vλ: An Interactive λ-Calculus Tool

Doug Zongker
CSE 505, Autumn 1996

December 11, 1996

"Computers are better than humans at doing these things."

– Gary Leavens, CSE 505 lecture

1 Introduction

The λ-calculus is a useful model of computation because of its power and its close relationship to functional programming languages. Working with it, though, can be a tedious process because of its austerity—a lot of information is carried in one long, heavily parenthesized string.

The goal of this project is to produce a graphical-interface tool which automates the tiresome bookkeeping portion of working with λ-expressions while giving the user useful feedback and allowing all the power and flexibility of the λ-calculus.

This report describes the "visual λ" tool (**vλ**) that has resulted. Section 2 consists of a short user manual. Section 3 discusses some interesting aspects of the implementation. The terms and definitions used in the implementation correspond to those in Hindley and Seldon [HS86].

2 Interface

Figure 1 shows an instance of the main evaluation window. It is composed of three sections: the top part displays the current λ-expression, there is an input line for entering new expressions, and there are controls for the window at the bottom. In this figure, a number of reductions have taken place. The bottommost expression is the current expression.

2.1 Entering a λ-expression

Because most keyboards and X fonts do not include the λ character, **vλ** substitutes a backslash in both input and output. Variables consist of single lowercase letters. Whitespace is unimportant, except for the entering of macro names (discussed below). The following are all valid entries:

Figure 1: The evaluation window.

```
ab
a b
\a.b
(\x.y)((\x.xx)(\x.xx))
```

Expressions with nested abstractions such as `\x.(\y.(\z.w))` can be abbreviated as `\xyz.w`. **vλ** also has a macro facility for entering commonly used subexpressions. Macro names begin with a capital letter, followed by zero or more letters or numbers. (If a macro is followed immediately by a variable name, they must be separated by whitespace or the variable will be parsed as part of the macro name.) A number of predefined macros are loaded from the file `library`, and additional macros can be defined interactively.

Pressing **Enter** or clicking the **Parse** button in the window will parse the expression and display it as the current expression.

2.2 Viewing λ-expressions

If an expression is not grayed out, then some information about it is available by pointing to parts of it:

1. Pointing at a parenthesis causes the subexpression enclosed by that parenthesis to be highlighted in purple.

2. Pointing at a variable causes all instances of that variable that are bound together to be highlighted in red—pointing to a free variable instance highlights only the one pointed to, while pointing to a bound instance also highlights all the instances that are bound by the same abstraction.

3. If an abstraction is reducible, then pointing to its "λ" (the backslash) will cause the variable that would be substituted to be highlighted in blue, and the "argument" of the reduction to be highlighted in green.

Normally, all expressions except the current expression are grayed out, so these features are not available. An expression can be "reactivated," however: to the left of each expression is a number, its position in the window. Holding down mouse button 1 on this number brings up a small popup menu (Figure 2):

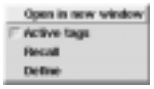

Figure 2: Expression popup window.

The **Active tags** checkbutton in this menu can be used to control the graying-out of expressions other than the current one. The **Open in new window** command causes a new evaluation window to be created with that expression as the current expression. The **Recall** command places the text of that expression in the input field, so it can be edited and entered as a new expression. The **Define** command brings up a dialog box allowing that expression to be defined as a new macro, and optionally saved in the `library` file.

The **display options** button in the evaluation window pops up a menu of checkbutton options:

wrap text Controls whether long expressions are wrapped or truncated in the window.

full parentheses If checked, expressions will be displayed fully parenthesized.

show subscripts If checked, each variable will be printed with a subscript (an underscore followed by a number), so that each unique variable gets a different subscript (free instances of a variable will all be numbered differently, as will each occurrence of a variable bound in an abstraction).

The last three options control how expressions which contain macros are displayed. Every subexpression that derives from a macro can be displayed as the macro name, or as the expanded definition. If none are checked, then no macros are expanded (every subexpression is displayed as its name).

expand all Expressions are printed with macros fully expanded.

expand normal Expressions are expanded as necessary to display the normal-order redex, if there is one.

expand applicative Expressions are expanded as necessary to display the applicative-order redex, if there is one.

2.3 Performing reductions

β-reduction of the current expression in a window can be done by pointing to any reducible lambda and clicking. The result of the reduction becomes the new current expression.

At the bottom of the window are two buttons: **normal** and **applicative** for performing the normal-order and applicative-order reductions respectively. (Moving the mouse into either of these buttons causes the appropriate redex to be highlighted in the window, if it is visible.) These buttons are grayed out if the current expression has no redexes.

α-conversion can be performed by clicking on the variable that you want to rename. A dialog box will pop up, prompting you for a replacement variable name. Converting a free variable will never cause it to become bound. For example, in the expression $\lambda x.xa$, renaming a to x will not change the meaning of the expression. Turning on display of subscripts will show that it is represented internally as $\lambda x_1.x_1x_0$.

2.4 Predefined macros and fun things to do

The macros **If**, **True**, and **False** are defined as in Paul Hudak's lecture, so that `If True x y` gives `x` and `If False x y` gives `y`. Using these, **Or**, **And**, and **Not** are defined, so Boolean expressions like

```
Or (Not True) (And True (Not False))
```

can be evaluated. This is one area in which normal order evaluation can be more efficient than applicative order, since normal order effectively produces short-circuit evaluation.

Hudak's representation of the integers, and the addition function, are defined as **Zero** through **Four** and **Add**. The twice function is defined as **Twice**, so `Twice f x` reduces to `f(fx)`. Using **vλ** we can verify that

```
Twice Twice Twice (Add One) Zero
```

really does reduce to (the λ representation of) sixteen. (It takes 140 reductions using normal order, or 69 using applicative order.)

Some combinators (**S**, **K**, **B**, **I**, **C**, and **Y**) are defined as well.

2.5 Miscellany

There are a few other buttons in the evaluation window:

clear Clears all expressions from the current window.

spawn Opens a new, empty evaluation window.

close Closes the evaluation window. Closing the last open window causes the application to exit.

quit Closes all application windows and exits.

3 Implementation

vλ is implemented in C and Tcl/Tk. All of the manipulation of λ-expressions is done by the C code, while the user interface is handled in Tcl/Tk. The Tcl/Tk code refers to expressions via handles returned from the C code. The project consists of roughly 1500 lines of C (including the input to the Bison parser generator and the Flex lexer generator) and 600 lines of Tcl/Tk.

3.1 Representation of λ-expressions

λ-expressions are represented internally as a binary tree with an associated linked list of variables, as in Figure 3. Leaf nodes of the tree represent variable. There are two types of internal nodes: abstractions and applications. An abstraction is an expression of the form $\lambda x.y$; an application represents the form (xy). A redex is thus an application whose left side is an abstraction.

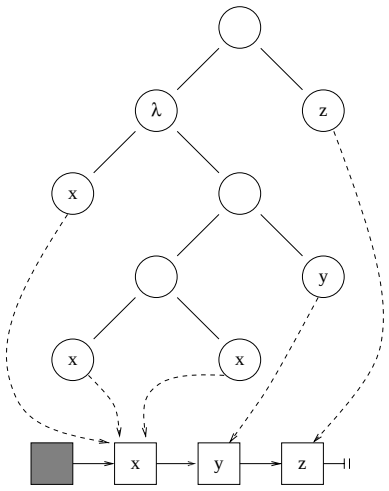

Figure 3: Internal representation of $(\lambda x.xxy)z$.

These structures are built from a user's text input by a Bison/Flex parser. When read, variables are distinguished only by their names, so, for instance, all instances of x refer to the same variable record. This is incorrect for expressions like $(\lambda x.x)x$, where the x outside the parentheses is free and the xs inside are bound. A variable separation process must take place to distinguish free and bound instances.

The procedure works like this:

> Initialize an empty global stack
>
> Begin a recursive depth-first search at the root:
>
> > If I am a leaf (variable) node, then search up the stack for the first instance of me. If there are no instances, then I am a free variable,

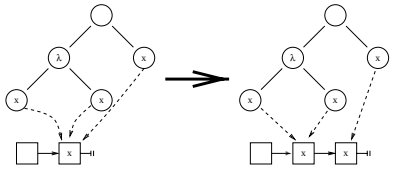

Figure 4: Effects of the variable separation process, on the expression $(\lambda x.x)x$.

> so separate me (duplicate the record in the linked list, and point me to the new copy). Otherwise, I am bound to the top instance on the stack.
>
> If I am an abstraction node, then separate my bound variable (left child) and push a pointer to that instance on the stack, then recurse down the right side (the body of the lambda). When that returns, pop the instance I added to the stack.
>
> If I am an application node, just recurse down the left and right subtrees.

When this routine is finished, the linked list can be traversed to assign a unique subscript number to each variable. This is purely for the benefit of the user; the program internally distinguishes variables by the addresses of the records in the linked list.

3.2 β-reduction

The principal operation that is performed on λ-expressions is β-reduction. Suppose that p is the root of a subtree that is chosen for reduction. For p to be reducible, p must be an application node, and p's left child must be an abstraction node.

If we let $l(\cdot)$ and $r(\cdot)$ represent the left and right children of a node, respectively, then the result of the reduction will be a copy of $r(l(p))$, with every instance of the variable $l(l(p))$ replaced by a copy of $r(p)$. The new subtree replaces the subtree rooted at p in the expression. Since the argument of the reduction may contain abstractions of its own, though, the whole expression must be put through the variable separation process again, in order to separate the variables in the various copies of $r(p)$. The reduction process is illustrated in Figure 5.

References

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* Number 1 in London Mathematical Society Student Texts. Cambridge University Press, 1986.

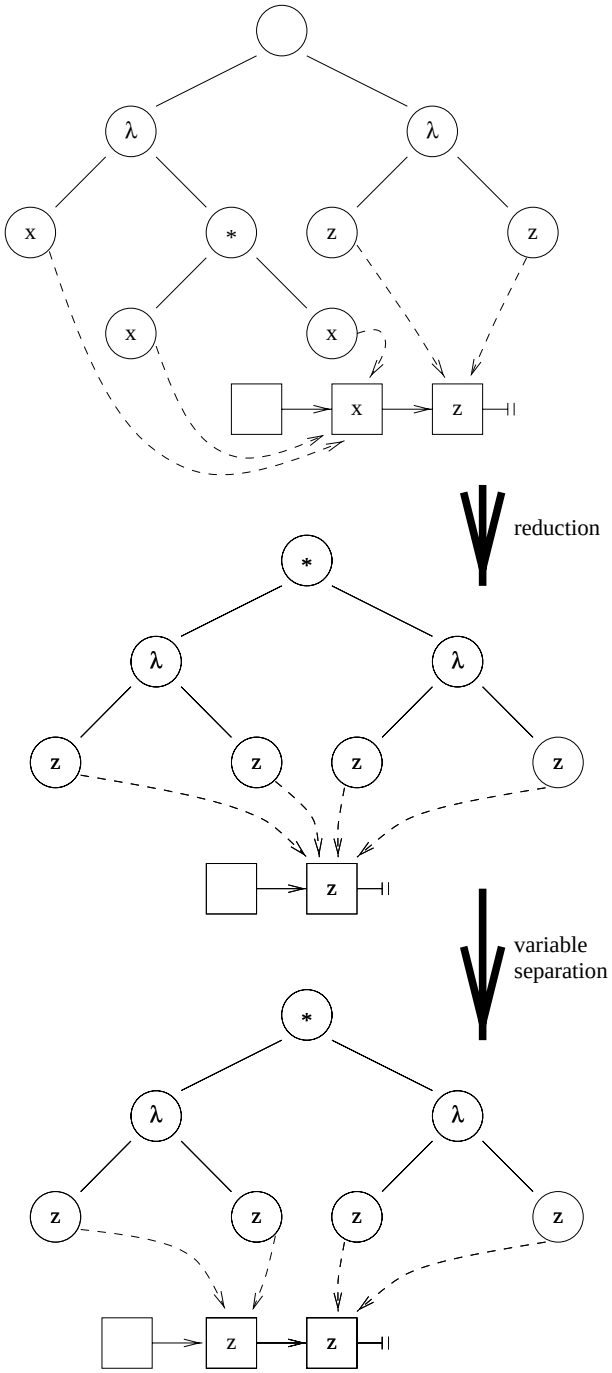

Figure 5: Reduction of expression $(\lambda x.xx)(\lambda z.z)$ to $(\lambda z.z)(\lambda z.z)$. The body of the abstraction is marked with an asterisk.