Functions as Data Objects in a Data Flow Based Visual Language

Alex Fukunaga†, Wolfgang Pree††, Takayuki Dan Kimura††

†Harvard University
Cambridge, Massachussetts 02138

††Department of Computer Science
Washington University
St. Louis, Missouri 63130

Abstract

Data flow based visual programming languages are an active area of research in visual programming languages. Some recent data flow visual programming languages have implemented higher order functions, allowing functions to be passed to/from functions. This paper describes a data flow visual programming language in which the first class citizenship of programs have been taken a step further, and programs can be manipulated as data with the same kind of flexibility that LISP offers in manipulating programs as data.

1. Introduction

It is widely accepted that higher order functions provide a substantial amount of power and flexibility to programming languages which support them. The ability to pass functions to and from functions allow for the creation of general functions which can easily be adapted to a variety of situations. Thus, programming languages like ML[Miln84] and Miranda[Turn90] are fully higher order - functions can be passed into and returned out of functions.

Although the implementation of higher order functions - the treatment of functions as first class citizens - is important in itself, it is also significant in that it is indicative of another concept in programming languages: the integration of programs and data. The ability of LISP and its dialects to represent and treat programs and data in the same way has been very significant in artificial intelligence and in program development environments.

The implementation of higher order functions is a recent development in the field of visual programming languages. Higher order functions have been implemented to varying degrees in several data flow visual programming languages. However, there have been no implementations of data flow languages in which programs could be treated as data with the same generality in which LISP treats its programs and data. This paper describes such a language.

2. Previous Work

Some earlier data flow visual programming languages have provided for higher order functions, including CUBE[Najo91], DataVis[Hils91], VPL[Lau91], Enhanced Show and Tell (ESTL) [Najo90], and Show and Tell [Kimu86].

Higher order functions have been represented in two ways in data flow visual programming languages. The first approach uses function slots inside icons for higher order functions, into which lower order functions are "slotted in." DataVis, CUBE, and ESTL use this approach. For example, Figure 1 shows an example of a higher order function using this approach (based on DataVis). The higher order function **Iterate** contains **F**, a first order function in its function slot. **Iterate** applies **F** to each member of the list **a**, **b**, **c**, **d**, and returns the list **F(a)**, **F(b)**, **F(c)**, **F(d)**.

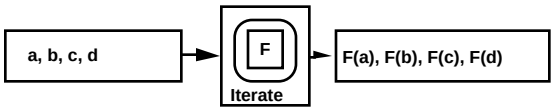

Figure 1: Higher Order Functions Using Function Boxes

In the alternate approach, functions are "quoted", and flow over links. That is, functions are treated as data objects, and an "unquote" or "apply" node is used to apply the function to its inputs. Show And Tell and VPL use this method, as does PHF, the language we designed and implemented for this work

However, although higher order functions have been repeatedly implemented in visual programming languages, there has been little effort to further eliminate the distinction between code and data. The language described below provides a sufficent set of primitives to treat functions as data objects with the same kind of generality that LISP allows.

3. ProtoHyperflow

Hyperflow is a dataflow visual programming language based on Show and Tell, and is designed for use with a pen-based multimedia system. Hyperflow is described in [Kimu92].

ProtoHyperflow (PHF), is a visual programminlg language which is a derivative subset of HF, and originated as a prototype implementation of HF. The current inplementation was done on a traditional mouse/CRT based system using C++ and the ET++ GUI applications framework [Wein88], and is a integrated editor and data driven interpreter system. The following is an informal description of the PHF language.

3.1 PHF Syntax

The syntax of PHF consists of boxes and arrows, a box representing a process and an arrow representing a data flow between processes. Boxes are called *vips* (visually interactive processes) in PHF, and arrows are called *connectors*.

Computation in PHF is carried out by a homogenous community of vips communicating with each other. Vips can be recursively nested.

The vip is the only unit of system decomposition in PHF, paralleling the design of LISP, in which lists are the only structure. This allows PHF's ability to treat programs as data objects, as described in section 4. A vip consists of a *mailbox*, a *body*, and an optional *name*. A mailbox holds a discrete data object, such as an integer or string. The body is the semantic content (the implementation of the semantics) of a vip. The body of a vip can be a system defined PHF primitive, a reference to another vip, a nested ensemble of vips, or it may be empty. A vip may also have a name, which appears on the top left corner of the vip. Names are necessary when defining functions (see section 3.5.1). A connector establishes dataflow between the two vips that it connects. A connector may also have a label. A vip ensemble is a directed acyclic graph, where the nodes are vips and the edges are connectors. A PHF program is a vip containing an vip ensemble.

It is necessary to introduce some shorthand terminology here, in order to facilitate a more detailed discussion of the constructs used in PHF. We shall define 'vip X' to mean 'the vip with the name X, and 'connector X' to mean 'the connector with the label X'. Also, an empty vip shall be called a variable vip.

3.2 Data Objects in PHF

The following data objects are currently implemented in PHF: 1)integers, 2)strings, 3)signals, 4)vips. Strings in PHF are prefixed with a quote (') in order to distinguish them from references to other vips. Signals are an enumerated data type which is either **valid** or **invalid**, and are used to denote the result of a predicate (such as = ,<>). The use of vips as data objects is detailed in section 4. All data objects can be transmitted via mail (see below), and can be displayed in a variable (empty) vip.

3.3 Communication Between Vips

There are two modes of communication in PHF: 1) *mailing*, and 2) *broadcasting*.

Mailing is communication of discrete data objects by dataflow across connectors. In mailing, the contents of the mailbox of the source vip is copied to the mailbox of the destination vip. This is the standard mode of communication between vips.

Broadcasting is a special mode of communication which involves no connectors. A broadcasting vip, which is denoted as a vip with a dotted border,(see Figure 6) transmits the contents of its mailbox to all of the children of its parent (its sibling vips).

3.4 PHF Execution Protocol

A PHF program is executed from its outermost vip. The execution mode implemented by PHF is, as with most current data flow visual languages, data driven.

A vip is executable exactly once, and will execute when it has the minimum number of valid inputs (input connectors on which the source's mailbox is ready to be transferred). The number of minimum inputs is a semantic property of a vip. For example, a variable vip (an empty vip) will execute as soon as it has one valid input (all other inputs will be ignored), while a + primitive (summation) will not execute until all of its inputs are valid. Thus, if a variable vip X has two input connectors with sources at vip Y and Z, then this results in a nondeterministic behavior, where the value transfered to X is the value of the source which is ready first (If Y is ready to transmit first, then X receives the value of Y, but if Z is ready to transmit first, then X receives the value of Z).

3.5 PHF Programming Constructs

The following sections describe key programming constructs in PHF.

3.5.1 Primitive and User Defined Functions, Binding Rules

A vip may invoke a system defined primitive, or a user defined function.

Figure 2 shows a PHF program which calculates the sum of 4 and 3. The + vip is a system defined primitive which returns the sum of all of its inputs.

Another type of function is one in which involves parameter binding. For example, for a binary division operation (a division with two inputs), it is necessary to distinguish which of the operands is subtracted from the other. Thus, binding rules are necessary.

The binding rules in PHF are name based. In the case of system defined primitives, the parameters which need to be bound to input values are defined by the system as **#1**, **#2**, ... **#n**, where **#1** is the first argument, **#2** is the second argument, and so on. Input values are bound to these parameters by labeling the connectors which connect the input values and the primitive. Thus, in a binary division, the dividend must be bound with system parameter **#1**, and the divisor must be bound with system paramter **#2**, and the '/' primitive will return the value of **#1**/**#2**. Figure 3 shows a PHF program which calculates 10/2.

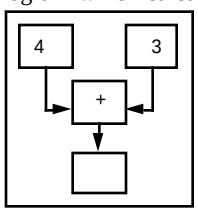

Figure 2: Addition

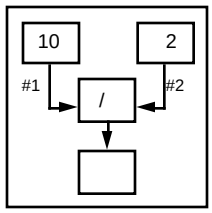

Figure 3: Division

User defined functions are implemented by naming a vip, and then referencing that vip from another vip. Figure 4 shows the definition of **Increment**, a vip which takes one input parameter in vip X, and returns X+1. The **@** vip is a system defined primitive which transfers the contents of its mailbox to its parent. Figure 5 shows **CalcInc**, a PHF program which calls the **Increment** function with input value 5. When **CalcInc** is executed, the 5 is mailed to the vip calling **Increment**. A copy of the function **Increment** is created on the execution stack, with 5 bound to the vip X, and is executed. The result of the addition, 6, is sent out of the **Increment** function to its parent, which is the vip which calls **Increment** in **CalcInc**, and the result then flows to the variable vip at the bottom of the **CalcInc** vip.

Binding of input values to unbound variable vip in the function is established by associating the labels of the connectors entering the function call vip with the names of the parameter vips of the function. Thus, in the example above, the empty variable vip X in Figure 4 is associated with the connector labeled X in Figure 5.

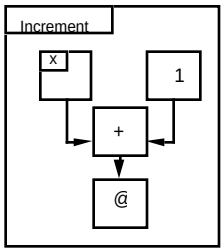

Figure 4: Increment Function

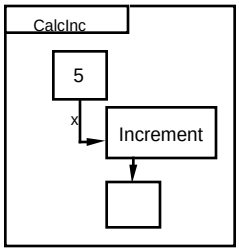

Figure 5: CalcInc

PHF functions have only one return value, which is mailed out via the **@** vip. Note that it is not possible to have side effects in PHF, because of its data flow based nature.

3.5.2 Conditionals

Conditionals are implemented in PHF using the broadcasting mechanism (section 3.3) and the signal data type (section 3.2). If a vip receives an **invalid** signal, it is inactivated so that it does not execute. A broadcasting vip can prevent all of its sibling vips from executing by broadcasting an **invalid** signal. Thus, conditionals can be implemented by having multiple vips, among which only one is selected by invalidating all of the others.

Figure 6 shows a PHF program which takes one input, X. The = and <> are PHF primitives which return **valid** or **invalid** depending on whether the 'equal' and 'not equal' predicate holds true for their inputs. If X=0, then the = predicate returns a **valid** signal, while the <> predicate returns an **invalid** signal. Thus, the string "Zero" flows to the vip labeled **result**. However, if X <> 0, then the string "Other" flows to the vip labeled **result**.

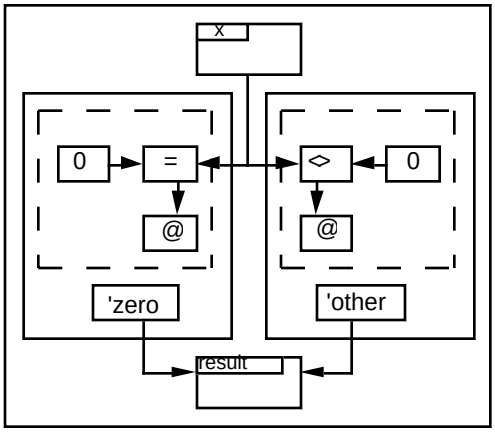

Figure 6: Conditional Example

Note that PHF uses an asynchronous, parallel execution model, so a vip executes as soon as its inputs are ready. Thus, when making conditional statements, the conditionals must be mutually exclusive, or the results will be unpredictable (the program will be syntactically correct, but will behave nondeterministically (Section 3.4).

3.5.3 Recursion

Recursive function calls are possible. Figure 6 shows a PHF program which calculates the factorial function (!). This function takes one integer input, X, and processes it as follows: if X=0 then return 1 else return X * fact(x-1). (X is assumed to be positive).

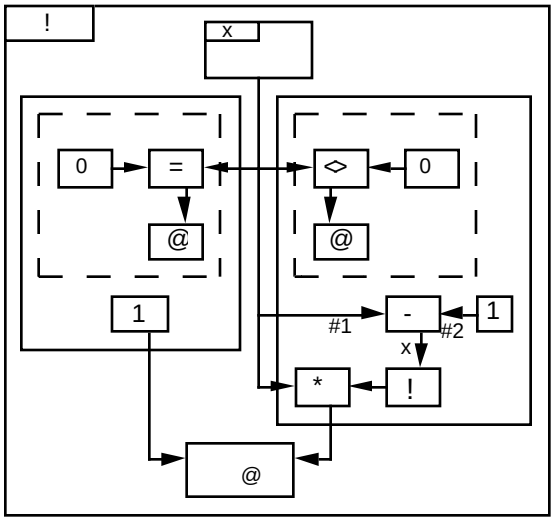

Figure 7: Factorial Function

4. The PHF implementation of Higher Order Functions

PHF implements higher order functions by treating vips as data objects which can flow over connectors.

A vip (which can be an ensemble of vips, since vips can be recursively nested) can be quoted by enclosing it within a parent vip with a thickened border, making it a data object. The quoted vip can then be executed later by the system defined **Apply** primitive vip. The **Apply** vip takes one quoted vip as an input, with the connector 'func', and can take any number of input parameters which are bound to input parameters of the quoted vip function. The **Apply** returns the result of the evaluation of the quoted vip. Note that the result of an **Apply** could be a quoted vip.

In addition to the quote/**Apply** operations, PHF implements primitives allowing the full manipulation of vips as data.

A sufficient set of visual language primitives to manipulate functions as data requires 1) a set of constructive primitives, and 2) a set of destructive primitives. In PHF, the constructive primitives are **Insert** and **Connect**, and the destructive primitives are **Extract**, **Remove**, and **Disconnect**. Furthermore, in PHF, there must be primitives which manipulate the other attributes of vips. **Unquote$**, **GetName**, **SetName**, **GetBorder**, **SetBorder**, **GetConnLabel**, and **SetConnLabel** manipulate vip names, border attributes, and connector labels.

These primitives, defined below, allow the full manipulation of vips as data objects. Note, however, that these primitives deal with the logical structure of vips, but not the visual layout of the results of these operations. Thus, it is possible to create any logical PHF program using these primitives, but it is not possible to specify the topology of the program in two dimensional space.

4.1 Primitive Definitions

For simplicity of syntactical description, in the primitive definitions below, we shall name variable vips which contain quoted vips, but it is not necessary to actually name these vips in all cases with the names that we have

given them, and in many cases, names are not necessary at all.

Unquote$ (unquote string) takes a string as input and removes its quote, thereby making it a possible function or primitive reference (Figure 8). Note that while some earlier visual languages use the term 'unquote' to mean evaluating a higher order function, the term 'unquote' in PHF strictly means to remove the quote from a string.

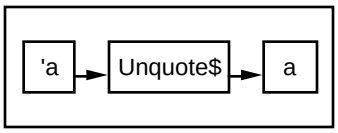

Figure 8: Unquote$

GetBorder takes a quoted vip as input and returns the type of its border (normal, quote, or broadcast) as a string (i.e. 'normal, 'quote, 'broadcast). See Figure 9.

SetBorder takes a quoted vip X with connector 'func', and a string S with connector label #1 as input, which can be 'normal, 'quote, or 'broadcast. **SetBorder** assigns the border type specified by S1 to X (Figure 9).

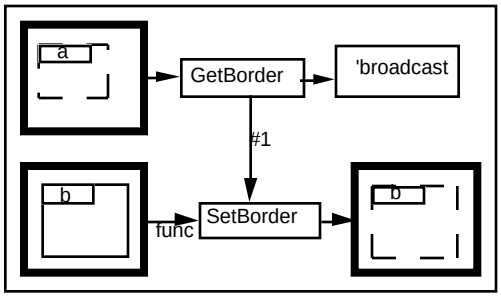

Figure 9: GetBorder, SetBorder

GetName takes a quoted vip as input and returns the name of the vip as a string (Figure 10).

SetName takes a quoted vip X with connector 'func' and a string S with connector label #1 as input. S is unquoted, and is assigned to the name of X (Figure 10).

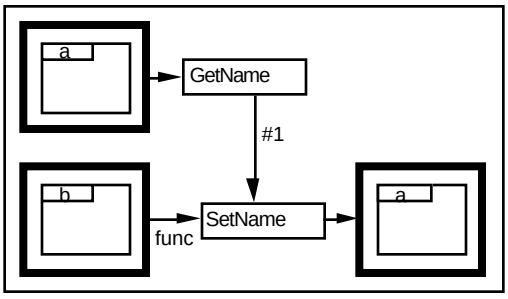

Figure 10: GetName, SetName

Insert takes two quoted vips, X and Y, as input, with their corresponding input connectors labeled #1 and #2, and returns a new vip in which the vip X has been inserted into the vip Y (Figure 11). Note that vip Y must be an ensemble.

Connect takes a quoted vip X with connector 'func', and three string inputs S1, S2, and S3, with connectors labeled #1, #2, and #3, respectively. **Connect** returns a new quoted vip, identical to X, except with a new connection between the vips S1 and S2, where the vip at the source of #1 is the source (Figure 11). The label of the new connector is S3, unquoted.

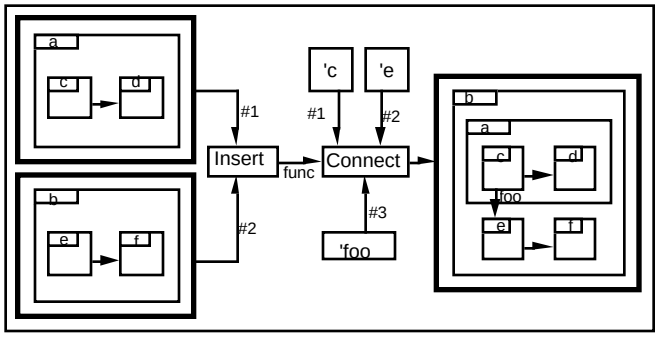

Figure 11: Insert, Connect

S3 is an optional parameter. If no label name is specified, the connector will have no label.

Extract takes two inputs: 1) a quoted vip ensemble X with connector 'func', and 2) a string S, with a connector labeled #1. The result is the vip named S found in X (Figure 12).

Remove takes two inputs: 1) a quoted vip ensemble X, and 2) a string S, with a connector labeled #1. The result is X, with the vip named S removed. All input and output connectors to/from vip S1 are removed (Figure 12).

Disconnect takes a quoted vip ensemble X with connector 'func', and three string inputs S1, S2, and S3, with connectors labeled #1, #2, and #3, respectively, where S3 is optional. The result is X without the connector S3 between S1 and S2, where vip S1 was the source. Note that the direction of the connector to be removed is significant, because it is possible that there is a bidirectional connection between vips named S1 and S2, where S3 is a common label such as '#1'. If S3 is not specified, all connections from S1 to S3 are removed (Figure 12).

GetConnLabel takes a quoted vip X with connector 'func', and two strings S1 and S2 as input with connectors #1 and #2, respectively. The label of the connectors from vips S1 to S2 in vip X is returned.

SetConnLabel takes a quoted vip X with connector 'func', and three strings S1, S2, and S3 as input with connectors #1,#2, and #3, respectively. The Connector in X from vip S1 to vip S2 is assigned the label S3, unquoted. The usages of **GetConnLabel** and **SetConnLabel** are illustrated in Figure 14 in the next section. The **GetConnLabel** and **SetConnLabel** operations were defined to be used when only one connection exists between two vips. These operations need to be refined to take into consideration cases when vips are multiply connected.

4.2 Scoping Rule

Note that many of the primitives described above must search for vips within vips. However, since vips can be indefinitely nested, a question arises over what to do if there are different vips with the same names, but at different levels of nesting. When searching for vips, PHF uses a breadth first

search method which finds the closest vip to the outermost vip. That is, it finds the least deeply nested vip. In Figure 13, if PHF searches for vip X in V, it will find the singly nested vip X, not the X which is nested within Y.

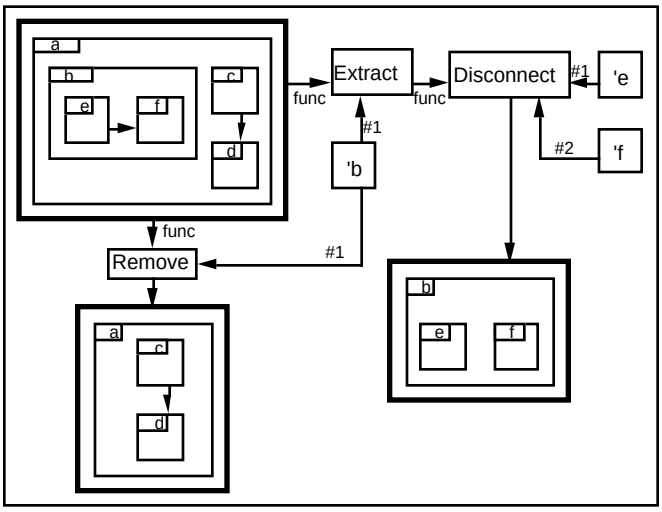

Figure 12: Extract, Remove, Disconnect

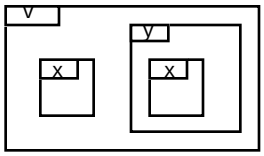

Figure 13: Scoping Rule

4.3 Example - ReverseOp

Figure 14 shows **ReverseOp**, a higher order function which takes a function F and three vips X, Y, and Z as input, where X, Y, and Z are all in F, and X and Y are both connected to Z, with the connections being directed from X to Z and from Y to Z. This is commonly encountered when Z is an operation, and X and Y are its operands. **ReverseOp** exchanges the connections between X and Z and Y and Z in F and returns the new function. The function works as follows: The labels for the X-Z and Y-Z connections are stored, and the connections are severed (**Disconnect**ed). Then, new connections are established (using **Connect**), switching the connector labels.

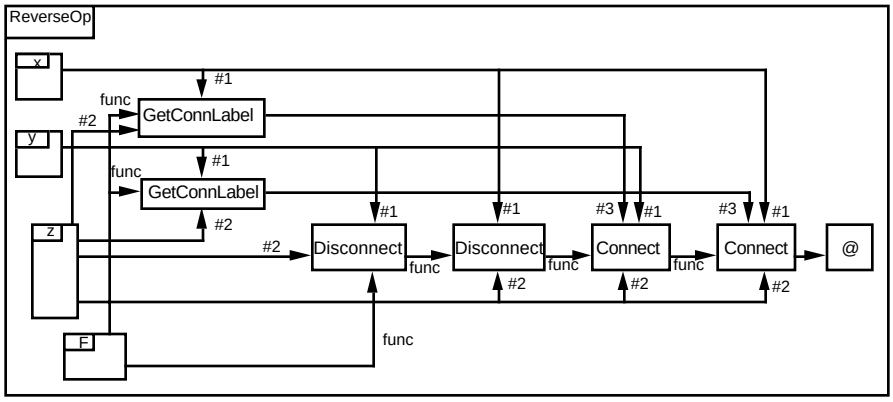

Figure 14: ReverseOp

A simple instance where this might be used is when Z is an arithmetic operation where the order of its operands is important, such as the '-' operation. Then, if **a**-**b** is calculated in F, the result of **ReverseOp** would calculate **b**-**a** (Figure 15a). A more sophisticated use of the function might be to reverse two arbitrary connections in a large, self modifying PHF program (Figure 15b).

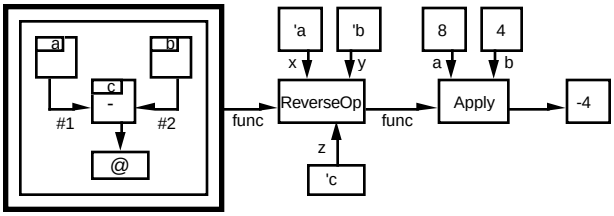

Figure 15a: ReverseOp Used to Reverse Operands for Subtraction

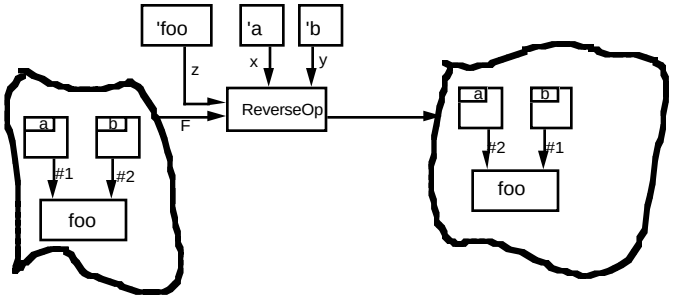

Figure 15b: ReverseOp Usage

4.4 Future Work: Visualization of the Output of Vip Manipulation Primitives

Under the current implementation of PHF, there is no way to see the result of operations which manipulate vips, other than by **Apply**ing the quoted vips and observing the results (the results of vip manipulations in Figures 9-12 were drawn by hand). The ability to directly observe the result of such manipulations would be a crucial facility in a practical system.

However, the question of how the spatial relationships of the underlying logical structure of a visual language should be manipulated poses a formidable future research problem in itself. Destructive operations are trivial to visualize, since they only involve the removal of visual elements. However, constructive operations such as the **Insert** operation pose a problem. For instance, in Figure 16, if

vip X is **Insert**ed into vip Y, exactly where in vip Y should vip X appear? We propose that there should be a simple default visual concatenation rule, such as simply inserting X at the leftmost extent of Y, and expanding Y to include X. A more flexible system would allow an exact specification of the spatial relationship between results of operations on vip data structures.

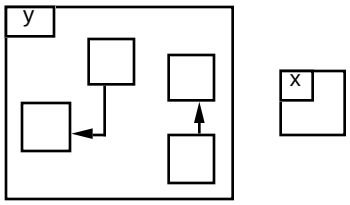

Figure 16: Layout Problem- where should vip x be inserted?

5. Discussion

This paper presented a data flow visual language in which programs could be fully manipulated as data objects.

LISP's powerful ability to manipulate programs as data stems from the fact that s-expressions are the only unit of system decomposition in LISP. Likewise, PHF's ability to manipulate PHF programs as data is due to the fact that vips are the only unit of system decomposition. If one thinks of vips as nodes and connectors as edges, then a PHF program is simply a directed acyclic graph. In essence, PHF can be thought of as a "graph processing language", in the same sense that LISP is a "list processing language".

There is one significant difference: while there is a strong isomorphism between LISP's one dimensional list data structure and its symbolic representation, the isomorphism between a PHF program's logical structure (a graph) and its visual representation (the layout of the graph in two dimensional space) is not clear. As we noted in Section 4.4, this poses the a new problem in that now, with visual languages, we must also specify the spatial as well as the logical relationships between vips. PHF can completely manipulate vips at a logical level, but further research is required before we have a visual language which can easily manipulate functions as both logical and visual entities.

There are clear advantages in using the list, a one dimensional data structure, as the fundamental structure in a one dimensional textual language. It is a clear, simple, paradigm, and higher dimensional structures can be constructed by the nesting of one dimensional structures. On the other hand, graphs are a more general structure, but are more difficult to visualize in a textual context.

However, with the two dimensional visualization that visual languages offer, it is not difficult to envision using the graph as a fundamental structure, since the medium makes it easy to visualize and manipulate the structure. Future research will reveal what impact this may have on data structure and algorithm design.

References

[Hils91] Hils, D.D. "Visual languages and computing survey: data flow visual programming languages," Journal of Visual Languages and Computing 3:1 (1992) pp 69-101.

[Kimu86] Kimura, T.D., Choi, J.W. and Mack, J.M. "A Visual Language for Keyboardless Programming," Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, June 1986.

[Kimu92] Kimura, T.D. "Hyperflow: A Visual Programming Language for Pen Computers," to appear in IEEE Workshop on Visual Languages, Seattle, Washington, 1992.

[Miln84] Milner, A.J.R.G. "A Proposal for Standard ML," Proceedings of the ACM Symposium on LISP and Functional Programming, Austin, Texas, 1984.

[Lau91] Lau-Kee, D., Billyard, A., Faichney, R., Kozato, Y., Otto, P., Smith, M., Wilkinson, I. "VPL: An Active, Declarative Visual Programming System," Proceedings of IEEE Workshop on Visual Languages, Kobe, Japan, 1991, pp 40-46.

[Najo90] Najork, M.A., Golin, E. "Enhancing Show-and-Tell with a polymorphic type system and higher order functions," Proceedings of IEEE Workshop on Visual Languages, Skokie, Illinois, 1990, pp. 215-220.

[Najo91] Najork, M.A., Kaplan, S.M. "The CUBE Language," Proceedings of IEEE Workshop on Visual Languages, Kobe, Japan, 1991, pp 218-224.

[Turn90] Turner, D. "An Overview of Miranda," in *Research Topics in Functional Programming*. Addison-Wesley Publishing Company, 1990, pp 1-16.

[Wein88] Weinland A., Gamma E., Marty R. "ET++ - An Object-Oriented Application Framework in C++," OOPSLA '88, Special Issue of SIGPLAN Notices 25:11 (1988).