Higher-Order Functional Composition in Visual Form

Dami, Laurent; Vallet, Didier

Higher-Order Functional Composition in Visual Form

Laurent Dami and Didier Vallet

Abstract

A visual formalism for functional composition is presented, which departs from many other visual systems in that it is not merely dataflow: the formalism also supports the notion of *higher-order* composition, i.e. treating functions as data. This is done through a simple notion of graph rewriting, which can be explained in a very intuitive fashion by moving software components along dataflow paths. The intended goal of this formalism is mainly didactical: some non-trivial aspects of the lambda calculus or other higher-order rewriting systems can be demonstrated without any mathematical background.

1. Introduction

It is common folklore to say that functions can be seen as black boxes, transforming some inputs into some output. This view makes it easy to explain functional composition as an assembly of connected boxes, where the output of a box is used as input to the next box. Furthermore, it is clear that such an assembly can be put in a bigger box, under a new black cover; in other words, encapsulation of functional composition has a direct graphical counterpart. Examples of such statements can be found in textbooks on functional programming, and a number of visual programming languages have implemented the "functions as boxes" approach [1][2][3][5][6]; therefore the visual representation of functions is widely acknowledged as an adequate tool for teaching functional programming and functional composition. In Figure 1 we provide an example of a graphical notation to support this common view. Functions are pictured, not as black boxes, but as white boxes (this is more convenient for complex graphs, as will be seen later); they get input and produce output across *ports*, which are pictured as small rectangles; finally, connections between functions are represented by arrows. The picture displays a function which takes three inputs. On the left, we see the function as an encapsulated box. On the right, we see in a white big box the "inside" of the function, which is a configuration of three other functions. The inputs of the encapsulated configuration are distributed across the internal components, and the final result is exported through the output port at the bottom.

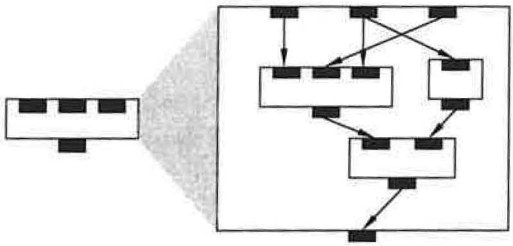

Figure 1 Example of a functional graph

The notation is convenient for functional composition, but nevertheless is fairly limited. The reason is that it does not capture the full story about functions: we can show how data flows from one function to the next, but we cannot show a fundamental aspect of functional programming, namely the fact that *functions themselves are data*, and can be used as input to other functions. Such *higher-order* programming features are essential for example for modelling recursion. Since the boxes and arrows of Figure 1 can only represent a static dataflow configuration, there are many black boxes which cannot be "opened up" graphically: to explain what they do, we have to go to other formalisms. As a matter of fact, functional visual programming systems often escape the dataflow limitation by adding name references to a global environment (this can bring support for recursion), or by adding the notion of primitive boxes written in some external language.

In order to capture higher-order programming in a visual formalism, what is needed is some notation for functional application. Instead of using only functions which are known statically, we should be able to get a function at some input port, and then *apply* it to some data available in the current configuration. This corresponds to a functional expression of the form $(x\ a)$, where x is a variable, while a is any expression. To do so, we introduce the notion of *grey box*. A grey

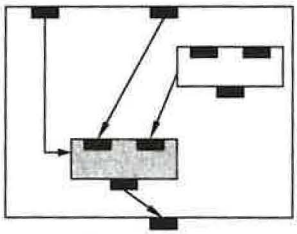

Figure 2 Example of a grey box

box is a template for a function which will be filled up later. It can be seen as a printed circuit, prepared to receive a chip, but in which the chip has not been plugged yet. The question, then, is: how and when will the chip be supplied? This is answered by a set of *rewrite rules* associated with the graphical notation. The chip (the function) travels in the system across usual data paths, until it reaches its grey box, at which point it can start to operate. Although it would obviously not make sense in terms of real hardware, the notion of a "chip travelling in the circuit" is extremely simple to explain and understand. The example in Figure 2 displays a configuration in which a function is expected to arrive at the leftmost input port, for filling the grey box on the bottom. All the circuitry to feed arguments to that function and to collect the result is already prepared.

The resulting graphical system has the full power of the lambda calculus. It is primarily intended as a teaching tool, maybe not only for functional programming, but also for more general purposes like principles of formal systems. However, it may also provide interesting insights into topics such as graph reduction, parallel reduction, or out-of-order parameter binding. Such possibilities are briefly discussed in the last section.

For the rest of this paper, we try to introduce all concepts at a purely graphical level, i.e. without requiring any background in functional programming and the lambda calculus; the objective is to show that these principles can effectively be taught in a graphical environment. The syntax is specified in section 2.; section 3. presents the rewrite rules; and section 4. illustrates the system with some simple examples. For specialists of functional programming, section 5. relates our work to known results.

The system presented here has been implemented in a tool *VisualLambda*, which supports interactive edition of box graphs, and translates them into lambda expressions. These expressions are then fed to a usual functional interpreter. The tool is described in section 6.

2. Box graph formation

This section presents the syntax of box graphs. Since our objective is pedagogical, the presentation is in textual form, rather than using some graph grammar formalism; however, it is hopefully precise enough to rule out most ambiguities.

A *box graph* consists of boxes, ports, and connections. Boxes are non-overlapping white or grey rectangles; ports are small black rectangles located on the border of a box; connections are directed arrows between ports and/or boxes. Box graphs must obey to some formation rules which are detailed below.

Boxes are divided into *white* and *grey* boxes. To stay with the metaphor introduced in the previous section, white boxes can be though of as chips, while grey boxes are holes in a printed circuit, prepared to receive chips. White boxes are furthermore divided into two kinds:

- *primitive boxes*. Primitive boxes (also called *constants*) may be entities such as numbers, truth values, strings, or primitive functions on these entities; they are pictured as white boxes, with a name inside the box identifying the entity. They may or may not have some ports on the border of the box, according to the rules for ports specified below. We do not bother to give a full specification of constants, since they operate at a distinct level from the rest of the system, and therefore can be chosen according to particular needs of the context. This is like what happens in the lambda calculus, which is often used in an *applied* form (with added constants and corresponding so-called δ-rules), without any interference with the mechanics of functional abstraction and application. In the following, we will mainly use integer constants, together with integer operators, such as: 

- *composite boxes*. Composite boxes are big white boxes which contain other boxes (either white or grey), with their associated connections. The inclusion is strict, i.e. boxes cannot intersect with each other. Composite boxes have their own ports. Grey boxes cannot contain other boxes.

Ports are small black rectangles located on the border of a box, either towards the inside or towards the outside of the box. Ports on the inside are called *input ports*, ports on the outside are called *output ports*. A box may have any number of input ports, but has exactly one output port.

Input ports are ordered; this can be indicated explicitly through integer labels, but most usually the ordering will be implicit, according to the following convention: ports on the left border of the box are numbered first, starting from the top; then ports on the top border are numbered, starting from the left; no input ports will appear on the other borders of a box. To make this clear,

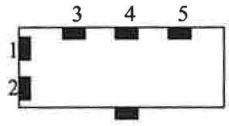

Figure 3 Port ordering

Figure 3 displays the port ordering for a box with five inputs.

For a given port, the side of the black rectangle which is shared with the adjacent box is called the *input side* of the port. The opposite side of the port is the *output side*. The notions of input/output sides are not to be confused with input/output ports; the difference is pictured in Figure 4, containing both input and output ports, with letters *i* and *o* indicating the input/output sides of ports.

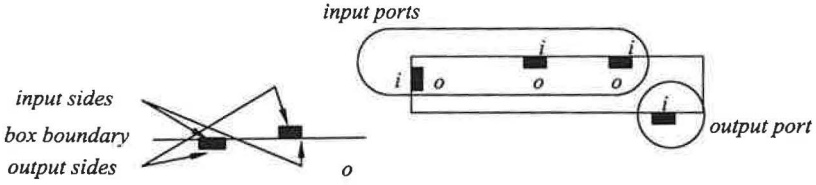

Figure 4 Input/output ports and sides

Connections are represented by directed arrows; they obey to the following rules:

- the *source* of an arrow is output side of a port (either input or output port), or the outside border of a white box (either primitive or composite)
- the *destination* of an arrow is the input side of a port, or the border of a grey box
- two arrows cannot share the same destination (but can share the same source)
- arrows cannot cross box boundaries.

A *path* is a non-empty collection of arrows $a_1 \ldots a_n$ such that the destination of a_i and the source of a_{i+1} are the same port. The source of a_1 is the source of the path, and the destination of a_n is the destination of the path. An arrow is *shared* if it belongs to several disjoint paths. By this definition, every path has at least one arrow which is not shared. *Removing* a path $a_1 \ldots a_n$ in a graph consists of removing its arrows wich are not shared, and the corresponding ports: if both a_i and a_{i+1} are not shared, then both arrows should be removed together with the port p whis is the destination of a_i and the source of a_{i+1}.

A box is *fully connected* iff all its input ports have incoming arrows; it is *free* iff all its input ports have no incoming arrow. By this definition, boxes like [1] without any input ports are both free and fully connected. Conversely, boxes with several input ports can be neither free nor fully connected, if just some of their input ports have incoming arrows.

A graph is *well-connected* iff

- every grey box is fully connected, and furthermore has an incoming arrow on its border
- if a white box is fully connected, then there is no outgoing arrow from its border; conversely, it is not fully connected, then there is no outgoing arrow from its output port
- the output port of every composite box has an incoming arrow
- there is no cycle, i.e. no path for which the source and the destination are the same port.

It may seem that there are many rules, and that building a well-connected box graph is a complex process. In fact, with just a little practice, most of these rules are obvious to the eye; the only rule which requires more complex, non-local analysis is the last rule about cycles.

An example of a correct graph is given in Figure 5. It corresponds to the lambda expression

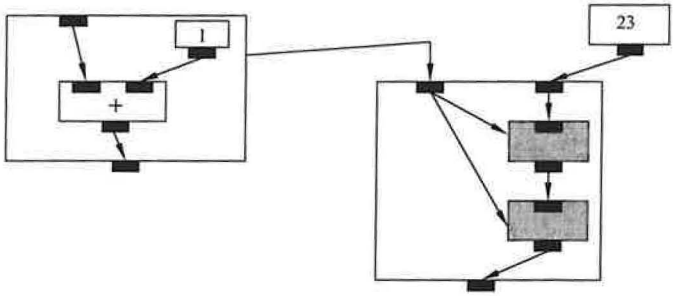

Figure 5 A simple box graph

(λf.λx. f(f x)) (λx. x+1) 23

This displays many features which are typical of box graphs. The composite box on the left is an incrementation function, using the constants '1' and '+'. The composite box on the right takes a function as first argument, and applies it twice to its second argument. What may seem somewhat unusual to a functional programmer is that functional application is not always pictured the same way. If the operator is statically known, then application is simply an arrow from the actual argument to the corresponding input port of the function. If the operator a variable, grey boxes must be used.

Figure 6 displays an incorrect graph, in which we tried to represent as many kinds of mistakes as possible. These are identified by small italic letters. The problems are the following:

a) a white box cannot be the destination of an arrow

b) the box has two output ports

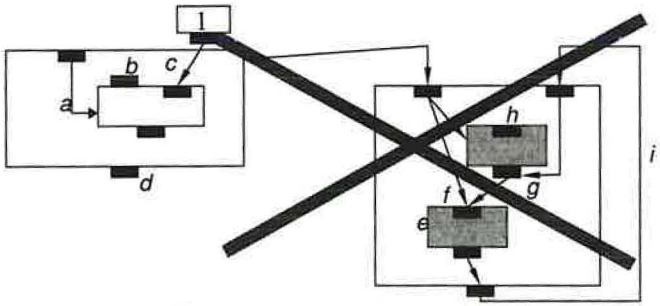

Figure 6 An incorrect graph

c) the arrow crosses a box boundary

d) the output port of the composite box has no incoming arrow

e) the grey box has no incoming arrow on its border

f) two arrows share the same destination

g) the destination is not the input side of the port

h) the grey box is not fully connected (input port without incoming arrow)

i) the arrow creates a cycle

3. Rewrite rules

Intuitively, the correct example of Figure 5 should yield the result 25. In order to specify how the system can produce this result, we need to introduce some rewrite rules. These are *delta* rules, for reducing graphs of constants, *beta* rules, for reducing usual abstractions, and *garbage collection* rules, for simplifying box graphs. The terminology is borrowed from the lambda calculus.

All rules are presented both with a textual explanation and with a more formal description using graph grammars [4]. A graph grammar rule has the shape of a 'Y'. Intuitively, the left-hand side represents the subgraph to be rewritten, the right-hand side represents the subgraph which replaces it, and the upper side represents the "context", i.e. nodes in the surrounding graph which have connections to either the original subgraph or the new subgraph. We will used small black circles ● to denote abstract nodes in the context, i.e. these may represent either ports or boxes.

3.1 Delta reduction rules

Delta reduction rules are external to the system; they are supplied together with constants, to be able to reduce constant expressions. For example if we are using constants for integer numbers and integer operators, we will assume the existence of delta reduction rules such as the one pictured in Figure 7; typically there will be an infinity of such rules, for all combinations of numbers to be added.

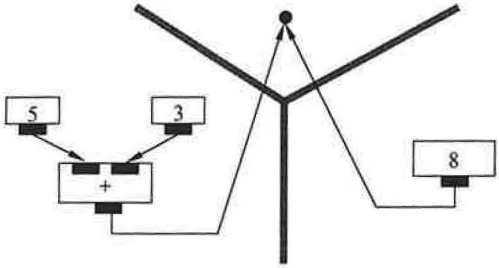

Figure 7 A delta reduction rule

3.2 Beta reduction rules

As explained above, delta reduction rules operate on constants at a distinct level. So the core of the dynamics of system comes from the beta reduction rules, which replace grey boxes by white boxes. This corresponds to the idea exposed in the introduction to have software chips which "travel in the system".

A *beta-redex* is a path from the border of a free white box to the border of a grey box, such that the white box has at least as many input ports as the grey box. The redex is *contracted* by

- replacing the grey box by a copy of the white box, joining the corresponding input and output ports
- removing the path

In the case of a grey box with n=3 input ports and a white box with $n+k$=5 input ports, this is expressed by the following Y-rule:

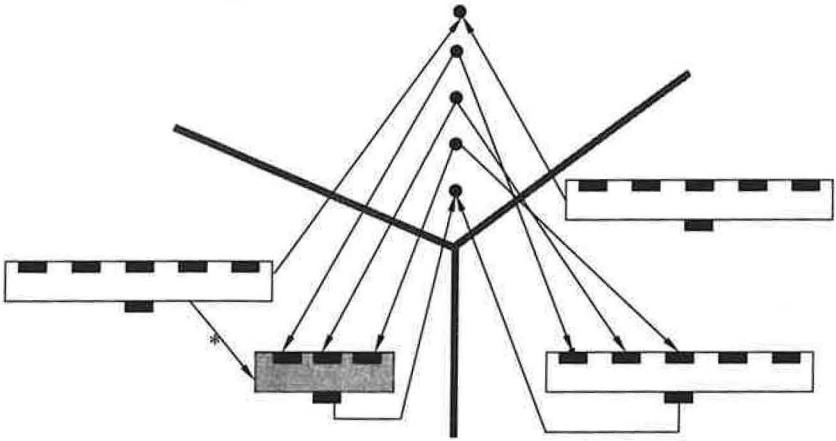

Figure 8 Beta reduction

The star '*' associated to the arrow from the white box to the grey box denotes that this connection can be any path, instead of just a single arrow. After the reduction step, a copy of the white box has been created, and is embedded in the graph by preserving all previous port connections of the grey box; the grey box itself has disappeared, together with its incoming path. The original white box remains, together with its other connections (if any).

3.3 Curry reduction rules

In case there is a path from a free white box with n input ports to a grey box with $n+k$ input ports, the beta reduction rule above cannot be applied. The following rule can solve such situations.

- a grey box with $n+k$ input ports ($n, k > 0$) can be split into two separate grey boxes with n and k input ports respectively, and with an arrow from the output port of the first box to the border of the second box.

Figure 9 pictures a curry reduction in the case where n=3 and k=2.

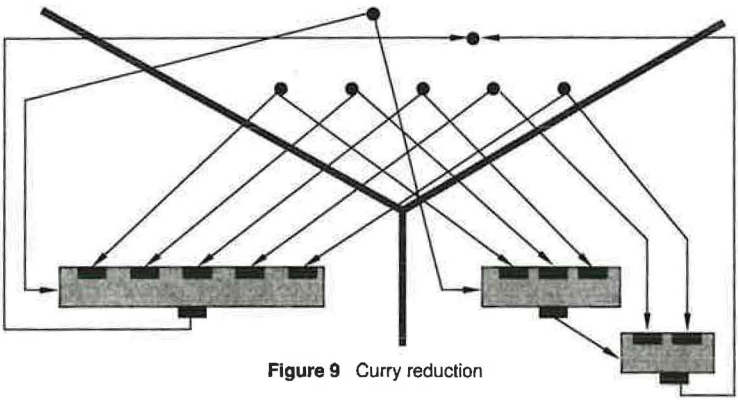

Figure 9 Curry reduction

3.4 Garbage collection

- a box without any outgoing arrow can be deleted
- let $a_1 \ldots a_n$ be a path with source s and destination d. If it is possible to draw a legal single arrow from s to d, i.e. without crossing any box boundary, then remove the path $a_1 \ldots a_n$ and add the arrow (s, d).

4. Examples

A simple example to illustrate our approach is to encode boolean values and boolean operations. Figure 10 displays such encodings. Values *True* and *False* are white boxes with two input ports, which connect either the first or the second input to the output. The conditional **if** a **then** b **else** c is a box with three inputs; we decorated them with corresponding labels to make the discussion easier. Input port a is fed into a grey box which receives connections from b and c. It is easy to

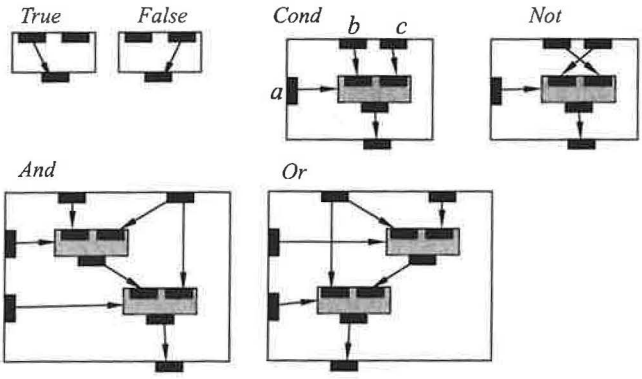

Figure 10 Encoding the Booleans

see that, depending of which boolean value is given at *a*, the circuitry will connect either *b* or *c* to the output port. Therefore this indeed corresponds to a conditional statement which selects either its left branch or its right branch. Similarly, one understands immediately how the *Not* box, which is very similar to *Cond* except that the connections are reversed, transforms *True* into *False* and vice-versa.

The encoding of *And* and *Or* operations is a little more complex. To help understanding it, and to illustrate the reduction rules in action, we will follow the reduction steps for the expression *False Or (Not False)*. The corresponding initial graph is pictured in Figure 11.

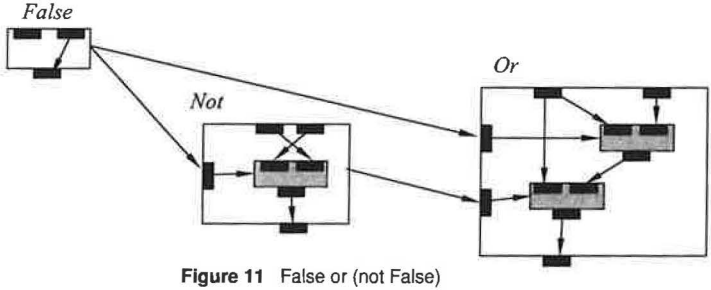

Figure 11 False or (not False)

In this initial graph there are to Beta-redexes: two paths starting from the *False* box, with grey boxes as destinations. Note that the path from the *Not* box to a grey box within *Or* is not a redex, because the *Not* box is not free (it has an incoming arrow on its first input port). After contracting both redexes, and performing appropriate garbage collection (for example deleting the *False* box, since no other path starts from it), the situation is the one of Figure 12: Now the *Not* box is free, and we have a new Beta-redex from *Not* to a grey box. Contraction of this redex yields a final graph in Figure 13: and at this point it is clear that the result is a box which behaves exactly like *True*, i.e. it is a box with two inputs, which connects its first input to the output. As a matter of fact, the garbage collection rules allow us to rewrite this result exactly as *True*

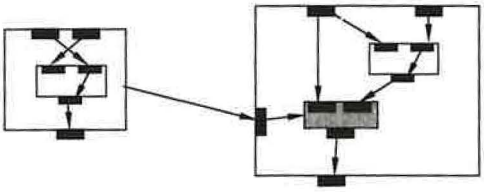

Figure 12 False or (not False), second step

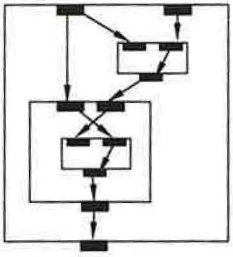

Figure 13 Final result of (False or (Not False))

5. Translation to Lambda Calculus

To anybody with some knowledge of the lambda calculus, the examples above are not surprising, since they are a direct translation of the usual Church encoding of boolean values:

True	=	λxy. x
False	=	λxy. x
Cond	=	λxyz. x y z
Not	=	λxyz. x z y
And	=	λlrxy. r (l x y) y
Or	=	λlrxy. r x (l x y)

In general, any box graph has one corresponding lambda term. In the reverse direction, every closed lambda term has several corresponding box graphs; however these are all equivalent modulo garbage collection rules. To make this correspondance explicit, we give in this section two translation algorithms.

5.1 From box graphs to lambda terms

We translate box graphs into an applied lambda calculus with constants. By assumption, every kind of primitive box of the graph language has an associated constant in the target lambda calculus.

A box graph is *closed* iff its outermost box is a composite box without any input port. Initially the output port of that box will be the *current port* for the algorithm; by the well-formedness rules, this port must have exactly one incoming arrow. The algorithm proceeds in two steps:

- **step 1**: label every input port of every composite white box with a distinct name
- **step 2**: the current port must have exactly one incoming arrow. Follow backwards that arrow, and, depending on its source, do the following:
  - if the source is a primitive box, print the constant corresponding to that box.
  - if the source is the output port of a composite box, then select that port as new current port, and go back to step 2. Note that the well-formedness rules ensure that the new current port indeed has exactly one incoming arrow.
  - if the source is the border side of a composite box, then this box must have input ports without incoming arrows. Let $x_1 \ldots x_n$ be the corresponding labels. Write the string "($\lambda x_1 \ldots x_n$."; recursively go to step 2 with the output port of that box as new current port; and finally close the expression with ")".
  - if the source is an input port, either print the label of that port if the port has no incoming arrow, or select that port as new current port and recursively go to step 2.
  - if the source is the output port of a grey box, then successively apply the algorithm to the incoming arrow on the border of the grey box, and then to incoming arrows on each input port.

5.2 From lambda terms to box graphs

Let *a* be a closed lambda term. Case *a* of

- $\lambda x_1 \ldots x_n$. a' : write a new composite box with *n* input ports; take the output port of that box as new target; recursively apply the algorithm to *a'*.
- x_i : draw an arrow from the the *n*th input port to the current target
- $(a_0 \ldots a_n)$: write a new grey box with *n* input ports; connect its output port to the current target; take the border of the grey box as new target and recursively apply the algorithm to a_0; then, for each *i* between 1 and *n*, take the *i*th input port of the grey box as new target and recursively apply the algorithm to a_i.

6. VisualLambda: implementation

In this section we describe the design and implementation of an interactive tool for editing box graphs. The tool supports hierarchical construction of graphs of arbitrary complexity, through a dual view of composite boxes: such boxes can be represented either as closed, encapsulated entities, which can be embedded in other graphs, or they can be opened up, in order to edit their internal definition. The tool allows users to work with multiple views, so it is possible to simultaneously edit a complex graph at different levels of detail.

6.1 Starting choices

Box graphs are graphs in the usual graph-theoretic sense. Formally, they could be described as graphs with several sorts of nodes (box nodes, input port nodes, output port nodes), and with several sorts of edges (arrow edges, port bordership edges, box inclusion edges). In consequence, it would seem that existing generic tools for graph edition could be adapted for box graph edition. However, after examining several of such tools, like Robochart [7], daVinci[8], XGrab or Graphed[9], it appeared that the adaptation task would not be easy. The reason is that the specific structure of box graphs involves some constraints and some visual presentation aspects which are quite different from generic graphs. For example. it is quite clear that the box inclusion relationship should not be represented visually as a collection of arrows, but rather as a true inclusion relationship between graphical shapes. Adding such graphical capabilities to the tools mentioned above would have been quite difficult.

The solution for avoiding to entirely write a new program was to adapt a generic drawing tool, already equipped for manipulating hierarchies of graphical shapes. The programming task, then, was to add the box graph structure, and to constrain the drawing primitives in order to ensure conformance with box graph formation rules. The selected drawing tool was *Draw*, a demonstration application supplied with the NeXTSTEP platform, which has the advantage of being written in an object-oriented environment, thereby making it easier to customize its general framework to a new functionality.

The core framework of VisualLambda consists of about 20 classes; most of them inherit from classes defined in the original *Draw* program. The class hierarchy is shown in Figure 14

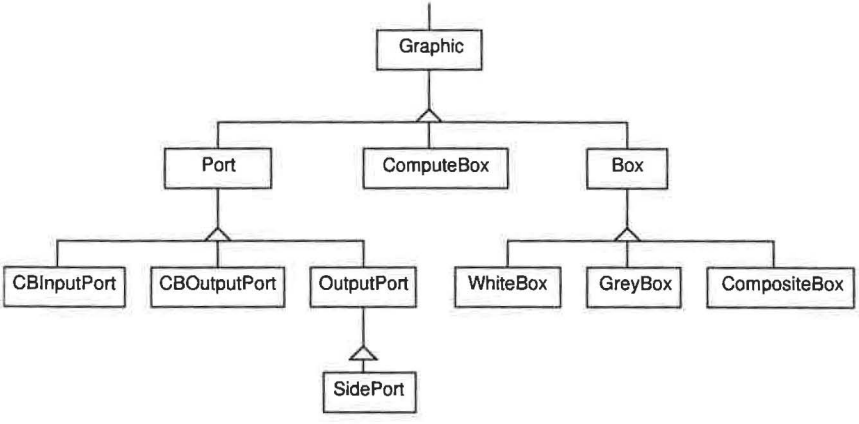

Figure 14 Partial Graphic class hierarchy.

and a class diagram in Figure 15. pictures the relationships between classes, using Booch notation. The most important of them are considered in more detail in the rest of this section.

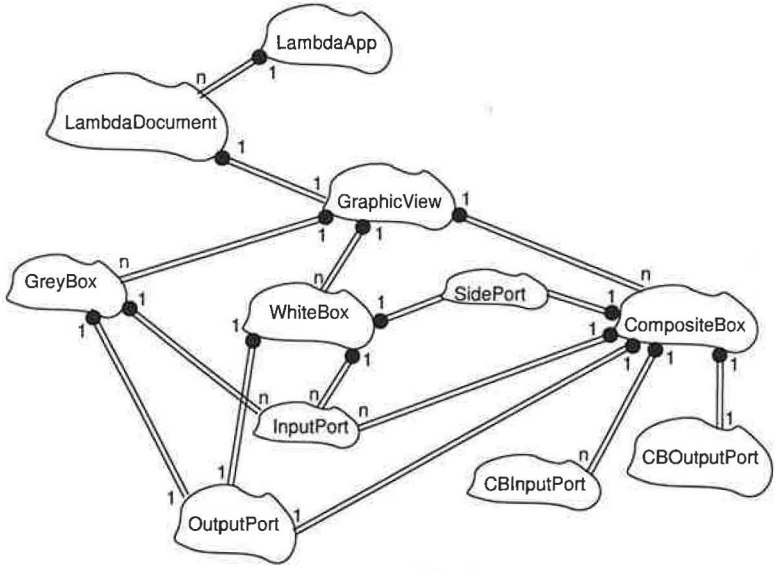

Figure 15 Booch Class Diagram of VisualLambda.

6.2 LambdaApp and LambdaDocument classes

As VisualLambda is a multi-document software, LambdaApp and LambdaDocument classes have the primary functions of creating and managing documents. Each document represents a piece of paper. LambdaApp manages and dispatches events from the application menu. LambdaDocument manages the representation of the view in which we create objects. This includes saving the view to disk.

6.3 Graphic class

Graphic is an abstract class for manipulating objects that appear on screen. All objects that can appear in GraphicView are subclasses of Graphic. By definition the main purpose of an abstract class is to define a common interface for its subclasses. This is why the vast majority of the graphics functionalities are contained in this class, e.g. object position, line width. We subclass Graphic for all the objects we need to have in our data-flow application, i.e. WhiteBox, GreyBox, CompositeBox, ResultBox and different kinds of Port. These objects have two basic responsibilities. They know (1) how to draw themselves, and (2) where they are. By using an object for each graphical element in the document, we promote flexibility. All objects are treated in the same way and we can extend VisualLambda without disturbing other functionalities.

6.4 GraphicView class

GraphicView is the heart of the program functionalities. It manages a display list in which are stored all the objects we draw. Such objects are subclasses from the abstract class Graphic. New kinds of graphic are simply created by subclassing and adding specific code.

The user is allowed to create objects, select them, move them around, group and ungroup them, change their font, cut and paste them to the pasteboard. One of our main preoccupation was to have good responsiveness from VisualLambda, because users do not want to wait when they draw or drag boxes. This is why we take care in our implementation to optimize the display in minimizing Postscript code sent to the display server. All the drawing is done in an off-screen window which is displayed back to the screen. This supports very fast redraw of areas obscured either by the moving object or the user's scrolling. Moving is accomplished by using a selection cache. The objects in the selection cache are drawn using opaque ink on a transparent background, so that when they are moved around, the user can see through them the objects that are not being moved.

6.5 Port class and its subclasses

Ports are small black rectangles bounds to boxes. Their mission is to keep the connections between boxes. Each box has one OutputPort from which it start its connection and one or many InputPort for receiving connections. Accordingly to section 2, Ports can have only one incoming, that is the reason why they can refuse a link if they are already connected.

During the connection between a source and a sink, users have graphical feed back about the validity of a connection. The drawing link transforms itself in an arrow if the pointer in on one of the valid destination zones described in section 2.

Others subclasses of Port are specializations. For example SidePorts are transparent and are on both sides of WhiteBoxes and CompositeBoxes.

6.6 ComputeBox class

ComputeBox is also a subclass of Graphic, but it has special properties. Its main job is to display the result of the graphical representation. Instances of ComputeBox have one InputPort and no OutputPort because they end the data-flow. The evaluation of the graphical program starts when the InputPort is connected. The box will launch a recursive parse of the structure and generates a Lambda expression (see section 7.7). Next the string is sent for interpretation through a pipe to a *Gofer* process who will send back the result. Another way to start the evaluation is to double-click on a already connected ComputeBox.

One of the most interesting features is that if a box is modified, the ComputeBox at the bottom of the data-flow is notified. It reparses automatically the new structure and evaluate the new expression according to the change. With this functionality VisualLambda works like an interpreter offering direct *"What-If"* feature. This is ideal for beginners and people wanting to experiment and see instantaneously their results.

As said previously, graphic programs are translated into Lambda expressions.

We think that it is important to allow the user to see the textual counterpart of his visual programs. That is why the translation result can be monitoring in a separate window, called the Expression Viewer. It is also possible to type directly an expression and evaluate its result. This service allows comparison of 2 expressions and offers another way to detect errors.

6.7 Implementation of Connections

In our implementation, connections between ports are not subclasses of Graphic, i.e. there is no corresponding graphical objects. Instead ports hold pointers to their destination(s) allowing them to request informations they need. For example, after a move, a box follows his connections, takes the screen locations of its destination ports and updates the connections. If the box is connected it will ask its source to adjust the link according to its new location. At the beginning this design choice was made in a simplicity concern. But now, that the prototype is finished we though that representing connections by objects could offer more flexibility though adding complexity. Adopting this option should also use more memory and introduce small slowness, especially in the drawing, due to an augmentation of message send.

7. Conclusions

Extensions may be required in the following areas:

- support more complex programs in WhiteBoxes,
- the creation of a container working as a library of the most often used basic elements. Archived components should be easily inserted in other program e.g. by dragging.
- implement rewrite rules in the tool.

The links between boxes must be improved with the goal of increasing the visualization of sources and destinations. One way of doing that should be to name more explicitly the ports. The actual design is open enough to allow such modifications without to big efforts. Finally, a way of simplifying the graphical representation of the CompositeBoxes encapsulation should be thought. Sometimes, and especially when the program is constituted of a lot of imbricate boxes, it could become quite difficult to represent the whole program structure due to the fact that a lot of windows are simultaneously open.

References

[1] Jorg Poswig, Guido Vrankar and Claudio Morara, *VisaVis : a High-order Functionnal Visual Programming Language*, Academic Press Limited, 1994.

[2] P. T. Cox, F. R. Giles & T. Pietrzykowski, *Prograph: A step forward liberating programming from textual conditionning*, Workshop on Visual Language, 1989, Rome, Italy, pp. 150-156.

[3] J. R. Rassure & C.S. Williams, *An integrated data flow visual language and software development environment*, Journal of Visual Languages and Computing 2, 1991, p. 217-246.

[4] Herbert Goettler, *Graphgrammatiken in der Softwaretechnik,* Informatik-Fachberichte 178, Springer-Verlag, 1987.

[5] C. M. Holt, *viz: A Visual Language Based on Functions*, 1990 IEEE.

[6] Marc A. Najork and Eric Golin, *Enhancing Show-and-Tell a polymorphic type system and higher-order functions*, 1990 IEEE.

[7] Robochart from Digital Insight.

[8] *daVinci* from University of Bremen, Deutschland.

[9] M. Himsolt, *Graphed*, University of Passau, Deutschland.

[10] Victoria de Mey, *Visual Composition of Software Applications*, PhD Thesis,University Of Geneva, 1994.

[11] Greg Michealson, An introduction to functional programming through Lambda Calculus. *International Computer Series.*

[12] ACM Sigplan Notices, *Haskell Special Issue*, Volume 27 Number 5 May 1992.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series.

[14] Laurent Dami, *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*, PhD Thesis, University of Geneva, 1994.