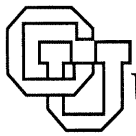
A VISUAL LAMBDA CALCULUS

Wayne Citrin, Richard Hall, Benjamin Zorn

CU-CS-757-95



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Visual Lambda Calculus

Wayne Citrin, Richard Hall, Benjamin Zorn
Department of Electrical and Computer Engineering / Computer Science
University of Colorado
Boulder, Colorado, USA
{citrin,rickhall,zorn}@cs.colorado.edu

ABSTRACT

The lambda calculus is a formal symbolic term rewrite system that has been used for many years both as a mechanism for defining the semantics of programming languages, and as the basis for functional programming languages. In this paper, we propose a visual representation of lambda expressions. Our representation, Visual Expressions (VEX) has several advantages over traditional textual lambda calculus. VEX can be used in teaching the concepts of lambda calculus as a replacement for or augmentation to the teaching traditional textual rewrite rules. Many semantic issues in lambda calculus that are confusing to students, including substitution, free variables, and binding, become apparent and explicit in VEX. Finally, VEX provides a framework in which both the static and dynamic (partially simplified) representation of lambda expressions can be understood.

1.0 INTRODUCTION

The lambda calculus is a widely used and powerful notation for describing computable functions. It serves as the basis of functional languages, and is also the basis of denotational semantics. In order to accomplish these tasks, lambda calculus provides a set of seemingly simple textual rewrite rules.

Although the rules seem to be simple, in fact they are not. This deceptive simplicity has led to errors even among experts. For example, it has been claimed that "most formulations of the rule for substitution which were published, even by the ablest logicians, before 1940, were demonstrably incorrect" [8].

It has been our experience that this difficulty is most apparent in teaching beginners. Not only does the notion of substitution and lambda capture confound students, as it does experts, but notions of higher-order functions, currying, abstraction, environments and free variables, and least fixpoint recursion also present difficulties.

Part of the problem resides in the syntax: nested lambda expressions break up expressions and separate related items, and the uninitiated student is unable to use such useful visual cues as adjacency to decipher the expressions.

Another problems lie in the abstractness of the material, and in the existence of certain features required to give the expression meaning that are only implicitly referred to, such as the notion of an environment that contains bindings of free variables.

In order to address these problems, we introduce VEX (Visual EXpressions), a completely visual representation of the lambda calculus. By completely visual, we do not mean that the notation contains no text (text is suitable for labels and constant values), but only that the semantics of the language is based on a purely graphical set of transformation rules – no knowledge of the underlying textual language is necessary to understand the semantics. VEX has been developed in the context of VIPR (Visual Imperative PRogramming language) [5, 6], a completely visual imperative programming language, in order to provide a representation for expressions and functions in that framework, but we believe that VEX has value outside that framework, particularly in the teaching of functional concepts.

This paper is organized as follows. In the second (next) section, we present a motivating example comparing use of a complex lambda expression (the Y combinator) with a simpler but equivalent VEX expression. In the third section, we build our visual lambda calculus, first by specifying the syntax, then by providing graphical equivalents for the substitution rule and by showing how graphical equivalents of the three rewrite rules (α, β, and η) may be specified using the graphical substitution rule, and finally by showing how the three rewrite rules may be represented by very simple and intuitive graphical transformation rules that make no explicit reference to the substitution rule. We then, in the fourth section, provide a formal specification of the operational semantics of VEX, employing a notation originally used to specify the semantics of VIPR and defined in [5]. In the fifth section, we discuss some additional issues, including the treatment of recursion, and the use of certain syntactic extensions (particularly, *let* expressions), and associated concepts such as environments and closures. In the sixth section, we discuss related work in graphical functional languages, completely visual languages, and the teaching of functional programming languages. Finally, we draw conclusions and report on the current status of the work.

2.0 MOTIVATING EXAMPLE

The *Y combinator*, defined by

$$Y \equiv \lambda f.(\lambda x.\, f(x\,x))(\lambda x.\, f(x\,x))$$

is a construct used, among other purposes, in order to prove results involving fixpoints and recursion. We employ it in our opening example in order to contrast the surprising and non-intuitive behavior of the textual version with the more intuitive behavior of the graphical version.

Textually, we wish to show that, for any lambda expression e, $(Y\,e) = e(Y\,e)$; that is, $(Y\,e)$ defines a fixpoint on e[1].We can do this by expanding $(Y\,e)$ as follows:

---

[1]In this example, and in the treatment of lambda calculus in the next section, we rely on the survey by Hudak [11]. The reader is referred to this survey and its excellent and extensive bibliography for further information on the lambda calculus and its application in functional programming.

$$
\begin{aligned}
(Y\,e) \quad &= (\lambda x.\, e(x\,x))(\lambda x.\, e(x\,x)) \\
(1) \qquad &= e\big((\lambda x.\, e(x\,x))(\lambda x.\, e(x\,x))\big) \\
&= e(Y\,e)
\end{aligned}
$$

The step denoted by the second line above is the difficult one, with its use of a functional argument, and the functional value substituted for the identifier x. We will attempt to do better in our graphical representation.

The VEX representation of the Y combinator is given in figure 1. A few features should be noted. First, an expression (abstraction or not) is represented by a closed figure. Parameters are represented by closed figures that are tangent to, and inside, another closed figure. Thus, the circles labeled f and x represent parameters. Items that would be identifiers in the textual lambda calculus are also represented by circles in VEX. Each of the circles in the two small clusters of three circles represents an identifier. One can determine that they are identifiers because they are connected by undirected edges to a labeled circle, representing the definition of a parameter (in the cases in figure 1) or a free variable (not in figure 1, but described in section 3.0). Application of functions is represented by closed figures tangent to, and external to, each other. A small arrow indicates which figure represents the applied function and which represents the argument (the arrow points to the argument).

One should note that VEX expressions consist of a small number of graphical elements (closed figures, undirected edges, arrows, and labels) used in multiple ways. As we will see in subsequent sections, however, the context in which a graphical element is used is sufficient to unambiguously determine its meaning.

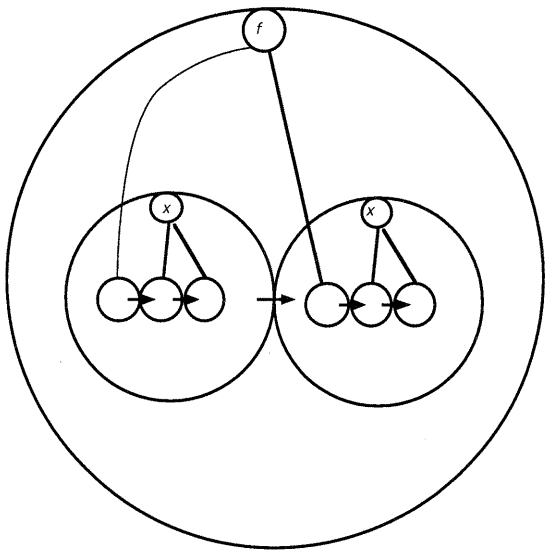


Figure 1. VEX representation of the Y combinator

The basic principle of the VEX expression evaluation is that the graphical expressions are elaborated until no further computation is possible. The resulting graphical expression is the value of the original expression. Evaluation usually continues from the outside inward, so that element in the interior of a figure (or some modification of that interior) is the result. Thus, when applying the expression of figure 1 to the value e, we can intuit that the eventual result will resemble one of the clusters of three elements, where the first (leftmost) element will be the argument (in this case, e), and the remaining elements will be copies of the initial configuration $(Y\,e)$. We will see this in the step-by-step evaluation below.

Figures 2 through 4 show the VEX equivalent of the evaluation given in equation (1) above. Figure 2 shows the Y combinator applied to the argument e. Figure 3 shows the results of this application, substituting the argument for all identifiers bound to the parameter f. The result of the application in figure 3 involves substitution of the functional argument represented by the righthand figure for the parameter x in the lefthand figure. The result, in figure 4, is the function e being applied to exactly the figure in figure 3. Since figure 3, as well as figures 2 and 4, represent $(Y\,e)$, figure 4 must also represent $e(Y\,e)$, and we have a fixpoint.[1]

The VEX representation has made explicit the notion of binding and substitution in a way that is much more straightforward than that of the textual lambda calculus. The distinction between the two different but identically named bound identifiers x is also made explicit. Finally, the notion of functional arguments and higher-order functions is also made explicit. These issues are all frequently confusing to novices. This example should give an idea of how VEX simplifies understanding of the lambda calculus, and the next section, which describes the graphical versions of the various substitution and rewrite rules will further illustrate the intuitive qualities of VEX.

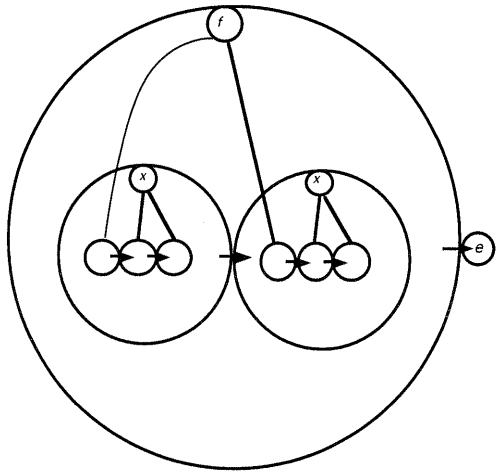


Figure 2. VEX version of $(Y\,e)$

[1] A QuickTime animation of this evaluation may be found at `http://soglio.colorado.edu/Web/vex.mov`.

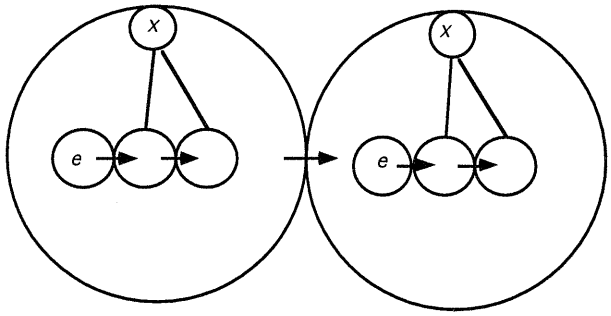


Figure 3. VEX version of $(\lambda x.e(x\,x))(\lambda x.e(x\,x))$

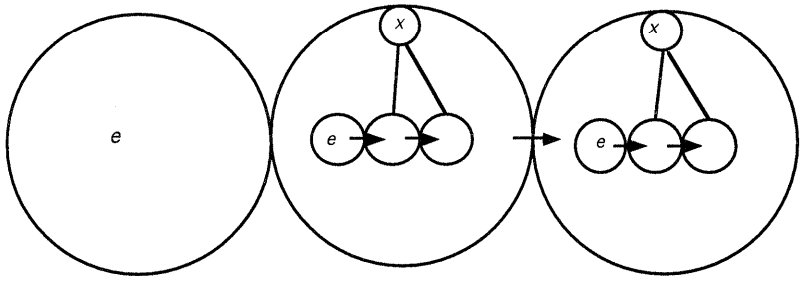


Figure 4. VEX version of $e(\lambda x.e(x\,x))(\lambda x.e(x\,x)) = e(Y\,e)$

3.0 A VISUAL LAMBDA CALCULUS

In this section, we build a graphical version of the lambda calculus employing simple graphical transformation rules instead of textual rewrite rules. We confine ourselves to the *pure untyped lambda calculus*. Additions to this lambda calculus will be discussed in section 5. Our treatment of the textual calculus is the one given in [11].

This section will describe the graphical transformation rules informally. A selection of formal rule specifications is given in section 4, and a more complete description appears in [7].

Syntax

VEX models the pure untyped lambda calculus. We assume that expressions consist of identifiers and other expressions, and that abstractions may only have a single parameter.

In VEX, identifiers and lambda expressions are both represented by sets of closed figures. Identifiers are recognized by the fact that they are connected to a labeled *root node* by an undirected edge. Compound (non-identifier) expressions have an internal structure representing their component subexpressions. Thus, in figure 5, ring 2 represents an identifier that is an instance of the identifier x (due to the fact that it is connected to the root node x – labeled 1), and ring 3 is a compound expression with one subexpression (labeled 5). (Note that the numeric labels on figure 5 are not part of the expression but are solely for purposes of explanation.)

Function application is represented by two expressions externally tangent to each other. An ordering on the two expressions is imposed by an arrow at the tangent point. The expression at the tail of the arrow represents the function being applied, and the expression at the head represents the argument. In figure 5, circles 2 and 3 represent a function application, where circle 3 and its contents represents the applied function, and circle 2 represents the argument.

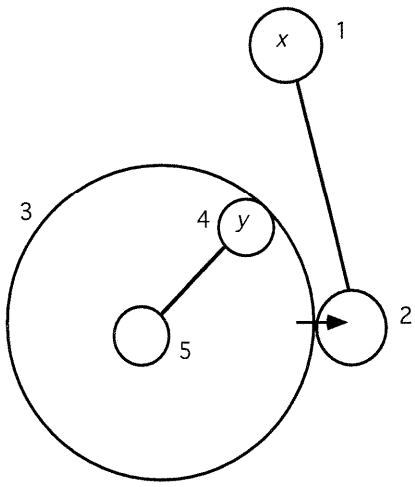


Figure 5. Syntax of a VEX expression

Abstraction is represented by an expression circle containing an identifier root node internally tangent to it. The root node represents the parameter, and the contents of the expression circle represents the abstraction body. Thus, in figure 5, circle 3 and its contents represent an abstraction, where circle 4 is the parameter and circle 5 is the body.

Figure 5, therefore, represents the expression $(\lambda y.\, y)\, x$.

Free and bound identifiers

As can be seen in figure 5, VEX expressions make explicit the distinction between free and bound identifiers. An identifier is free in a given expression if its root node is not internally tangent to a ring in that expression. For example, in figure 5, the identifier labeled 2 (and consequentially all identifiers connected to node 1) are free in the VEX expression represented by the whole diagram. On the other hand, the identifier denoted by ring 5 (y) is bound in the expression represented by ring 3 and its contents, although it is free in the expression represented by ring 5.

One can see that certain seemingly anomalous situations can occur, such as that in figure 6, in which the identifier x seems to be both free and bound in the expression labeled 1. However, textual names in VEX have no semantic value; the actual "name" of an identifier ring is the root node to which it is connected. Thus, the "name" of the identifier represented by ring 3 is the root node labeled 2 in the diagram, while the

root node whose label is 4 is the "name" of the identifier represented by ring 5. The textual label on the root node is simply a comment meant to enhance readability, although we currently require root nodes to have names in order to distinguish them from other types of rings. In this case, it is simply a coincidence that both identifiers are named x. There is no difference in power between VEX and conventional textual lambda calculus in this regard: any expression like that given in figure 6 can be written textually by renaming one of the variables (e.g., $\lambda x.x'(x)$). It is recommended, however, that names be chosen to avoid such confusion.

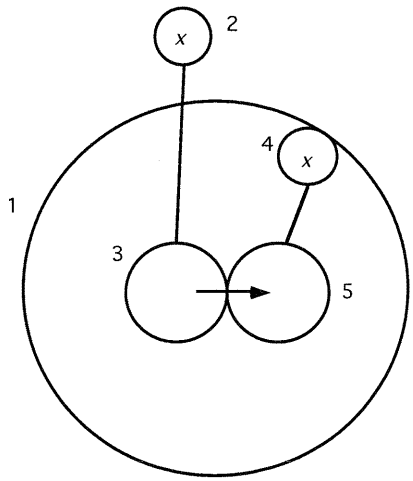
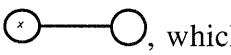
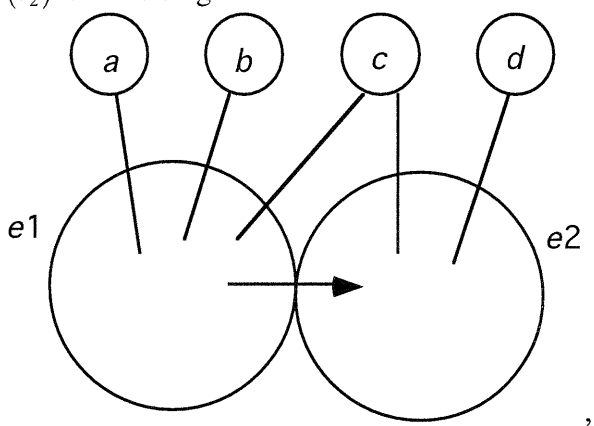


Figure 6. Free and bound identifiers - representation of $\lambda x.x'(x)$

It should be noted that this graphical definition exactly reflects the traditional definition of free variables in textual lambda calculus. If we define $fv(e)$ as the set of free variables in a given expression e, then for a single-identifier expression x, $fv(x)=\{x\}$, and likewise  , which represents the expression x, clearly indicates that the set of free variables in that expression is $\{x\}$.

Similarly, $fv(e_1\ e_2)=fv(e_1)\cup fv(e_2)$ and the diagram

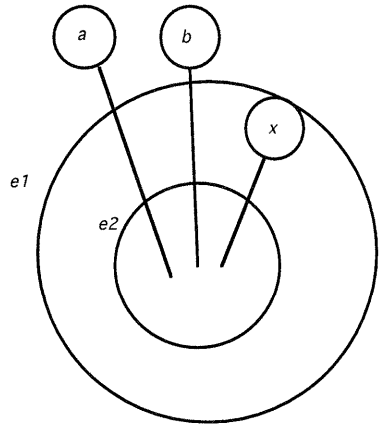


,

where the unconnected lines are a notation that indicates that the undirected edges each connect to some ring in the interior of rings e_1 and e_2, suggests the same relationship, since, inductively, if a, b, and c are

the free identifiers in e_1, and c and d are the free identifiers in e_2, then clearly a, b, c, and d are the free variables in $(e_1\ e_2)$.

Finally, concerning abstractions, $fv(\lambda x.e) = fv(e) - \{x\}$. Similarly, it is clear from



that if a, b, and x are free in e_2, then only a and b are free in e_1.

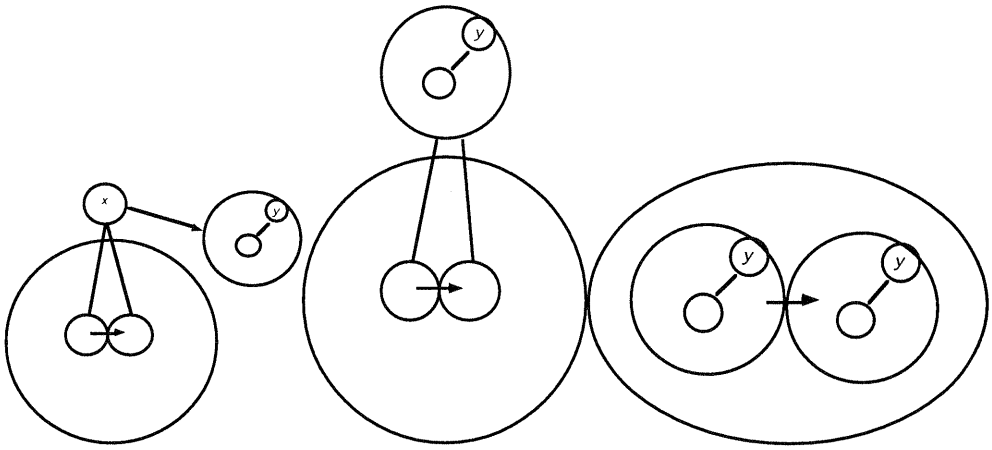
Substitution rules

The rules for substitution are one of the chief stumbling blocks of the textual lambda calculus, tripping up both experts and novices. Although they seem intuitive, they turned out to be quite difficult to formulate. The chief problem has to do with renaming of free variables that are "imported" into a context in which variables of the same name are bound. The different cases can be complex, and result at least in part from the problem of naming in textual lambda calculus. When naming does not exist (or rather, when it simply becomes a matter of identifying a particular root node), and when name clashes cannot occur, the process is greatly simplified. Thus, while the lambda calculus substitution rule, as described in [11], involves an extensive and complex case analysis, in VEX, where there is no possibility of name clashes, and variables carry their "freeness" explicitly, the substitution rule is much simpler.

To substitute for a variable in VEX, a "substitution arrow" is used. The arrow originates in the root node of the identifier being substituted for, and ends at the expression being substituted. Thus, as in figure 7, if we wish to substitute the given expression (equivalent to $\lambda y.y$) for x, we run an arrow from x's root node to the figure for the expression. The first step is to substitute the thing pointed to for the thing at the tail of the arrow, as in figure 8. In the next step, all identifiers connected to the former root node are substituted for, and the original expression and the undirected edges disappear (figure 9).

Variables may also be substituted for other variables, as is shown in figure 10.

The same simple two-step process holds for all substitutions, thus greatly simplifying the substitution rule by eliminating consideration of various combinations of free and bound variables and by avoiding concerns about renaming.



7 8 9

Figure 7. Substitution - initial configuration
Figure 8. Substitution - first step
Figure 9. Substitution - second step

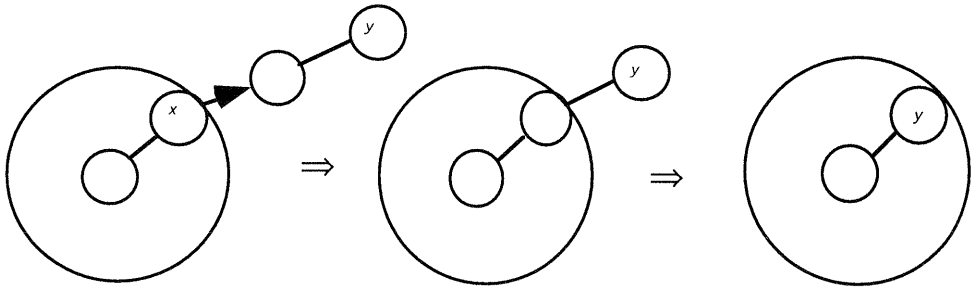


Figure 10. Substitution of one variable for another

Rewrite rules

The main part of lambda expression evaluation is the application of the various rewrite rules, commonly known as α-conversion, β-reduction, and η-reduction. We will consider each of these, first providing the direct graphical equivalent of the textual rule (generally employing the substitution rule), and then, if possible, providing a simple rule without substitution.

The first rule, α-conversion, also known as renaming, is specified textually as

$$\lambda x_i.e \Leftrightarrow \lambda x_j.\left[x_j/x_i\right]e, \text{ where } x_j \notin fv(e).$$

In other words, one may rename any bound variable with any other name, as long as it doesn't conflict with a name already used for a free variable in the function body.

The VEX equivalent is given in figure 10 above. Note that variable "freeness" is graphically explicit, so we need only worry about using a free variable to the extent that the new identifier not employ a root node already free in the expression. Examination of figure 10 suggests that, in VEX, it is sufficient simply to change the label on a root node representing a bound variable; no explicit substitution need be performed. Thus, the α-conversion rule may be applied as in figure 10 without the intermediate step. This

simplification is clearly in the spirit of the renaming conversion; the substitution operation in the textual version is merely an artifact resulting from the textual process of renaming.

β-reduction is a symbolic formulation of function application, and is specified textually as

$$(\lambda x.e_1)e_2 \Rightarrow [e_2/x]e_1.$$

In other words, application involves substituting the argument for all instances of the parameter. In VEX, the substitution rule may be explicitly employed to perform the substitution, as in figure 11, or the intermediate steps may be skipped, and the values propagated to their destinations, as was done in the example in figures 2 through 4.

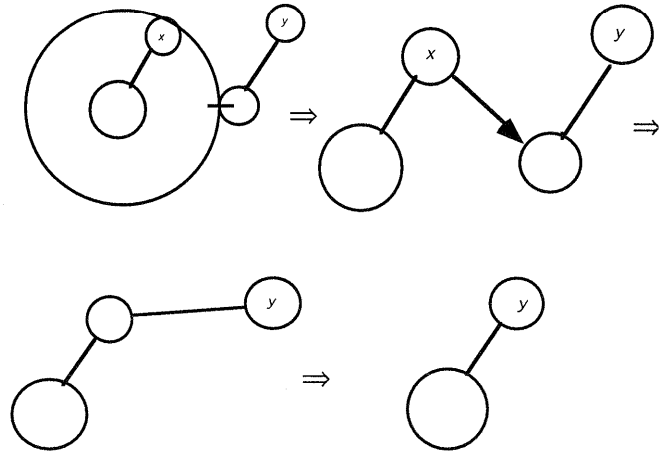


Figure 11. Application

The final rule, η-reduction, relates an abstraction to its underlying expression body in certain cases:

$$\lambda x.(e\,x) \Rightarrow e, \text{ if } x \notin fv(e).$$

Again, in VEX we detect whether or not x is free in e by looking for connecting links from x's root node into e. Figure 12 suggests that VEX's equivalent of η-reduction simply involves collapsing the connection between the parameter and the argument and eliminating the pair.

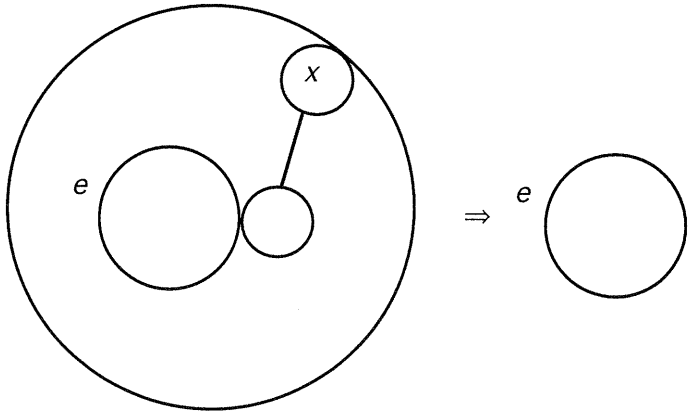


Figure 12. η-reduction

As we can see, the graphical versions of the rewrite rules provided by VEX at the very least lend insight to the mechanisms and rationale of the more arcane textual rewrite rules, and at best provide a simple, intuitive framework for understanding lambda calculus that requires no knowledge of the underlying textual language.

4.0 FORMAL SPECIFICATION OF VEX

Space does not permit a complete discussion of the formal specification of VEX, or of the formalism used to generate that specification. The complete VEX specification will appear in [7], and an extensive discussion of the underlying formalism appears in [5]. Here we will give some of the flavor of the formal specification by giving the specification of the two-step substitution rule.

The specification formalism for the graphical transformation rules is a two-dimensional system based on Milner-style operational semantics [16]. Each rule has a "before" part that must match the current expression configuration to have an effect. If a "before" part matches, the configuration is altered to conform to the corresponding "after" part. A configuration may match more than one rule, in which case one must be chosen. Where this leads to ambiguity, it will be the same ambiguity that would occur in reducing a lambda expression.

Figure 13 shows the graphical transformation rule for the first step of substitution. (For technical reasons, the complete rule is actually a bit more complex than this – the rule given here shows the manipulations on all items that actually participate in the substitution. See [5] for a more complete discussion of this.) In figure 13, *R, R′, R″*, and *Name* are metavariables. *R, R′, R″* represent collections of closed figures, labels, and any arrows or undirected edges connecting them. *N*ame stands for a label on a root node. Any closed figures or edges must exist in the diagram. Dotted edges refer to zero or more copies of the arrows.

In figure 13, the root node labeled *Name* is replaced by the collection of items *R′*. If *R′* refers to any other items (denoted by *R″*), those connections will be carried along, too. *R* represents to the things that refer to *Name*. Note that there must be at least one edge into some circle in *R* from *Name* (represented by the solid edge), although there may be more (represented by the dotted edge).

The second step of the substitution rule is given in two parts by figures 14 and 15. It shows the newly substituted expression being substituted into all occurrences. The first part, in figure 14, shows that if there are two or more occurrences of an identifier, one of them should be substituted into, and the connection to that occurrence eliminated. In the second part (figure 15), where there is exactly one occurrence, the occurrence is substituted for, and both the connection and the root node should be eliminated. All other transformation rules are of a similar nature.

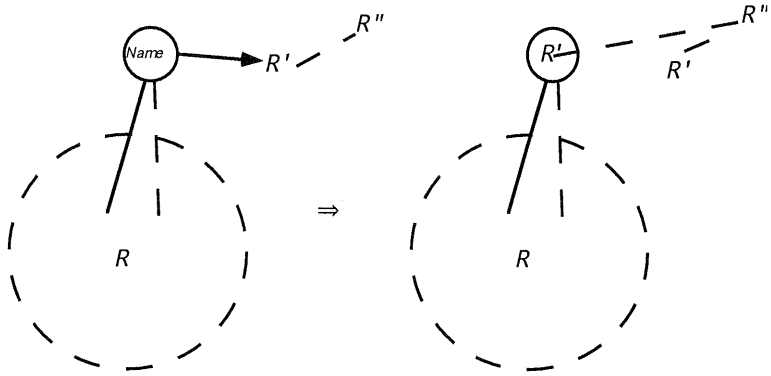


Figure 13. Specification of first step of substitution rule

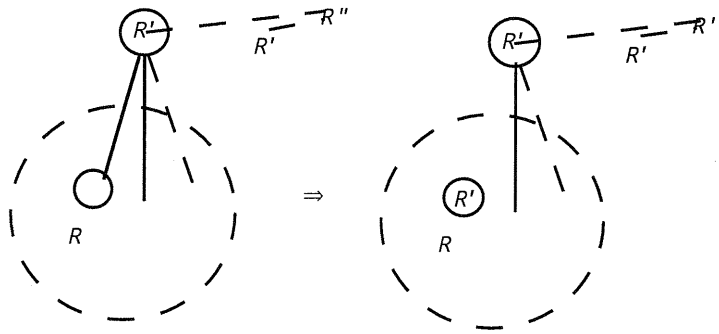


Figure 14. Second step of substitution rule (part I)

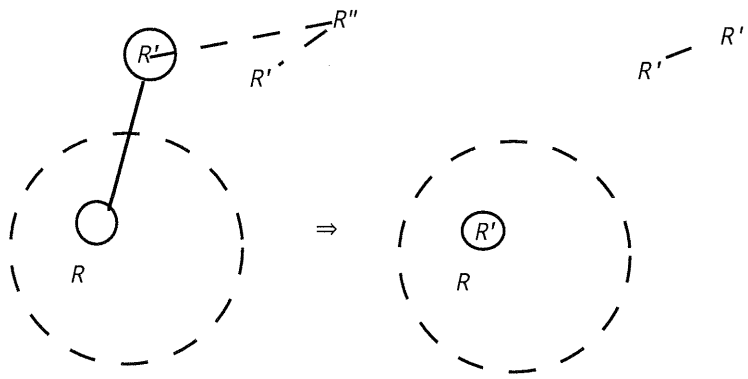


Figure 15. Second step of substitution rule (part II)

5.0 ADDITIONAL ISSUES

In order to make the lambda calculus (and functional programming languages based on it) more usable, certain extensions are often added, including constants, tuple constructors and selectors, conditional operators, primitive functions, and local definitions. None of these extensions increase the power of the calculus, but they increase readability and writability of expressions. Similarly, VEX has equivalent extensions to enhance readability and writability while maintaining the expressiveness of the notation. In addition, both the lambda calculus and VEX must address the notion of recursive definitions.

5.1 Constants and primitives

Although primitive constants and primitive operators are not necessary for the expressiveness of lambda calculus, such entities are generally considered useful and are added to the language. Similarly, we add equivalent operators to VEX.

Constants in VEX are represented by closed figures containing the constants' values. Figure 16 shows sample VEX constants (0, 1, true, and false), and an example of function application involving constants.

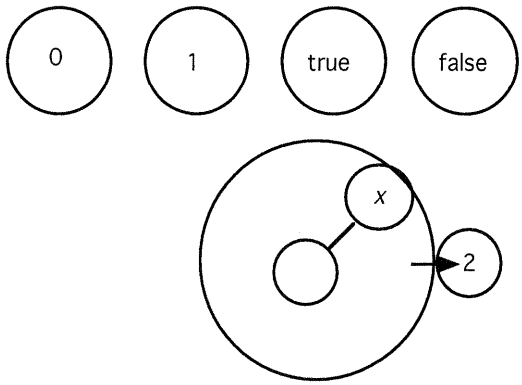


Figure 16. VEX constants

In order to aggregate information in lambda calculus, ordered tuple constructs are used. VEX provides the equivalent in the form of the *selector* construct. Selectors model generalized tuples that may be indexed by values from any domain; that is, for any index set I, and for a set of sets A_i each corresponding to an element i in I:

$$\prod_{i \in I} A_i = \left\{ f : \mathrm{I} \rightarrow \bigcup_{i \in I} A_i \middle| f(i) \in A_i, \forall i \in I \right\}.$$

In other words, tuples are represented as functions, and each tuple comes with its own built-in selector operations.

A selector construct is drawn as a function abstraction containing a set of closed figures each representing one possible index value. Each of the internal closed figures is labeled with the corresponding index value and its content is an expression representing its value. Figure 17 shows a selector construct representing the tuple (3,4) and the application of the selection operation $(3,4)\downarrow 1$ to obtain the first element of the tuple.

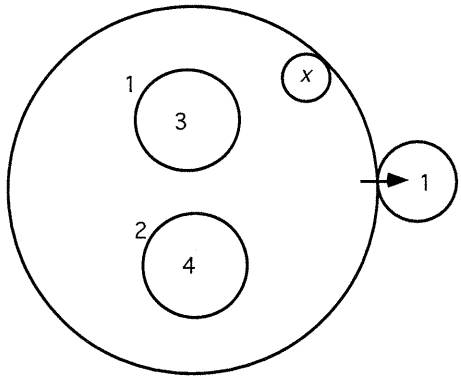
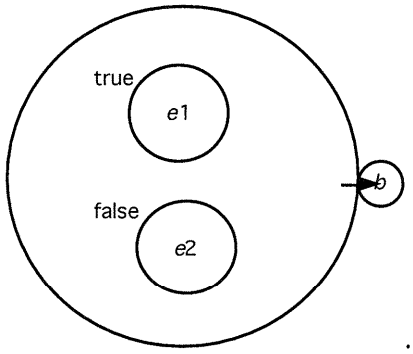


Figure 17. A selector construct

If the parameter is only used for indexing, it may be omitted. The parameter may also be bound to identifiers inside the internal expressions, in which case the conventional undirected edges are employed.

The selector construct generalizes to other things besides tuple construct and selection. It may be used to implement the conditional construct. The internal elements may be indexed with the values true and false, and the result of indexing such a construct is the true (resp. false) case. Thus, the conditional construct **if** b **then** e_1 **else** e_2 may be represented as



This, incidentally, represents the parallel conditional expression, in which the true and false cases are both evaluated, rather than the sequential conditional expression proposed by McCarthy [15]. We plan to investigate representation of sequential constructs in the future.

In addition to constants, VEX provides for primitive operators, as do most uses of the lambda calculus. A primitive operator, like a constant, is a closed figure labeled with the name of the operator. Thus, the addition operator is a circle labeled '+'. Figure 18 shows the binary addition operator as part of the expression +(3,4). Note the use of the tuple/selector construct.

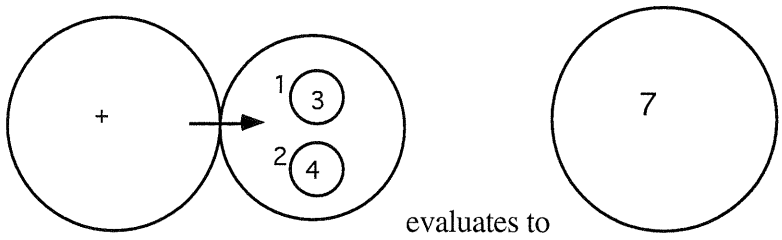


Figure 18. Evaluation of $+ \equiv \lambda(x, y).\, x + y$

The semantics of all the primitive operators can be specified by graphical transformation rules employing the conventional semantics of the operators.

Our representation of primitive binary operators such as addition, above, is useful in that it may be used to expose the mechanisms of currying, another concept that students often have a hard time grasping. Comparison of the representation of addition in figure 18 and the representation of a curried addition (ultimately employing the primitive binary addition operator) in figure 19 ahould reveal the links between the two functions.

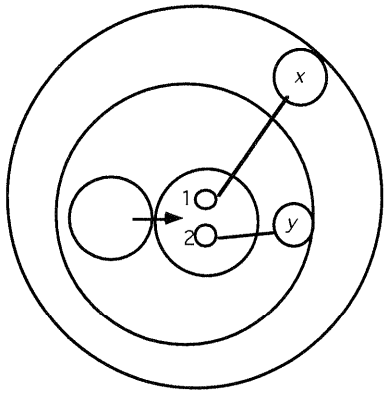


Figure 19. A curried version of addition $\lambda\, x.\, \lambda\, y.\, x + y$

5.2 Closures and environments

It is quite common for functional programming languages to allow the user to define a local binding for an identifier. This is typically done through the use of a **let** expression. For example, the expression **let** f **be** 3 **in** g means that all free occurrences of f in g are bound to 3. VEX provides a representation of **let** expressions based on the conventional definition **let** f **be** g **in** $h \equiv (\lambda f.h)g$. Thus, the VEX equivalent of **let** f **be** 3 **in** g is given in figure 20. Note that the free occurrences of f in g may be readily identified, and that the binding may be considered as a substitution that may be performed immediately or deferred until later.

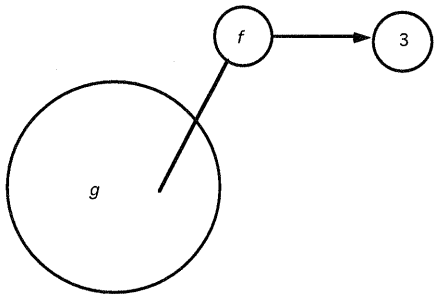


Figure 20. VEX representation of a local binding

It is useful to be able to collect the bindings given to all free variables of a given expression. This construct is generally referred to as an *environment*. In figure 20, the environment of g consists of the root node of f and its associated bindings (that is, the node labeled '3' and the arrow connecting it to f's root node). In the VEX support environment, one may construct environments by indicating to the system the desired expression and requesting the environment. From this, the system will collect copies of the free variables (possibly making duplicates of the graphical representations of the bindings, but this will not affect the meanings of the expressions), and group them together, placing a double box around them. VEX will also collapse the environment into a single object, if desired (as a space-saving feature), which can later be re-expanded. Figure 21 shows the representation of g's environment in the expression from figure 20.

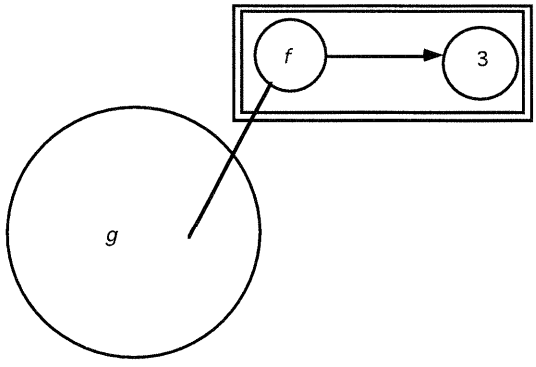


Figure 21. Representation of environments

This environment representation is useful in explaining the concepts of local bindings, and the way that subexpressions may possess separate environments. For example, consider the expression

$$\textbf{let } y \textbf{ be } 1 \textbf{ in}$$
$$\big((\textbf{let } x \textbf{ be } 2 \textbf{ in } \lambda z.x+y)(\textbf{let } x \textbf{ be } 3 \textbf{ in } x+y)\big)^{\cdot}$$

Figure 22 shows how the two subexpressions possess differing environments with different bindings for x, but the same binding for y.

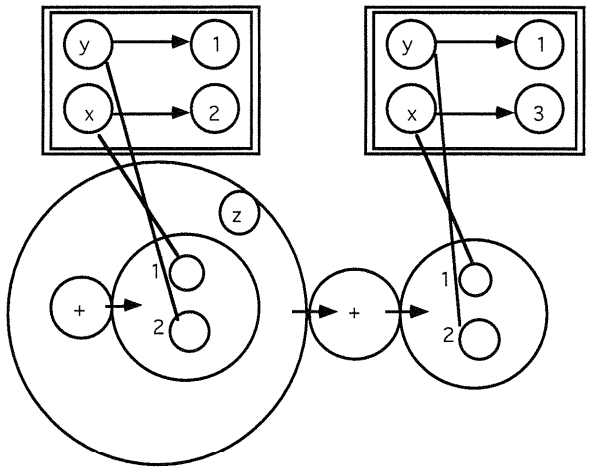


Figure 22. Multiple environments

Using VEX's concrete representation of environments, it is quite straightforward to explain to students the concept of a *closure* – the combination of an expression and its associated environment.

5.3 Recursion

The treatment of VEX, up to this point, has avoided dealing with recursion. However, the VEX representation simplifies discussions of least fixpoints and their role in recursion. Consider the recursive local definition **letrec** f **be** $\lambda x.f$ **in** g. Figure 23 illustrates the recursive definition of f. Because of the circular definition (f may be substituted for $\lambda x.f$ and the use of f refers back to the binding that refers to itself), we can get a diagram with infinite regression, as is shown in figure 24. We can also see that the environment of the abstraction $\lambda x.f$ (and of g) clearly includes itself. That circular definition is in fact the least fixpoint of the recursively defined environment of g that includes all approximations of f allowing a bounded number of recursions. The substitution semantics specified by the VEX graphical transformation rules yields the desired execution semantics.

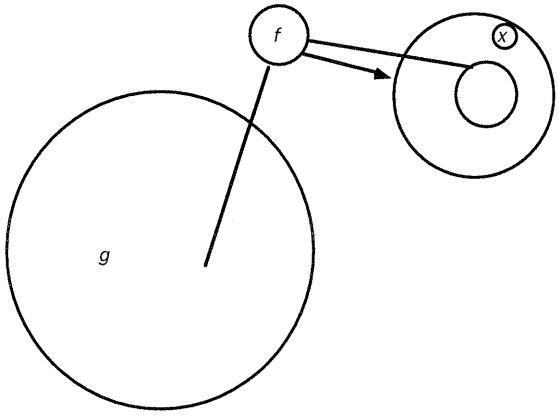


Figure 23. A recursive function definition

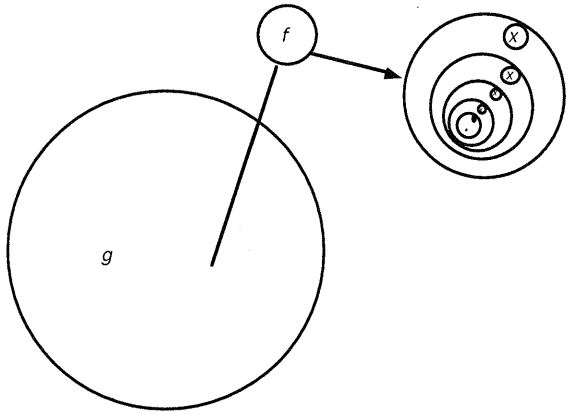


Figure 24. Recursion as infinite regression

6.0 RELATED WORK

While there has been some work in visual techniques and functional programming, none of it has been related to the visualization of lambda calculus and higher-order functions.

One related area is the visualization of LISP expressions and data structures, and visualization of the dynamics of their evaluation. Typically, visualization of data structures involves the visualization of structures built from Lisp cons cells. KAESTLE [3, 4] is a typical example of such a system, displaying a box-and-arrows diagram of the cons cells of a given data structure.

In order to display dynamic Lisp execution, Lieberman [14] presented a visual system, employing three dimensions and color, to illustrate the process of evaluation. Still, programs visualized by Lieberman's system were written in textual Lisp, and the visualization environment displayed visual representations of the underlying textual programs.

In contrast to these systems, Edel's Tinkertoy [9] provided a visual version of Lisp, in which Lisp S-expressions were drawn as trees. This system had the advantage of simplifying complex nested expressions by eliminating parentheses, but the system still did not address the problem of visualization of dynamically executing code.

A number of systems have been designed to help students learn functional programming. None of them employ visual techniques, however. The LISP Tutor system [1, 17] is a typical system developed to assist students in the acquisition of Lisp programming skills through the presentation of structured exercises. However, while useful for teaching basic Lisp programming skills, it is not clear that the benefits of the LISP Tutor carry over to the teaching of advanced and abstract concepts in functional programming and denotational semantics.

One important feature lacking in all of the representations, visualizations, and teaching techniques described above is a unification of the representation of the static program that was written, the representation of the dynamic program being executed, and the user's mental model of the executing program. Completely visual programming languages provide this unification, and the completely visual model has been adopted for VEX..

The concept of a completely visual programming language is due to Kahn [13], and is an extension of graphical transformation languages, including BITPICT [10] and ChemTrains [2]. In graphical transformation languages, a program is a set of graphical transformation rules, represented as "before/after" pairs of pictures, and program state is represented by a picture. Execution of a program involves repeatedly attempting to match the program state picture to the "before" pictures of the rules.

Completely visual languages take this model further by combining the program and the program state in a single picture. A graphical configuration is a snapshot representing both the current state and the remaining program at some point in the program execution. The separate set of graphical transformation rules now corresponds to the semantics of the programming language rather than the semantics of the program, which is completely incorporated into the single program configuration diagram. In a completely visual language, the static program, the dynamic visualization of the executing program, and (if the language has been properly designed with reference to a particular application domain) the user's mental model of the problem are all represented using the same visual notation, and can be understood through a set of simple and intuitive graphical transformation rules. If the completely visual language is a graphical representation of a textual programming language, as is the case with Pictorial Janus, described below, a programmer can write and understand programs in the graphical language without knowing anything about the underlying textual language, but simply by understanding the set of graphical transformation rules.

The best known completely visual language is Pictorial Janus [13], based on the concurrent constraint language Janus [18]. As mentioned above, a programmer can write and comprehend a Pictorial Janus program without knowing anything about the underlying Janus program despite the fact that there is a non-to-one correspondence between Pictorial Janus and Janus constructs. All that is necessary is an understanding of Pictorial Janus's graphical transformation rules. A detailed explanation of Pictorial Janus is beyond the scope of this paper; readers should consult [12, 13].

7.0 CONCLUSIONS

VEX is a graphical representation of the lambda calculus and functional programming extensions that we believe addresses a number of problems encountered by students when they first encounter the lambda calculus. It addresses the confusing aspects of naming, substitution, and freeness by replacing textual naming with explicit connectedness. It provides concrete visualizations for all values and expressions,

including functional arguments, higher-order functions, and abstractions. It also provides concrete visualizations for environments and least fixpoints.

VEX also provides simple and intuitive graphical transformation rules in place of the more complex textual rewrite rules of the lambda calculus, which often need to incorporate special cases to handle different combinations of freen and bound variables. We believe that these simpler rules will also help students understand the important issues in functional programming and lambda calculus. We do not believe that VEX should replace the lambda calculus and be taught exclusively – the lambda calculus is more compact than VEX and possesses a well-founded and extensive theoretical foundation – but we feel that it is useful as a supplement to the lambda calculus, and may profitably be used in parallel with introduction of new and complicated material.

We also intend to continue to refine the VEX representation. We hope to make the representation cleaner, and to continue work on the graphical formalism by which the graphical transformation rules are defined.

Finally, we are incorporating VEX into the VIPR programming language as the functional and expression-oriented component, to complement the imperative control structures that already have a graphical representation in VIPR. We are currently implementing a VIPR environment, and will soon begin to incorporate VEX into it.

ACKNOWLEDGMENTS

This work was funded by the Colorado Advanced Software Institute and USWEST Technologies.

REFERENCES

[1] Anderson, J. R., F. G. Conrad, and A. T. Corbett, "Skill Acquisition and the LISP Tutor." Cognitive Science, 1989. 13(4): 467-505.

[2] Bell, B. and C. Lewis, "ChemTrains: A Language for Creating Behaving Pictures," in *IEEE Symposium on Visual Languages*. 1993. Bergen, Norway, 188-195.

[3] Böcker, H.-D., G. Fischer, and H. Nieper-Lemke, "The Role of Visual Representations in Understanding Software." *Artificial Intelligence and Software Engineering*, 1989. .

[4] Böcker, H.-D. and H. Nieper, "Making the Invisible Visible: Tools for Exploratory Programming," in *First Pan Pacific Computer Conference*. 1985. Melbourne, Australia, 563-579.

[5] Citrin, W., M. Doherty, and B. Zorn, "A Formal Definition of Control Semantics in a Completely Visual Language," Technical report CU-CS-673-93, Department of Computer Science, University of Colorado, Boulder, September 1993 - revised June 1994.

[6] Citrin, W., M. Doherty, and B. Zorn, "Formal Semantics of Control in a Completely Visual Programming Language," in *IEEE Symposium on Visual Languages*. 1994. St. Louis, 208-215.

[7] Citrin, W., R. Hall, and B. Zorn, "Formal Specification of VEX transformation rules," Technical report *in preparation*, Department of Computer Science, University of Colorado, Boulder, January 1995.

[8] Curry, H. B. and R. Feys, *Combinatory Logic I.* 1958, Amsterdam: North-Holland.

[9] Edel, M., "The Tinkertoy Graphical Programming Environment," in *Visual Programming Environments: Paradigms and Systems,* Glinert, E., ed. 1990, IEEE-CS Press: Los Alamitos, CA. 299-304.

[10] Furnas, G. W., "New graphical reasoning models for understanding graphical interfaces," in *Human Factors in Computer Systems: CHI '91 Conference Proceedings.* 1991. New Orleans, 71-78.

[11] Hudak, P., "Conception, Evolution, and Application of Functional Programming Languages." *Computing Surveys*, 1989. **21**(3): 359-411.

[12] Kahn, K. M., "Towards Visual Concurrent Constraint Programming," Technical Report SSL-91-092, Xerox Palo Alto Research Center,

[13] Kahn, K. M. and V. A. Saraswat, "Complete Visualizations of Concurrent Programs and Their Executions," in *IEEE Workshop on Visual Languages.* 1990. Skokie, IL, 7-15.

[14] Lieberman, H., "A Three-Dimensional Representation of Program Execution," in *IEEE-CS Workshop on Visual Languages.* 1989. Rome, 111-116.

[15] McCarthy, J., "A basis for a mathematical theory of computation," in *Computer Programming and Formal Systems,* Braffort, P. and D. Hirschberg, eds. 1963, North-Holland: Amsterdam. 33-70.

[16] Milner, R., "Program semantics and mechanized proof," in *Foundations of Computer Science II, part 2,* Apt, K. R. and J. W. de Bakker, eds. 1976, Mathematical Centre: Amsterdam. 3-44.

[17] Pirolli, P., "A Cognitive Model and Computer Tutor for Programming Recursion." *Human-Computer Interaction*, 1986. **2**: 319-355.

[18] Saraswat, V., K. M. Kahn, and J. Levy, "JANUS: A step towards distributed constraint programming," in *North American Logic Programming Conference.* 1990. Austin, TX, 431-446.