is specified in the next section. As an example, Figure 5 shows the same expression $\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))$ in the bubble notation.

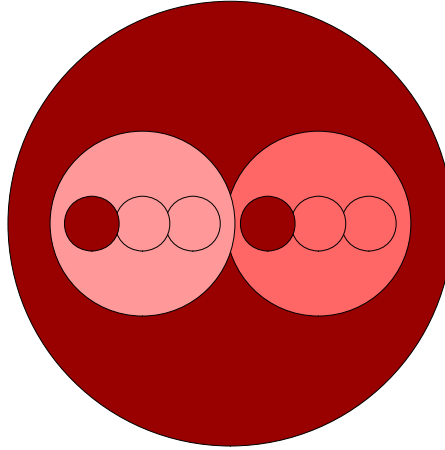


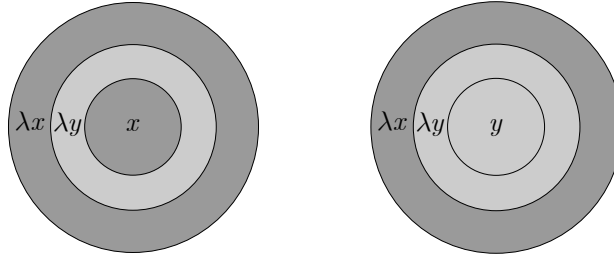
Figure 5. $\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))$ in the bubble notation.

3. Bubble notation

The implemented programming environment Visual Lambda uses the bubble notation for representation of expressions and reductions of lambda calculus. In this section we describe the basic rules of the notation.

The bubble notation represents λ -expressions as two-dimensional structures of rings. These rings are called *bubbles*. Each bubble has a specific color. Bubbles may be grouped together. A bubble may have a group of bubbles inside.

Each bubble represents a variable or an abstraction. So we have two types of bubbles: *variable bubble* and *lambda bubble*. A variable is represented as an empty bubble. A lambda bubble λa of an abstraction $\lambda a.E$ has the bubbles inside, which represent E . Each bubble of variable a in E has the same color as the corresponding lambda bubble. Variable bubbles associated with different lambdas always have different colors. Bubbles of free variables are white. So the expressions $\lambda xy.x$ and $\lambda xy.y$ are drawn as



An application is represented as partial overlapping bubbles. The bubble of function is over the bubble of argument. The expressions fx and $\lambda x.xx$ are represented as

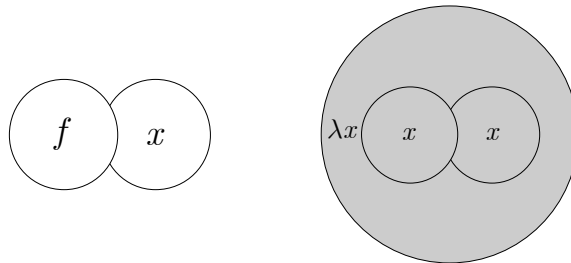


Figure 6 illustrates composition of applications.

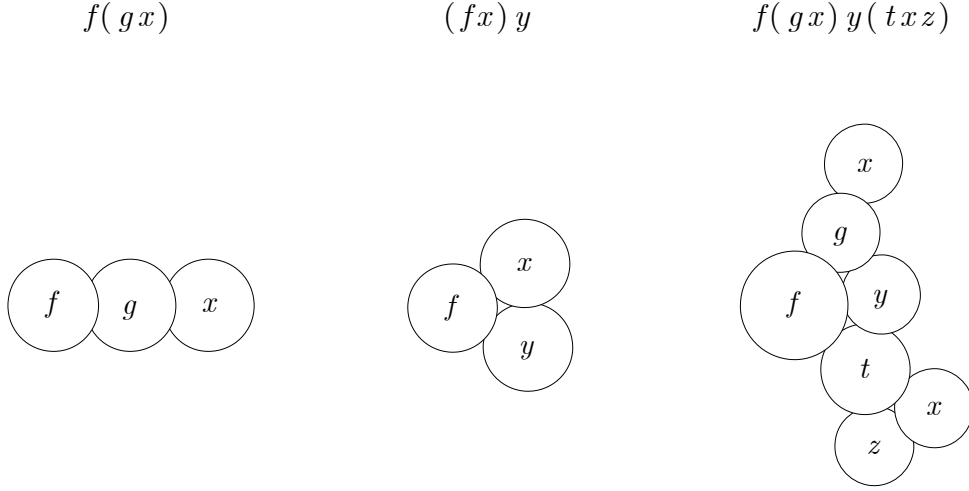
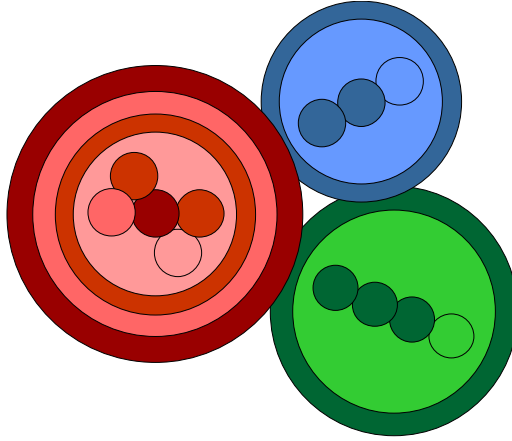


Figure 6. Composition of applications.

The following figure illustrates the expression

$$(\lambda m n f x. n f(m f x)) (\lambda f x. f(f x)) (\lambda f x. f(f(f x))) \equiv_{\text{PLUS } 2 \ 3}$$

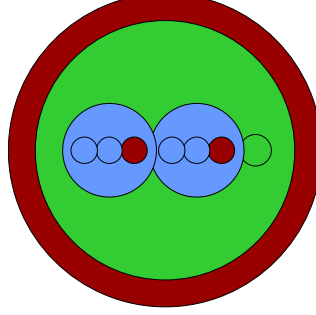


Note that colors of the bubbles in a *closed expression* (an expression without free variables) have the same hue. It makes the recognition of a structure of an expression easier.

The bubble notation also allows representing let-expressions. Consider an expression $\text{let } x = E' \text{ in } E$, where the let-bound variable x appears in E several times. Then all occurrences of x in E are represented by the same colored duplicates of the representation of E' . So the expression

$$\lambda f x. \text{let } h = \lambda g. g(gf) \text{ in } h(hx)$$

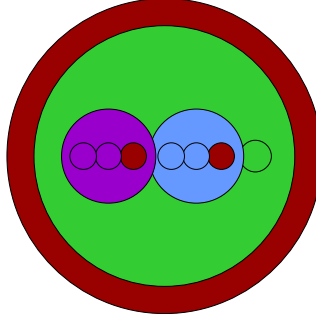
is drawn as



The copy of $\lambda g. g(gf)$ changes its color after dereferencing. So that the expression

$$\lambda f x. \text{let } h = \lambda g. g(gf) \text{ in } (\lambda g. g(gf))(hx)$$

is drawn as



Further extensions of the bubble notation concern animation and interaction. So a β -reduction of a λ -expression is shown as a continuous animation. It allows easier following a modification of the expression. Consider a β -reduction of the redex $(\lambda x. E) E'$. We shall use the metaphor of *eating*. Eating act includes three phases: (1) the *eaten bubbles* (the bubbles of E') are moving under the *eating bubble* (the bubble λx) so that the eating bubble covers them, at the same time the copies of the eaten bubbles are appearing inside the bubbles of the lambda-bound variables (the bubbles x within E); (2) hiding of the eating bubble λx and the associated bubbles x (their transparency grows to 100%); (3) rearranging of

remained bubbles according to resulting expression. Figure 7 shows phases of the reduction of the expression

$$\lambda f. (\lambda g x. g(gx)) (\lambda h y. h(hy)) f.$$

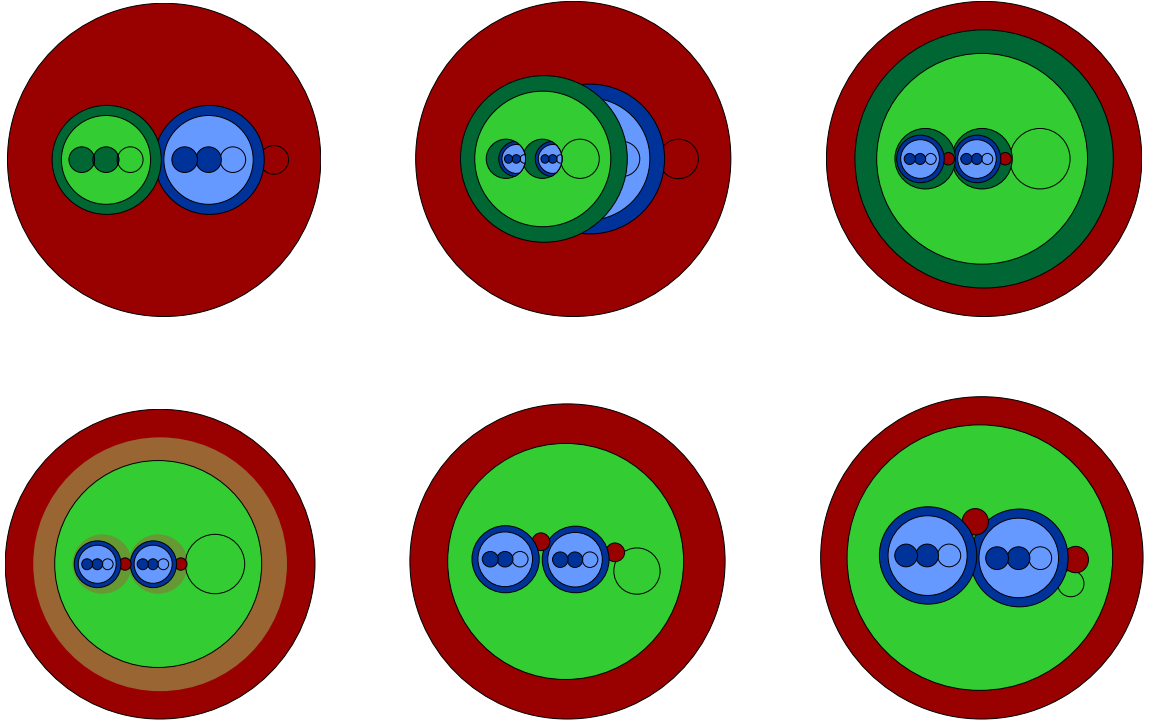
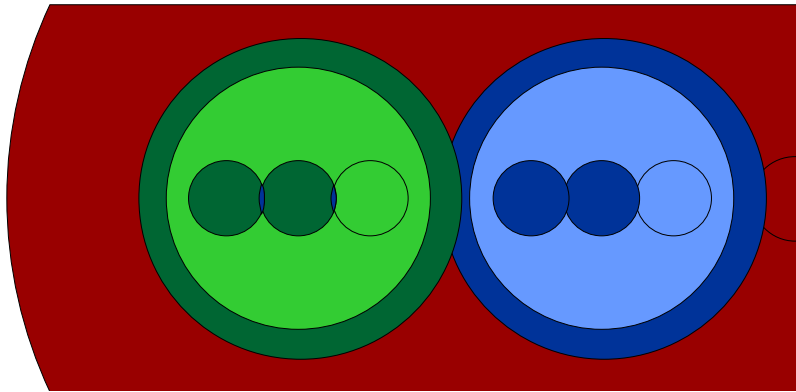


Figure 7. Eating act.

It should be noted that a part of the nearest eaten bubble is seen inside the lambda-bound variable bubbles even before the eating. It signifies that the eating is possible:

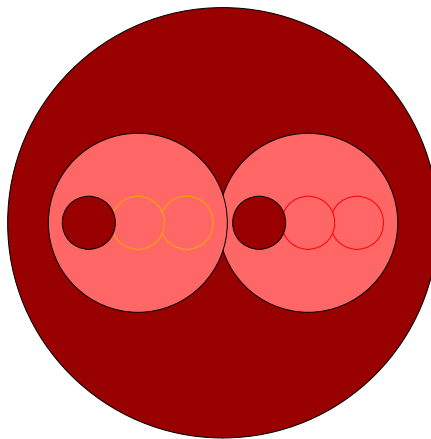


A selection of a sub-expression occurs by a simplest interaction: mouse picking results in the red *highlighting* of some bubbles. A highlighted bubble corresponds to selection of a variable or an abstraction. Multiple bubble highlighting corresponds to selection of an application.

Now note that if we select some sub-expression within the let-bound variable bubble y then the bubbles of the same sub-expressions within other let-bound variable bubbles y get orange highlighting. The expression

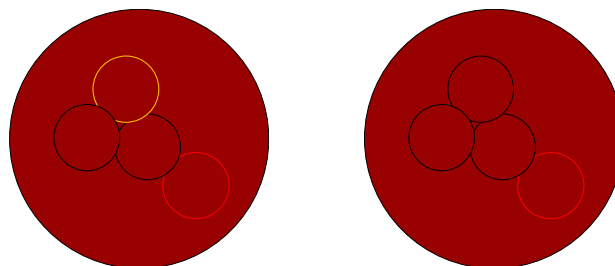
$$\lambda f. \text{let } y = \lambda x. f(xx) \text{ in } yy$$

is drawn below. The application xx within the second let-bound variable y is selected.



The orange highlighting also allows distinguishing the expressions

$$\lambda x. \text{let } y = xx \text{ in } yy \text{ and } \lambda x. xx(xx):$$



4. Interface

The Visual Lambda environment contains a workspace window and a console. Figure 8 shows the environment. Console is provided for textual output and input of lambda expressions. The workspace window allows creation and manipulation of the *figures* by mouse and hotkeys. Each figure represents a lambda expression. Three toolbars (left, right and bottom) are available.

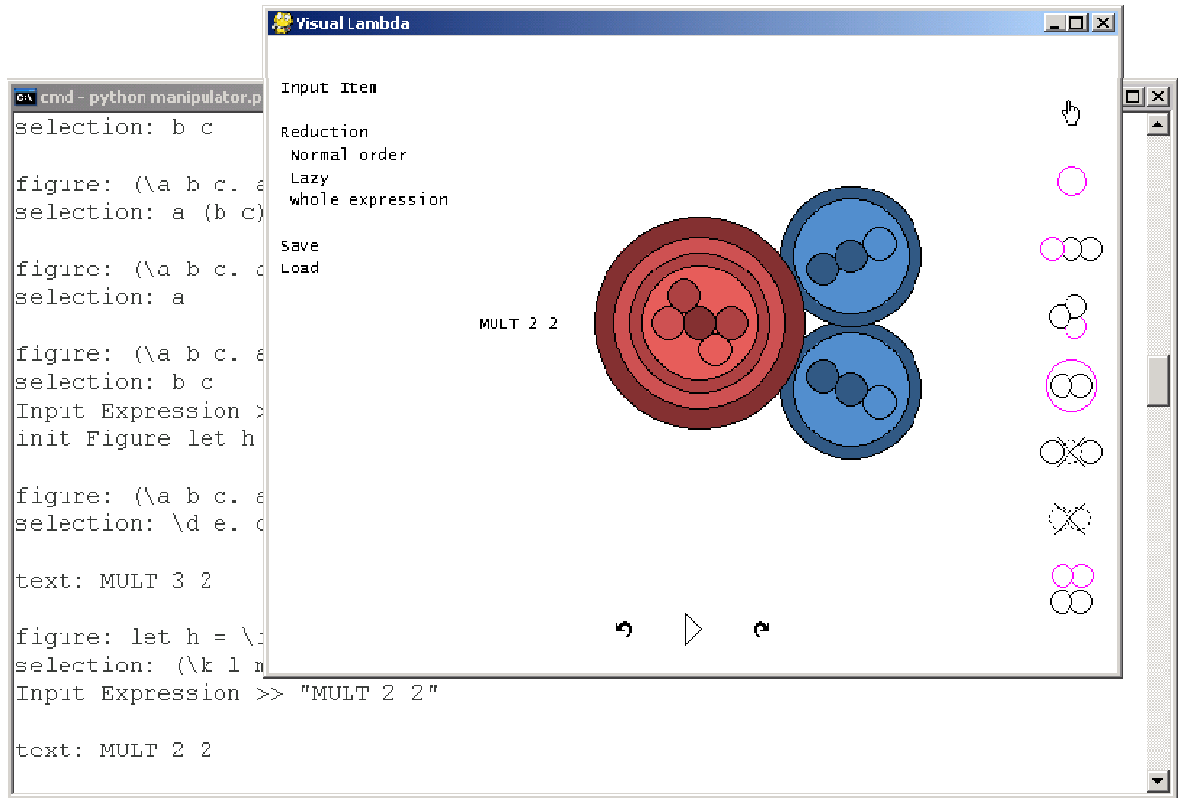


Figure 8. Visual Lambda enviroment.

Figures are movable with mouse. Mouse wheel zooms the workspace. A clicked figure is brought to front and gets selection. An expression of a whole clicked figure and a selected sub-expression are shown in console. The bubbles of selected sub-expression are highlighted with red. If a clicked bubble is a function part of some application, then the whole application gets selection. A repeated click allows selection of a clicked bubble only. The space key expands a selection of a sub-expression.

The right toolbar provides buttons for construction of a figure. Figure 9 describes the buttons of the toolbar and corresponding keyboard shortcuts.









	Quick reduction mode	Q
	Add a new figure (free variable)	V
	Insert an application before the selected bubbles	A, Insert
	Insert an application after the selected bubbles	Ctrl+A, Ctrl+Insert
	Insert a lambda bubble	L, Alt+Insert
	Delete the selected bubbles	Delete
	Delete the selected figure	D
	Duplicate the selected figure	C

Figure 9. Right toolbar.

Right-click on a bubble tries to bind a clicked variable/lambda bubble with a selected lambda/variable bubble.

Dropping a figure to another figure produces an application. More properly, dropping a figure A to a sub-expression B of another figure substitutes B for an application AB . The bubbles of sub-expression B are highlighted with dark red before dropping.

The Input item command on the left toolbar requests for an expression in console and creates a corresponding figure. An exemplary input is

```
let a=2 in MULT a a
```

All the accepted synonyms of lambda expressions are defined in Figure 10. It is possible to add synonyms to the file library.txt in the working directory.

The Play button from the bottom toolbar (or the Enter key) performs a reduction step of a selected figure. A reduction step contains a visible reduction (a β -reduction or a dereferencing) and invisible reductions of let-expressions when required (arranging and garbage collection). Ctrl+Enter starts a nonstop reduction. Undo and Redo buttons (Ctrl+Z and Ctrl+Y) allow returning a figure to one of the previous states.

Combinators	Arithmetic
$I \equiv \lambda x. x$	$ISZERO \equiv \lambda n. n(\lambda x. F) T$
$K \equiv \lambda x y. x$	$SUCC \equiv \lambda n f x. f(n f x)$
$S \equiv \lambda x y z. x z (y z)$	$PLUS \equiv + \equiv \lambda m n f x. n f(m f x)$
$W \equiv \lambda x. x x$	$MULT \equiv * \equiv \lambda m n f. m(n f)$
$Y \equiv \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x))$	$POW \equiv \lambda m n. n m$
$OO \equiv (\lambda x y. y(x x y)) (\lambda x y. y(x x y))$	$PRED \equiv \lambda n f x. n(\lambda g h. h(g f)) (\lambda u. x) I$
	$FACT \equiv Y(\lambda f n. (ISZERO n) 1$ $(MULT(f(PRED n)) n))$
Logic	
$T \equiv \lambda x y. x$	$0 \equiv \lambda f x. x$
$F \equiv \lambda x y. y$	$1 \equiv \lambda f x. f x$
$NOT \equiv \lambda p. p F T$	$2 \equiv \lambda f x. f(f x)$
$AND \equiv \lambda p q. p q F$	$3 \equiv \lambda f x. f(f(f x))$
$OR \equiv \lambda p q. p T q$	etc
$COND \equiv \lambda p x y. p x y$	
Pairs	Lists
$FST \equiv \lambda p. p T$	$NIL \equiv \lambda z. z$
$SND \equiv \lambda p. p F$	$CONS \equiv : \equiv \lambda x y. PAIR F (PAIR x y)$
$PAIR \equiv , \equiv \lambda a b f. f a b$	$NULL \equiv \lambda z. z T$
	$HEAD \equiv \lambda z. FST (SND z)$
	$TAIL \equiv \lambda z. SND (SND z)$

Figure 10. Accepted synonyms for expression input.

The left toolbar shows a current reduction mode, such as a reduction strategy (normal or applicative), a calculus mode (pure or lazy lambda calculus) and the bounds of reduction (within a selection only or the whole figure). The quick reduction mode (the finger cursor) allows selection and then reduction within selection by a single click.

Besides figures, a workspace may contain text labels. Input item command adds a text label if the inputted text is quoted.

A workspace may be saved and loaded from an XML file (Ctrl+S, Ctrl+O). The name of the file is requested in console.

Some basic options of the environment (such as speed of eating, default workspace or the size of text) may be changed in configuration file config.cfg.

5. Implementation

The prototype of Visual Lambda environment is written in the Python programming language (version 2.5) [10] using the PyGame library (version 1.8) [11]. Python allows easy writing, reading and rewriting code. PyGame is the most commonly used Python library for creating interactive graphics programs. Next we describe the main modules and classes of the environment program.

Manipulation of lambda expressions is realized in module `let.py`. Modules `let.py`, `library.py` and `lambdaparser.py` may be used as a console evaluator of lambda expressions. The Library class allows defining the synonyms. The Parser class admits of parsing lambda expressions including let-expressions and synonyms, for example ‘let a=2 in MULT a a’. Module `let.py` allows step-by-step reduction of an expression according to selected reduction strategy. This module contains classes Variable, Abstraction, Let, Application and the base class Expression, which implement corresponding constructions of lambda expressions. Neither the variable nor the abstraction class contain a string attribute for an identifier. The module `refnames.py` is used for representing the identifiers. This module keeps a dictionary {Abstraction: identifier}. If we need to represent some new bound variable, a new pair (Abstraction: identifier) is added into the dictionary. Identifiers are added in order 'a','b',...'z','aa','ab',..

The rest of modules realize the graphical representation and the interface of the environment. We use the class Bubble, which specifies a bubble size and position. Thus we attach a Bubble object to each Variable and Abstraction object. Each variable or abstraction node may correspond to more than one bubble if this node is reached through some let-bound variable. Therefore each node has a dictionary {key: Bubble}, where key is a tuple ($var_1, var_2, var_3, \dots$) of involved let-bound Variable objects. We use an object of class Noke, which contains a pair (Node, key), as a pointer to some bubble of a figure. Figure 11 shows the tree and the figure of the expression $\lambda a. \text{let } b = \lambda c. a(c c) \text{ in } b b$. One of the bubbles c is selected.

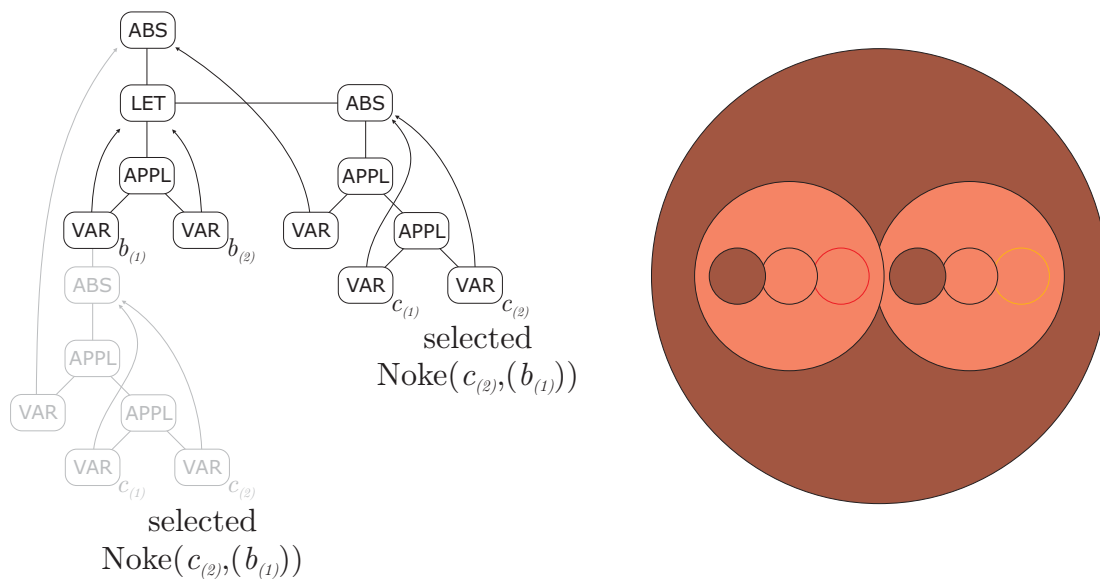


Figure 11. Noke pointer to a bubble.

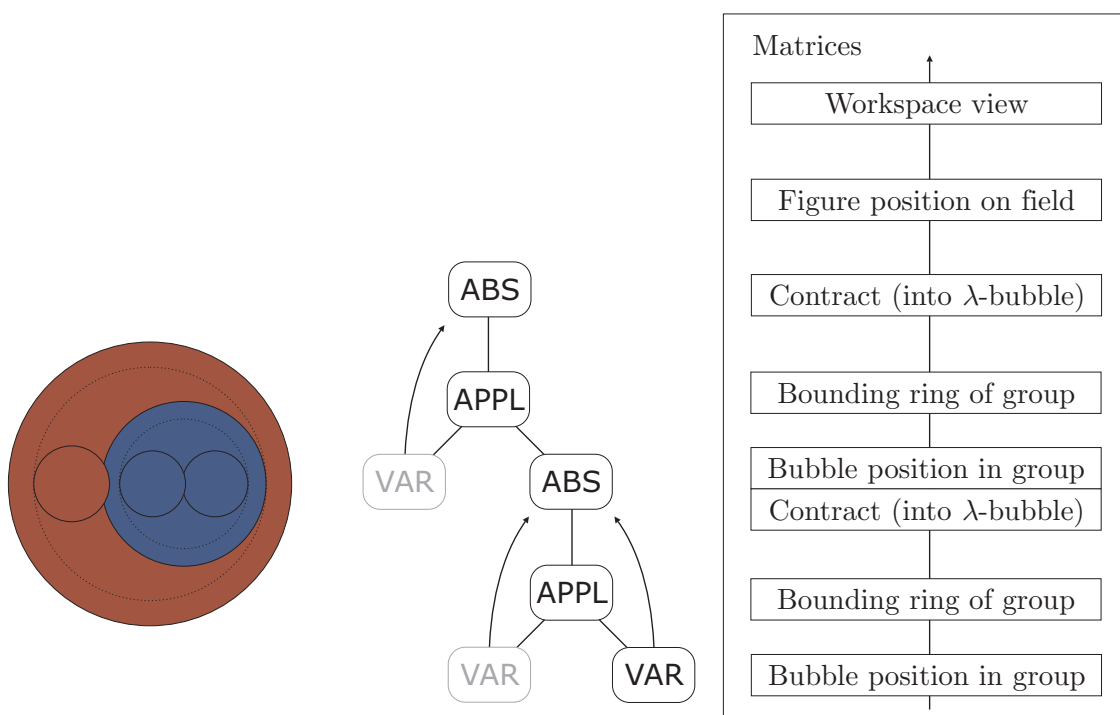


Figure 12. A sequence of transformation matrices.

In case of a variable or an abstraction takes part in some application then the corresponding Bubble object refers to a Group object. Each group of bubbles is built in own space and has a bounding ring as an attribute. The groups are nested by transformation matrices. Figure 12 shows the figure and the tree of the expression $\lambda a. a(\lambda b. b b)$ and the corresponding sequence of transformations used in drawing the variable bubble b .

The radiuses of bubbles in each group are determined iteratively. Each iteration step makes a correction of a bubble radius if the neighbor bubbles do not fit around it. Figure 13 shows a step of building a group $aa(aaaaa)$. The radius of the middle bubble has been enlarged.

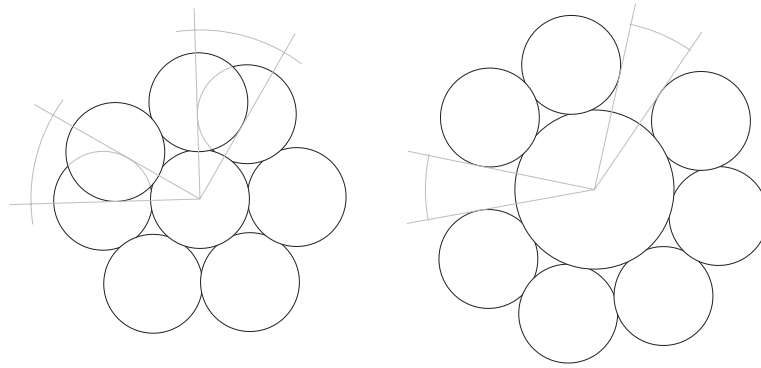


Figure 13. A step of iterative building a group.

Each Figure object has a position on a workspace. A Figure object also keeps a history of an expression (copies of previous expressions without Bubbles) and a ColorSpace object. Similarly to the module `refnames.py`, a ColorSpace object associates each abstraction node of the figure with a representation color. The class ColorSpace uses the HSV color model [12] for grouping colors by hue (colors of bubbles in closed expressions differ in lightness only). Note that the function $\log_2(2n+1)$ is used to determine a hue value for a new group. This function allows quite uniform distribution of any number of groups in color space. Figure 14 shows an arrangement of nine groups in color space. The same function is used for determining a lightness value for a new bubble within a group.

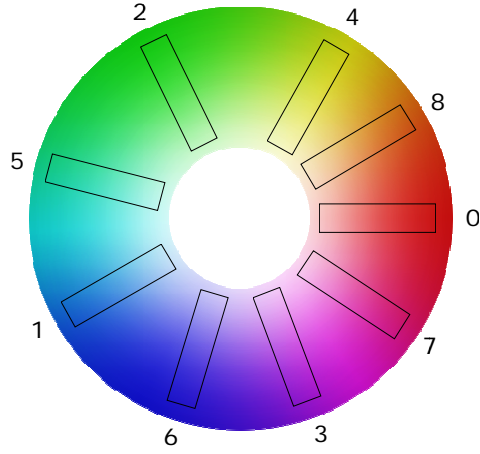


Figure 14. Grouping colors by hue.

Finally we describe how the “eating” works. The variable bubbles a in redex $(\lambda a.E)A$ are drawn as *holes*, through which the bubbles of A are seen. A hole drawing uses a blitting by a circular mask. The first phase of “eating” is a moving the bubbles of A until they are fully seen in the holes. Rebuilding the figure according to reduction (redistributing the bubble groups, recalculation the matrices) occurs at the second phase.

6. Student experiments

We have tested the Visual Lambda environment prototype in some experiments with students. The aim of the experiments was to explore the possibilities of the environment and the bubble notation.

For the first experiment we have chosen a 14-year-old student. The experiment lasted one hour with a short break. First the notions of functional and imperative programming languages and their applications were given for motivation. Lambda calculus was introduced as the basis for functional programming. Further the bubble notation rules were explained. The following terms were using: a bubble, a figure, an eating bubble, a bearing bubble, eaten bubbles, an eating act etc. The standard lambda notation was not mentioned.

Then the basic figures (such as $\lambda a.a$, $\lambda a b.a$, $\lambda a b.b$ etc) and their combinations were considered. The figures $\lambda a b.a$, $\lambda a b.b$ were associated with the *True* and *False* values. The operator *Not* was also considered. At the same time the interface of the environment was explained and some tasks were given, such as to predict a result of some eating, or to construct a copy of some given figure.

There was a task to construct such a figure X that $XA \rightarrow A(AA)$ for each A . The student found the answer $X \equiv \lambda a.a(aa)$ quickly. Then a figure $\lambda a.aAB$ was associated with a pair of A and B . A task to construct such a figure X that $(\lambda a.aAB)X \rightarrow A$ was given. The student was able to find the answer $X \equiv \lambda a b.a$. However, the task to solve $X(\lambda a.aAB) \rightarrow A$ turned out to be too hard ($X \equiv \lambda p.p(\lambda a b.a)$).

For the second experiment we have chosen a graduated in informatics student. The result was also not delightful. The student was able to solve the task $X(\lambda a.aAB) \rightarrow A$, but the task $X(\lambda a.aAB) \rightarrow \lambda a.aBA$ was too difficult.

It is clear that the two experiments are not enough for a valid final conclusion. However it shows that the interface of the environment is not intuitive enough and may prevent from directing attention to a task. Further experiments are needed before conclusions about the bubble notation can be drawn. For example, an experiment in a lecture on lambda calculus might compare conceiving of lambda calculus using the bubble notation and in the usual way. Experiments with modifications of the bubble notation might discover a better variant of the notation.

7. Conclusion

7.1. Summary

This work aimed to develop a visual notation, which represents lambda expressions as clearly as possible, and to implement a visual programming language environment based on this notation, which allows intuitive manipulation of lambda expressions.

Four existing approaches to visualize lambda expressions have been considered during development of the notation. The following features of the considered notations have been accepted as advantageous: (1) nested syntax (which allows representing the tree structures omitting tree edges); (2) using of color (allows representing the bindings without using identifiers, numbers or edges); (3) smoothly animated reductions (helps to follow transformations of expressions); (4) possibility to represent let-expressions and (5) a notation should be easily implementable. The bubble notation proposed by the author combines the features above.

The implemented Visual Lambda environment prototype allows for the construction and manipulation of figures, which represent lambda expressions. The environment has a graphical interface combined with a console. The mouse allows zooming, dragging figures, dropping one figure to another (producing an application), picking a redex for reduction etc. The toolbars allows construction of figures, reducing, returning figures to previous states, changing a reduction strategy, saving and loading workspaces etc. The environment was tested in experiments with students. The experiments had showed that the environment interface is not intuitive enough and needs to be improved.

The Visual Lambda is an open source project and is available under the GPLv3 license [13] for all interested in lambda calculus at address <http://code.google.com/p/visual-lambda/>. A copy of the project sources (as of 27.05.2008) is also available on the attached CD.

7.2. Future work

The Visual Lambda environment permits considerable further improvement and modifications. Let us note some potential features.

Smother construction and manipulation of figures might make it easier to follow transformations. In particular, one could implement smooth transformations

for the undo and redo commands, the smooth appending and removing of bubbles to a figure and the smooth changing of colors by dereferencing.

The possibility to substitute some bubble figures for other objects in the manner of synonyms might simplify some compound figures. As an example, Figure 15.a shows a possible representation of the expression

$$\text{FACT } 3 \equiv Y(\lambda f n. (\text{ISZERO } n) 1 (\text{MULT}(f(\text{PRED } n)) n)) 3$$

It also may motivate to solve the task $X(\lambda a. a A B) \rightarrow A$ described in the previous section if we substitute A and B for some images (Figure 15.b).

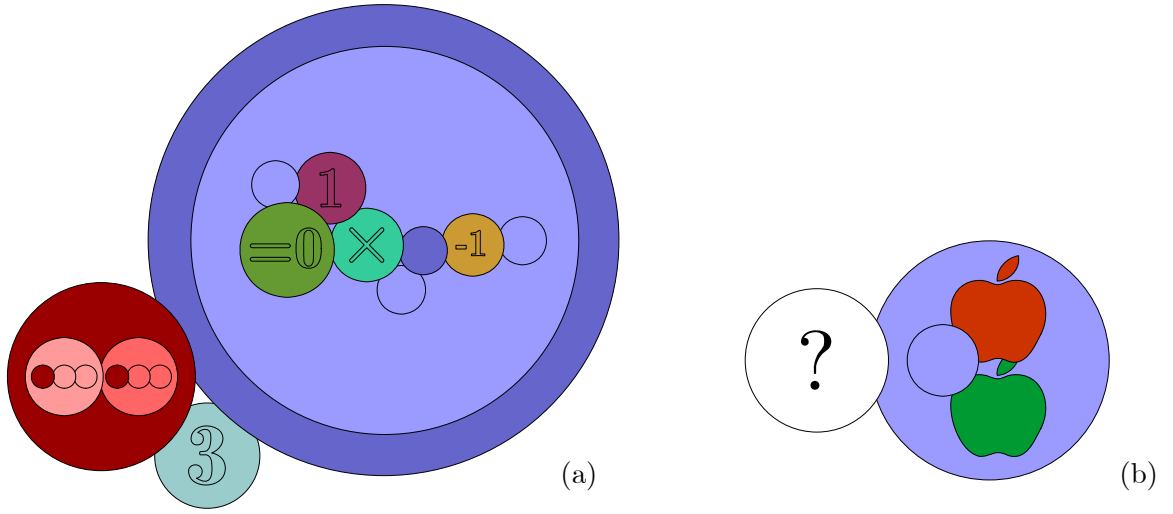
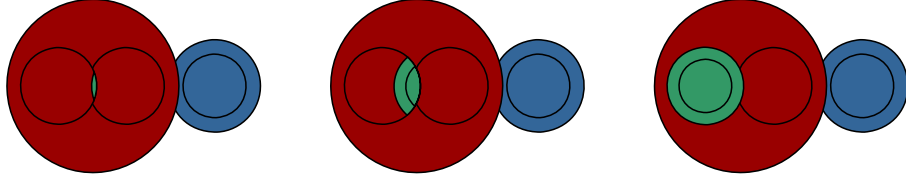


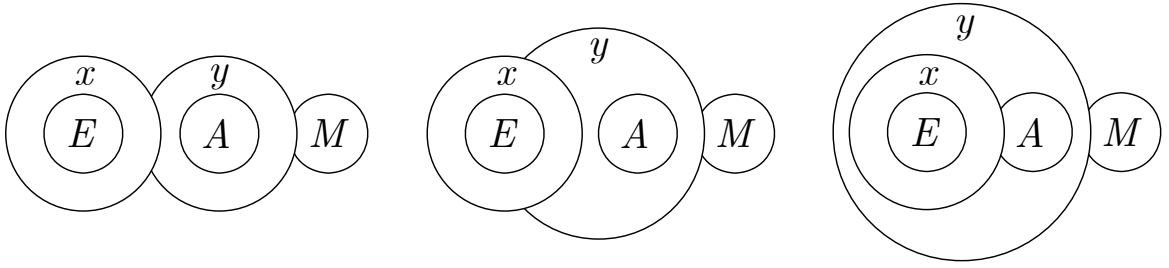
Figure 15. Synonyms in bubble notation.

Following potential features concern the bubble notation. One of the main drawbacks is that the notation does not differentiate free variables (all of them are represented as white bubbles). A more correct notation, for example, might assign different whitish colors to different free variables.

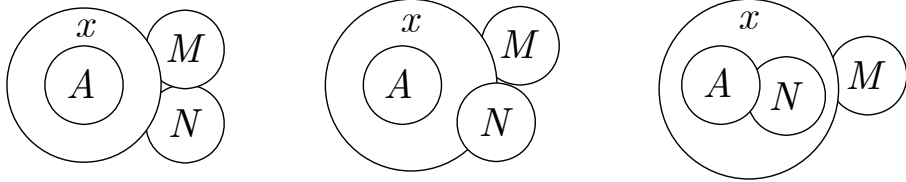
A more proper notation of let-expressions has emerged during writing this thesis. The notation may be implemented in the future and is described below. Each let expression $\text{let } x = A \text{ in } E$ is represented in the same way as a redex $(\lambda x. E) A$. Then a dereferencing is represented as a kind of threading. For example the dereferencing $\text{let } x = \lambda a. a \text{ in } x x \xrightarrow{\text{deref}} \text{let } x = \lambda a. a \text{ in } (\lambda a. a) x$ is represented as:



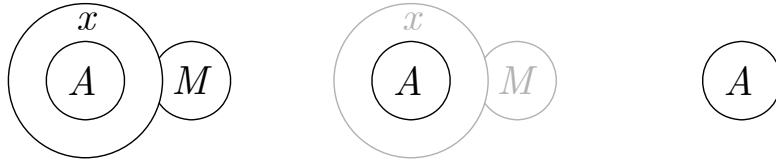
An advantage of this notation is that other reduction rules, such as *assoc*, *lift* and *garbage* [3], become visible. Figure 16 shows the corresponding representations.



$$\text{let } x = \text{let } y = M \text{ in } A \text{ in } E \xrightarrow{\text{assoc}} \text{let } y = M \text{ in let } x = A \text{ in } E$$



$$(\text{let } x = M \text{ in } A) N \xrightarrow{\text{lift}} \text{let } x = M \text{ in } A N$$



$$\text{let } x = M \text{ in } A \xrightarrow{\text{garbage}} A \quad \text{if } x \text{ is not free variable of } A$$

Figure 16. Representations of reductions in the alternative notation.

Visuaalne lambda arvutus

Magistritöö (20 AP)

Viktor Massalõgin

Resümee

Töö eesmärgiks oli disainida visuaalne esitus lambda-arvutusele ja selle alusel realiseerida visuaalse programmeerimiskeele keskkond. Põhieelduseks oli, et lambda-arvutuse reeglite lihtsus lubab arendada visuaalse keskkonna, kus on võimalik manipuleerida lambda-arvutuse termidega intuiitiivsel tasemel. Samal ajal oleks loodud visuaalne programmeerimiskeel Turingi täielik ning võimaldaks määratleda loogilisi tehteid, aritmeetikat, loendeid, rekursiooni ja teisi konstruktsioone. Sellist keskkonda saaks kasutada lambda-arvutuse õpetamiseks, samuti abivahendina lambda-arvutuse uurimisel ning lastele mõeldud loogilise mänguna.

Töö koosneb seitsmest peatükist, mis jagunevad kolme loogilisse ossa. Esimene osa, peatükid 1 ja 2, on referatiivne, kus antakse ülevaade lambda-arvutuse põhimõistetest, tutvustatakse Martin Erwigi ideed visuaalsete programmeerimiskeelte abstraktselt visuaalsest süntaksist ning vaadeldakse nelja olemasolevat lambda-arvutuse visuaalset notatsiooni: Wayne Citrini VEX programmeerimiskeel, Dave Keenani graafiline notatsioon, Mike Thyeri lambda-animaator ja Bret Victori alligaatori munade mäng. Analüüsides olemasolevaid notatsioone selgus, et neist ükski ei rahulda kõiki soovitavaid omadusi.

Töö teine ja ühtlasi põhiosa (peatükid 3-5) kirjeldab väljatöötatud notatsiooni lambda-arvutuse visualiseerimiseks ning annab ülevaate selle alusel realiseeritud visuaalsest programmeerimiskeskonnast. Autori poolt väljatöötatud nn. mulli-notatsioon kasutab lambda-termide esitamiseks värvilisi ringe (mulle). Muutuja on kujutatud tühja ringina (muutuja-mull), abstraktsioon aga ringina (lambda-mull), mille sees on abstraktsiooni kehale vastav kujund. Aplikatsiooni kujutatakse kahe teineteisele asetatuna ringina. Muutuja-mullile ja sellega seotud lambda-mullile omistatakse sama värv. Antud notatsioon on puhtalt visuaalne (ilma literaalsete märgisteta). Notatsioon võimaldab kujutada reduktsiooni pideva animatsiooniga: üks mull sujuvalt „neelab” teisi mulle, samal ajal ilmuvad neelatavate mullide koopiad vastavate seotud muutujate asemele.

Keskkonna prototüüp on realiseeritud programmeerimiskeeles Python kasutades graafilist paketti PyGame, mis lubavad kergesti luua interaktiivseid

graafilisi programme. Keskkond koosneb põhiaknast ja konsoolist. Põhiaknas on hiire abil võimalik konstrueerida lambda-terme ja neid visuaalselt manipuleerida. Tööriistariid lubavad käivitada reduktsioone, muuta redutseerimisstrateegiat ja ennistada eelnevaid olekuid. Terme on võimalik sisestada ka konsoolilt. Samuti on võimalik defineerida termide sünonüüme, salvestada ja laadida liidesealasid jt.

Töö viimases osas (peatükid 6 ja 7) on raporteeritud keskkonna testimisel kahe õppuriga saadud tulemustest ja vaadeldud keskkonna võimalike edasiarendusi. Õpputega eksperimenteeriti lambda-arvutuse õppimist antud keskkonna baasil. Kahjuks ei saa eksperimentide tulemusi väga õnnestunuteks pidada. Mingil määral räägib see sellest, et keskkonna kasutajaliides ei ole piisavalt intuitiivne. Samas on selge, et rohkem põhjendatud järeldused vajavad edasisi uuringuid. Töös ongi pakutud ka mõningaid lahendusi, mis võimaldaks notatsiooni muuta natuke intuitiivsemaks ning kasutajaliidest mugavamaks. Keskkonna projekt on avatud lähtekoodiga ja on kättesaadav veebist aadressil <http://code.google.com/p/visual-lambda/>.

References

- [1] Marat Boshernitsan, Michael S. Downes. *Visual Programming Languages: A Survey*. EECS Department, University of California, Berkeley. Technical Report No. UCB/CSD-04-1368. December 2004.
<http://www.eecs.berkeley.edu/Pubs/TechRpts/2004/CSD-04-1368.pdf>
(27.05.2008).
- [2] Raúl Rojas. A Tutorial Introduction to the Lambda Calculus. FU Berlin, WS-96/98.
- [3] Zena M. Ariola, Matthias Felleisen. *The Call-By-Need lambda Calculus*. Journal of Functional Programming, Vol. 7, Issue 3, 1997, pp 265-301. (doi:10.1017/S0956796897002724).
- [4] De Bruijn index.
http://en.wikipedia.org/wiki/De_Bruijn_index (27.05.2008).
- [5] Martin Erwig. *Abstract Syntax and Semantics of Visual Languages*. Journal of Visual Languages and Computing, Vol. 9, Issue 5, October 1998, pp 461-483. (doi:10.1006/jvlc.1998.0098).
- [6] Wayne Citrin, Richard Hall, Benjamin Zorn. *Programming with visual expressions*. Proceedings of the 11th International IEEE Symposium on Visual Languages, 1995, pp 294-301 (doi:10.1109/VL.1995.520822).
- [7] David C Keenan. *To Dissect a Mockingbird: A Graphical Notation for the Lambda Calculus with Animated Reduction*, 2001.
<http://users.bigpond.net.au/d.keenan/Lambda/> (27.05.2008).
- [8] Mike Thyer. *Lambda Animator*.
<http://thyer.name/lambda-animator/> (27.05.2008).
- [9] Bret Victor. *Alligator Eggs! A puzzle game*.
<http://worrydream.com/AlligatorEggs/> (27.05.2008).
- [10] Python Programming Language.
<http://www.python.org/> (27.05.2008).
- [11] Pygame - Python game development.
<http://www.pygame.org/> (27.05.2008).
- [12] HSL and HSV. http://en.wikipedia.org/wiki/HSL_and_HSV (27.05.2008).
- [13] The GNU General Public License.
<http://www.gnu.org/licenses/gpl.html> (27.05.2008).

Appendix

The attached CD contains the readme.txt file and the following high-level directories:

/source	Visual Lambda environment source code (as of 27.05.2008).
/compiled	Visual Lambda environment compiled with py2exe extension.
/installers	Python 2.5 and PyGame 1.8 installers.