(a) Box Elaboration

(b) Memory Box Constraints

(c) Operation Box Constraints

Figure 4.1.5: Local Constraints
(a) Elaboration of $T = \text{XOR}(F; T)$

(b) Elaboration of $F = \text{XOR}(T; T)$

Figure 4.1.6: Elaborations of XOR
Figure 4.1.7: Boxgraph Representation of a Full Adder
4.2 Recursive Boxgraph

The computational power of the model presented in the previous section is limited to that of Boolean circuits or that of Propositional Logic. In this section we introduce naming, representation of numbers, the successor operation, and the predecessor operation, into the boxgraph model. We call the extended model a recursive boxgraph.

A name of a boxgraph is any icon (bitmap) of any size. Any rendering of a text can be used as a proper name of a boxgraph. A declaration of a name and its denotation is represented by a meta-expression connecting the name icon and a boxgraph by an `:=` sign. A recursive definition of a boxgraph is allowed; a box may contain either nothing, a trivial

![Diagram of Name Declaration]

(a) Name Declaration

![Diagram of Using the Named Operation Box - I]

(b) Using the Named Operation Box - I

![Diagram of Using the Named Operation Box - II]

(c) Using the Named Operation Box - II

Figure 4.2.1: Naming Boxgraphs
boxgraph, a non-trivial boxgraph, or a name of a boxgraph. In Figure 4.1.1, 'T' and 'F' are used as the names of two trivial boxgraphs. When a box contains the name of an operation box, we equate the frame of the named operation box with that of the box containing the name. For example, if we name the operation box of Figure 4.1.4(a) by '^' as in Figure 4.2.1(a), then Figures 4.1.4(b) and 4.1.4(c) can be represented by Figures 4.2.1(b) and 4.2.1(c), respectively.

In the recursive boxgraph model, we represent the set of natural numbers by a tally system where the number zero is represented by an empty box and the successor of the number n is represented by a box containing the representation of the number n. There are only two primitive number operation boxes; the successor box and the predecessor box, named by '^' and '↓', respectively. The semantics of the primitives are illustrated in Figure 4.2.2. Note that the predecessor box is inconsistent if its input is zero.

(a) Number Definition

(b) Successor Function

(c) Predecessor Function

Figure 4.2.2: Arithmetic Primitives
The standard set of arithmetic operations are defined in Figure 4.2.3. Let $a^\uparrow$ denote the successor of the natural number $a$. Let $a^\downarrow$ denote the predecessor of $a$. Let $a$ correspond to the first input (upper left) to an operation and $b$ correspond to the next input. Then Figure 4.2.3(a) shows the addition operation implemented as follows:

$$+(a,b) \Rightarrow \begin{cases} a & \text{if } b = 0 \\ +((b^\downarrow,a))^\uparrow & \text{else} \end{cases}.$$

Notice how if $b$ is 0, the 0 in the open box stays consistent, thus allowing the $a$ value to flow through to the output of the box and allowing the 0 to flow into the open box containing 1. This open box becomes inconsistent and propagates that inconsistency to stop the recursive call to the + operation. Figure 4.2.3(b) show the multiplication operation implemented as follows:

$$x(a,b) \Rightarrow \begin{cases} 0 & \text{if } a = 0 \\ +(x(a^\downarrow,b),b) & \text{else} \end{cases}.$$

The symmetric difference operation, $\neq$, is defined as $-(a,b) = |a - b|$, where the minus sign in the absolute value expression is normal subtraction. Figure 4.2.3(c) shows this operation implemented as follows:

$$-(a,b) \Rightarrow \begin{cases} b & \text{if } a = 0 \\ a & \text{if } b = 0 \\ -(a^\downarrow,b^\downarrow) & \text{else} \end{cases}.$$

Figures 4.2.3(d) through 4.2.3(g) show some comparison operations. If the result of the comparison is true, then the operation boxes remain consistent. If the result is false, then the operation boxes become inconsistent. The 'Less Than or Equal' operation shown in Figure 4.2.3(c) can be understood as follows.

We are trying to determine whether $a \leq b$ is true. If $a$ is 0, then the comparison must be true. Otherwise, if $b$ is 0, as determined by an attempt to apply the predecessor operation to $b$, then the comparison is false. Lastly if $a$ and $b$ are both non-zero, we make a recursive call to determine whether $a^\downarrow$ is less than or equal to $b^\downarrow$. 
(a) Addition

(b) Multiplication

Figure 4.2.3: Arithmetic Operations
(c) Symmetric Difference

Figure 4.2.3: Arithmetic Operations
Figure 4.2.3: Arithmetic Operations
Recursive definitions of the Fibonacci function and the GCD function are given in Figure 4.2.4. Figure 4.2.4(a) shows the Fibonacci function implemented as follows:

\[ \text{fib}(a) \Rightarrow \begin{cases} 1 & \text{if } 1 < a \\ + ( \text{fib}(a^{\uparrow}), \text{fib}(a^{\downarrow}) ) & \text{else} \end{cases} \]

Figure 4.2.(b) shows the GCD function implemented as follows:

\[ \text{gcd}(a,b) \Rightarrow \begin{cases} a & \text{if } a = b \\ \text{gcd}( -(a,b), a ) & \text{else} \end{cases} \]
(a) Fibonacci Function

(b) GCD Function

Figure 4.2.4: Fibonacci and GCD Functions
5. Completeness of Recursive Boxgraph Model

In this section we present our proof of the computational completeness of the model. We will show that the recursive boxgraph model can be used to construct programs which emulate a computational model already shown to be computationally equivalent to Turing machines. The model used is the program machine of Minsky [31].

5.1 Program Machines

As an alternative to the Turing machine model of computation, Minsky presents the program machine and shows that for any Turing machine there is an equivalent program machine. A program machine has a finite number of infinite capacity registers,

![Diagram of a program machine]

Figure 5.1.1: Structure of a Program Machine
denoted a, b, c, etc. Each program machine also has a finite set of operations which can affect or detect the contents of the registers. Each operation can access only one register at a time. The number of operations which a machine can perform is usually quite small, about 3 or 4. A program machine also contains a programming unit which contains a numbered sequence of instructions called a program. Each instruction contains an indication of the operation to be performed, the register on which to perform the operation, and (possibly) the number of another instruction in the program. Figure 5.1.1 shows the structure of a program machine.

It is worth noting that program machines are not "stored-program" computers. The program for any given machine is "built into" the programming unit of the machine. No instructions can manipulate the program.

An example program machine will illustrate the concepts presented above. Our example program machine has three registers (a, b, and w) and is capable of the three operations shown in Table 5.1.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Zero&quot;</td>
<td>a^0</td>
<td>Set the content of register a to 0. Go on to the next instruction.</td>
</tr>
<tr>
<td>&quot;Successor&quot;</td>
<td>a'</td>
<td>Add 1 to the content of register a. Go on to the next instruction.</td>
</tr>
<tr>
<td>&quot;Decrement or Jump&quot;</td>
<td>a^-n</td>
<td>If content of a is not zero, decrease by 1 and go on to next instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If content of a is zero, jump to the nth instruction.</td>
</tr>
</tbody>
</table>
The letter a in the notations given in the table could be replaced by the letter corresponding to any other register, and then the operation would affect the designated register.

The program for our machine is given in Table 5.1.2. The instruction number corresponds to the Program Counter of digital computers. Execution begins with instruction number 1. Unless a jump operation is executed, the next instruction to be executed is the one with a line number one greater than the current instruction. Execution halts when there is no next instruction to be executed. In the case of this example, when the next instruction to be executed is instruction 8 (which does not exist), the machine halts.

<table>
<thead>
<tr>
<th>Instruction Number</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b₀</td>
</tr>
<tr>
<td>2</td>
<td>w₀</td>
</tr>
<tr>
<td>3</td>
<td>a⁻(7)</td>
</tr>
<tr>
<td>4</td>
<td>b'</td>
</tr>
<tr>
<td>5</td>
<td>b'</td>
</tr>
<tr>
<td>6</td>
<td>w⁻(3)</td>
</tr>
<tr>
<td>7</td>
<td>w₀</td>
</tr>
</tbody>
</table>

Table 5.1.2: Program for Example Program Machine

Our example program will calculate twice the initial content of register a and leave the result in register b. To see this, we can trace the operation starting with 2 in a and anything in b and w. The trace is shown in Table 5.1.3. Instruction number 6 sets up a loop. Each time around, the contents of register a are reduced by 1 and the contents of register b are increased by 2. Thus when a at last contains 0, b contains twice the original content of a. Notice that register w serves only as a device for getting the program to jump to another instruction.
<table>
<thead>
<tr>
<th>Instruction Number</th>
<th>Effect</th>
<th>Register contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>initial state of registers</td>
<td>a: 2</td>
</tr>
<tr>
<td>1</td>
<td>set b to 0</td>
<td>b: 0, w: ?</td>
</tr>
<tr>
<td>2</td>
<td>set w to 0</td>
<td>b: 0</td>
</tr>
<tr>
<td>3</td>
<td>content of a is non-zero, so</td>
<td>a: 1</td>
</tr>
<tr>
<td></td>
<td>subtract 1 from it and go on</td>
<td>b: 0</td>
</tr>
<tr>
<td>4</td>
<td>add 1 to content of b</td>
<td>b: 1</td>
</tr>
<tr>
<td>5</td>
<td>add 1 to content of b</td>
<td>b: 2</td>
</tr>
<tr>
<td>6</td>
<td>content of w is 0, so go back to instruction 3</td>
<td>b: 2</td>
</tr>
<tr>
<td></td>
<td>content of a is non-zero, so</td>
<td>a: 0</td>
</tr>
<tr>
<td></td>
<td>subtract 1 from it and go on</td>
<td>a: 2</td>
</tr>
<tr>
<td>4</td>
<td>add 1 to contents of b</td>
<td>a: 3</td>
</tr>
<tr>
<td>5</td>
<td>add 1 to contents of b</td>
<td>a: 4</td>
</tr>
<tr>
<td>6</td>
<td>content of w is 0, so go back to instruction 3</td>
<td>a: 4</td>
</tr>
<tr>
<td></td>
<td>now content of a is 0, so go to instruction 7</td>
<td>a: 4</td>
</tr>
<tr>
<td>7</td>
<td>set w to 0</td>
<td>a: 4</td>
</tr>
<tr>
<td></td>
<td>ending state of registers</td>
<td>a: 4</td>
</tr>
</tbody>
</table>

Instruction numbers serve only to indicate the relative position of instructions in a program. We can leave them out of our program notation if we provide some other means of indicating the destination of a 'jump' operation. We can use arrows for that purpose. Figure 5.1.2 illustrates our new notation and shows the creation of a subroutine named 'double' which has the same effect as the program listed in Table 5.1.2.
Using this definition of a program machine and the construction of subroutines as illustrated above, Minsky shows that, for any Turing machine, a program machine can be constructed which computes the same function as the Turing machine. Such a program machine needs only five registers and uses only the two operations 'Successor' and 'Decrement or Jump'. He also shows how, by using prime-factorization of integers, a single register, a, can be used to simulate three registers, x, y, and z, by storing $2^x3^y5^z$ in a. Thus Minsky proves the following theorem.

For any Turing machine $T$ there exists a program machine $M_T$ with two registers that behaves the same as $T$. This machine uses only the 'Successor' and 'Decrement or Jump' operations.
5.2 Sequential Control Flow via Data Flow

In order to show that the recursive boxgraph model has the same computational power as the Turing machine, we intend to show that Minsky’s program machines can be constructed using the recursive boxgraph model. Since the boxgraph model is a data flow based model, we need some method of emulating sequential flow of control using data flow. This section presents that method.

Consider the structure of a program machine shown in Figure 5.1.1. The sequential execution of the set of instructions shown and the ability to identify an instruction by its relative place in the set are essential parts of the program machine model. Suppose we change the method for determining the order of execution of instructions in the programming unit from sequential execution to data flow based execution. How then can we enforce a sequential execution of instructions which in turn allows the assignment of instruction numbers? Figure 5.2.1 shows the extra structure which can be added to accomplish that goal.

Notice that the arrows inside the programming unit are data flow arrows and the numbering of instructions is a direct result of the data flow dependencies created by those arrows. In fact, the instruction numbers are no longer necessary because the order of execution is determined by the data flow dependencies. These data flow dependencies form a linear order on the set of instructions. On the other hand, the arrows between the programming unit and the registers indicate that each instruction still has reading and writing access to the set of registers. These arrows can not be data flow arrows in the sense that they are used in the boxgraph model because they create cycles in the graph. That problem is solved in the next section.
5.3 Maintaining Register State via Data Flow

We need to allow each instruction access to the registers of the machine without creating data dependency cycles. In other words, we need to make memory access an acyclic process. This is accomplished by passing the state of the registers from one instruction to the next. This is only possible since a program machine has a finite number of registers. Figure 5.3.1 illustrates how this is done. Each register has a box to which its value arrives from the previous instruction and a box from which its value is passed to the next instruction. This allows the instruction to 'read' the value, manipulate it as necessary, and then pass a new, possibly different, value on to the next instruction. We have essentially moved the registers into the instructions of the machine. The data flow
dependencies established by the arrows which pass register values from instruction to instruction form a linear order on the set of instructions. This makes the previously established linear order redundant. However, we continue to include that construct in the figure because we will build upon it in the next section in which we show how recursion is used to emulate a jump operation.

Figure 5.3.1: Passing Register State using Data Flow
5.4 Using Recursion to Emulate a Jump Operation

In this section we show how recursion can be used in a data flow model to emulate a jump operation. The idea is that whenever a jump is necessary, say from instruction $m$ to instruction $n$, then we recursively call the entire program. We pass to the recursively called program the number of the instruction to which we want to "go", namely $n$. The program must then ignore all instructions prior to instruction $n$. That is, those instructions prior to instruction $n$ in the sequence must pass the register values without making any changes. From this point on, the recursively called program executes normally, possibly making further recursive calls. Upon returning from a recursive call at instruction $m$, instructions $m+1$ to the end of the program must also be ignored. Figure 5.4.1 illustrates these ideas for both a forward jump ($m < n$) and for a backward jump ($m > n$). Instructions which are in the cross hatched areas of the program are not executed. By 'not executed' we mean that the set of register values is passed through such instructions unchanged. In both the forward jump and backward jump cases, the instructions which are executed are those from the beginning of the program up to and including instruction $m$ and those from instruction $n$ to the end of the program. Figure 5.4.1(a) illustrates how this leads to the skipping over of the instructions from $m + 1$ to $n - 1$. Figure 5.4.1(b) illustrates how this leads to the repetition of instructions from $n$ to $m$. The figure shows how instructions would be executed for one level of recursion. The same constructions will work for multiple levels of recursion.

We need to accomplish this type of instruction execution using data flow. To do this we will pass three more values (in addition to the register values) from instruction to instruction. These three values are called the start number, the line number, and the stop bit. Figure 5.4.2 illustrates how these control values are incorporated into the program.
Figure 5.4.1: Recursive Jump Operation from \( m \) to \( n \)

\( \square \) = instructions not executed
Figure 5.4.2: Passing Control Values using Data Flow
The start number, line number, and stop bit are labeled s#, l#, and sb respectively. Each instruction has access to these control values similar to its access to the register values. The operation part of each instruction, that is, the part which affects the state of the registers, will not be executed if the start number is less than the line number. Similarly the operation will not be carried out if the stop bit value is not zero. Regardless of whether the operation is carried out, each instruction increments the line number value before passing it on to the next instruction. We begin the program with the start number set to 1, the line number set to 1, and the stop bit set to 0. The initial register values are set as necessary. If an instruction encounters a condition in which a jump to instruction n is necessary, then a recursive call to the entire program is made with the passed start number set to n, the passed line number set to 1, and the passed stop bit set to 0. Upon returning from such a recursive call, the stop bit is set to 1 before passing it along to the next instruction. This effectively disables the rest of the instructions.

Figures 5.4.3 through 5.4.6 show the trace of the execution of a program machine which calculates the same function as the program given in Table 5.1.2. This version of the program machine uses data flow to determine the order of instruction execution; includes the passing of control values to determine whether or not an instruction is to be executed; passes register values from one instruction to the next; and uses recursion to emulate jumping. Any instruction which is cross hatched with diagonal lines going down from right to left is not executed because start number is greater than line number. Any instruction which is cross hatched with diagonal lines going down from left to right is not executed because the stop bit is not zero.

It remains to be shown that operations which behave as shown in Figures 5.4.3 through 5.4.6 can be created using the constructs available in the recursive boxgraph model. That is the subject of the next section.
Figure 5.4.3: Trace of Program Machine Execution
Recursion Level 0
Figure 5.4.4: Trace of Program Machine Execution
Recursion Level 1
Figure 5.4.5: Trace of Program Machine Execution  
Recursion Level 2
Figure 5.4.6: Trace of Program Machine Execution
Recursion Level 3
5.5 Data Flow Program Machine Operations

Figures 5.5.1 through 5.5.3 show the recursive boxgraph constructions of the Successor, Zero, and Decrement or Jump operations. In each case the version of the operation which affects register a is shown, but these operations can be modified to affect any one of the other registers. The operations can also be modified to include any finite number of registers. They are shown with three registers because our example program uses three registers.

On the left of each boxgraph is the control section consisting of the three control values, start number, line number, and stop bit. To the right is the section of the boxgraph in which 'register' values are passed. The center of the graph is where register value manipulation occurs.

The Successor operation and the Zero operation (Figures 5.5.1 and 5.5.2) are almost identical. Since neither of these operations can result in a jump, there is no need for any recursive calls. If the start number is not less than or equal to the line number, then the '<=' box becomes inconsistent. This inconsistency propagates to prevent the changing of register values. Similarly, if the stop bit value is not zero, inconsistency prevents the changing of register values. In the case of the Successor operation, if the status of the control values allows, the incoming a value is incremented by 1 before passing it on. The zero operation ignores the incoming a value and passes zero as the new value. The line number value is always incremented before passing it on.

The Decrement or Jump operation tests the control values in the same way the other two operations do. The register manipulation section of this operation is divided into the decrement section, a box containing 1 and a box containing a decrement operator, ↓, and the jump section, which contains a recursive call represented by a box containing 'prog'. If
Figure 5.5.1: The Successor Operation – $a'$

Figure 5.5.2: The Zero Operation – $a^0$
Figure 5.5.3: The Decrement or Jump Operation – $a^{-}(n)$
the control values allow, an attempt is made to decrement the incoming a value. If this value is non-zero and therefore can be decremented, then the jump section of the graph is made inconsistent by a 1 from the decrement section flowing into an open box containing a zero in the jump section. If the incoming a value is zero, then the decrement operation box, \( \downarrow \), becomes inconsistent. This inconsistency allows the jump section of the graph to stay consistent and execute. In the Jump section, a recursive call is made. The number of the instruction to which to jump, \( n \), is passed as the start number for the recursive call. The incoming register values are also passed to the recursive call. Upon returning from the recursive call, the new register values are passed to the next instruction and the stop bit is set to 1. Setting the stop bit to 1 prevents any further manipulation of the returned register values.

5.6 Proof Conclusion

In the preceding sections, we have shown that the recursive boxgraph model can be used to construct operations which behave like the instructions in a program machine. We have created those operations for a three register, three operation machine. By the theorem presented in section 5.1, we need only use two registers and two of the three operations in order to match the computational power of the Turing machine. Therefore, the recursive boxgraph model is computationally complete, i.e. as powerful as the Turing machine.
6. Oberon Implementation

We have implemented a visual programming language called Simple Hyperflow. Simple Hyperflow consists of the Basic Boxgraph model described in section 4.1 plus an iteration box. The iteration box is a means of horizontal abstraction which was introduced in the Show and Tell Language [27]. An iteration box is denoted by an operation box with a thick frame as in Figure 6.1(a). It represents an abstraction of folding the horizontally spreading boxgraph of Figure 6.1(b) into one place. When an iteration box is executed, it is unfolded dynamically until the operation box becomes inconsistent. Simple Hyperflow does not provide the vertical abstraction mechanism of naming which is found in the Recursive Boxgraph model. Without naming, no recursion is possible.

![Iteration Box Syntax](image1)

(a) Iteration Box Syntax

![Iteration Box Semantics – Unfolding](image2)

(b) Iteration Box Semantics – Unfolding

Figure 6.1: Horizontal Abstraction
We started the implementation of Simple Hyperflow with a number of goals. Besides the obvious goal of creating a working version of the Simple Hyperflow language, we also wanted to be sure to use the pen as the means of interacting with the system. We wanted the system to be implemented on a widely available hardware platform and be written in an object oriented manner so as to allow for easy extensibility. We also wanted the implementation language to be one for which a very compact and efficient compiler was available. This last goal reflects our belief that users of pen-based notebook computers are going to expect good performance on systems which are limited in memory and processing power due to their size.

The platform chosen was the Sun SPARCstation IPX running the SPARC-Oberon System created by Templ [38]. SPARC-Oberon is an implementation of both the Oberon System [45] and the programming language Oberon-2 [32,44] for SPARC processors. One of Project Oberon's over-all goals is to provide power and flexibility to users while using a small fraction of the computing power and storage capacity required by commercial operating systems and languages [46]. The Oberon compiler meets our compactness and efficiency requirements, and the Oberon-2 language allows object oriented development. The SPARCstation is a widely available and widely used, but a pen interface for a SPARCstation is not widely available. The pen interface was developed as part of the implementation.

We equipped the SPARCstation with a WACOM HD-648A pen tablet [42] and a Vigra VS10 video adapter card [41]. The output from the tablet is connected to the SPARCstation through a serial port. Integrating the pen tablet and video card into the SPARC-Oberon environment involved creating two small shared object libraries of code written in C, one library for retrieving input from the tablet's digitizer and another for communicating with the video card in order to display output on the tablet's screen. These libraries were then used by Oberon modules designed to allow easy access to the tablet.
One module was written for each of the two libraries. The approach taken in the design of these modules was to write only enough code in C to allow access to the tablet. Significant drawing algorithms were written in Oberon-2.

The current implementation of Simple Hyperflow includes a rudimentary graphics editor which incorporates a shape recognition algorithm [5]. The shape recognizer converts user inputs into the two primitives of the Boxgraph model, boxes and arrows. The editor allows the user to create, select, and delete groups of boxes and arrows. In addition, the user can save programs to an ASCII text file which can be read back into the system later. Provisions for dumping the screen to a TIFF [39] file have also been added.

Also included is a reduction system which is used to execute programs written using the system. In addition to the simple box type and the arrow, predecessor, successor, and iteration boxes have been implemented. Figure 6.2 shows a screen dump of a binary full adder program written using the system. Figure 6.3 shows the result of executing the program after supplying appropriate inputs. Figures 6.4 and 6.5 show similar “before and after” diagrams for the Fibonacci function. Iteration boxes are shown as boxes with a thick frame. Successor boxes and the predecessor boxes are shown containing an ‘s’ and a ‘p’ respectively. The current implementation of the iteration box assumes all inputs and outputs are connecting points for horizontal concatenation.

The system was created using an object oriented design. Using the type extension facilities of Oberon-2, the following hierarchy of types was designed. Indentation is used to show the type/extension relationship; comments are shown, in Oberon style, enclosed in ""(" and ")"".
Object
   LocatedObject
      Option
      Shape
         Line
            Arrow
            Rectangle
               HFBox
                  HFPredBox
                  HFSuccBox
               HFIterBox
            SortedListNode
            ArrowListNode
            OptionListNode
            TreeNode
               HFBoxTreeNode
               HFIterBoxTreeNode
      (* Used to implement option menu *)

      (* Hyperflow Box *)
      (* Predecessor Box *)
      (* Successor Box *)
      (* Iteration Box *)
      (* Abstract List Type *)
      (* Instantiation of Abstract List Type *)

      (* Abstract Tree Type *)
      (* Instantiation of Abstract Tree Type for Hyperflow Boxes *)
      (* Iteration Box Tree *)

The arrows are maintained by the system as a sorted list; the boxes are maintained in a tree in which the parent-child relationship corresponds to box containment.

Further information on the implementation is included in the Appendices.
Figure 6.2: Screen Dump of Full Adder Before Execution
Figure 6.3: Screen Dump of Full Adder After Execution
Figure 6.4: Screen Dump of Fibonacci Function Before Execution
Figure 6.5: Screen Dump of Fibonacci Function After Execution
7. Evaluation

7.1 Evaluation of Model

Our goal was to design the simplest visual programming language which is computationally complete. We have shown that the recursive boxgraph model is computationally complete. But is it simple?

Recursive Boxgraph uses data flow semantics and a directed graph syntax. Both of these constructs are relatively simple. We attempted to add as few concepts and constructs as possible to the model. In order to make the language complete, we added the concept of inconsistency, the predecessor and successor operation boxes, and naming and recursion to the model. Could any of these concepts be left out and still leave a computationally complete language? For example, notice that in the completeness proof, naming and recursion are only necessary in order to emulate the “jump” operation of Minsky’s program machines. It is conceivable that by removing naming and recursion from the model and allowing cycles in the directed graph syntax, with some suitable interpretation of the cycles, that we could implement the jump operation and thus create a different computationally complete language. The key to whether such a language would be simpler than the recursive boxgraph model lies in the “suitable interpretation” of cycles. Would this interpretation be necessarily so complex as to make the language difficult to interpret?

One suggested method of removing recursion is to alter our data flow program machine to include a control decoder which determines the order of execution of instructions. The control decoder would be programmed using the basic boxgraph model and would signal the instruction set, using data flow, as to which instruction is to be executed. After an instruction executes, the resulting set of register values would be passed back to the control decoder along with an indication of which instruction is to be executed
next. At this point, the control decoder and the set of instructions would have to be "reset" in some way and then re-executed.

One significant criticism of the Recursive Boxgraph model is that using recursion to emulate jumping makes the model an infinite control model as opposed to both the Turing machine and Minsky's Program machine models which use finite control. The control decoder method described above has the advantage of being a finite control model. However, it departs from a pure data flow model when it requires the resetting and re-execution of the control decoder and the instruction set and the maintenance of register values during the reset.

It is also possible that by replacing our current unary number representation with a binary method of number representation, the predecessor and successor operations could be built using the Basic Boxgraph model. These operations would then not have to be assumed as primitives of the computationally complete model.

In contrast to that possibility, it would be interesting to use abstractions of the successor and predecessor operations. These abstractions would be based on the observation that the successor and predecessor operations are actually wrapping and unwrapping operations respectively. The successor operation takes its input and simply wraps a new box around it. The predecessor operations unwraps an outer box from its input. An abstracted version of the successor operation could take multiple inputs and combine them by wrapping a new box around the entire input set. Similarly, an abstracted predecessor operation could remove the outer box around its input and output the resulting multiple boxgraphs.

The notion of simplicity is difficult to quantify. We realize that some may find the other types of visual programming systems, constraint based systems, programming by demonstration systems, or forms based systems, to be easier to understand than the data
flow based system we have created. We think that efforts similar to this one, to find the simplest such systems, would be beneficial.

Suppose we grant that the recursive boxgraph is the simplest computationally complete language. Then we must question the simplicity of the given proof. We believe that it can be successfully argued that the constructions of the $a'$ and $a'\langle n \rangle$ are not simple or easy to interpret. Are there simpler constructions of these operations? Is there a simpler proof?

7.2 Evaluation of Implementation

The current implementation of Simple Hyperflow is a significant first step in the implementation of the language Hyperflow on a widely available platform and in such a way as to be adaptable to future hand-held, pen-based systems. One of the contributions of this system is the creation of a pen interface for the Sun SPARCstation under the Oberon operating system.

As is true of every object oriented system with which the author is familiar, the design of the system allows for extension in the directions that the system creator anticipated, but extending such a system in unanticipated directions is generally difficult. Whenever someone tries to do so, new abstractions are discovered which call for a redesign of the type hierarchy. There are, however, a number of more concrete criticisms of the system that need to be made.

The current implementation does not include naming and recursion. It does however include the iteration box. The shape recognition aspects of the system make drawing good looking boxgraphs much easier than would otherwise be possible. However, the editing functions of the system are limited. As we can see in Figure 6.2, the menu of options for the system appears on the left hand side of the screen. This location
was chosen fairly arbitrarily. Experience using the system has lead to the conclusion that this placement is very poor for right-handed users, such as the author. When making a selection from the list of options, the right arm tends to cover the diagram. This placement is probably a good choice for left-handed users. For right-handed users, placing the list of options on the right of the screen would be better. Allowing the user to choose the location would, of course, be ideal.
8. Future Work

8.1 Future Work on the Model

There are some areas in which the presented model should be extended. The group of boxes in the model can be viewed as a communicative organization utilizing two modes of communication, 'telephone' (point-to-point) communication as modeled by the transmission of data over flow lines and 'broadcast' (to-whom-it-may-concern) communication as modeled by the propagation of inconsistency. Another communication mode, called 'posting' [23, 24], is not part of the boxgraph model. In posting, the sender of the information does not know who receives the posted message, but the receiver knows the location of the posted message. This mode of communication is embodied in the concept of a variable (memory cell) used in traditional programming languages. Adding such a communication mode to the boxgraph model will be future work.

The boxgraph model currently does not include provisions for a data type which is likely to be prevalent in the not too distant future, continuous data. As multimedia applications become a more significant genre of computer usage, processing of continuous data types, such as audio and video signals, will become more essential. Traditional message passing communication models are not suitable for that purpose. We believe that data flow based communication/computation models are better suited to deal with continuous data types. The extension of the boxgraph model to include such data types is also planned.
8.2 Future Work on Implementation

Again, many of the plans for the future of the implementation come in direct response to the criticisms of section 7.2. Naming and recursion need to be added to the system. Additional editing functions, such as moving and resizing shapes, grouping and ungrouping shapes, adding patterns to the frames of boxes and the lines of arrows, and zooming and scrolling, are also needed. Allowing the user to specify where the set of options should be displayed should also be included.

One addition that will make the system much more practical will be the addition of a character recognition algorithm to allow the user to input text-based data. Using such a system, more extensive box types could be designed allowing the user to enter text based programs which define the internal workings of the box.

8.3 Future of Visual Programming Languages

A recurring theme in the study of visual programming languages is that graphs, pictures, and diagrams are best at providing information about the relationships among objects. They are not necessarily the best way of providing information which is independent of an object's relationships with other objects. Of course, it is not always easy to classify what type of information is being provided. The information that an integer cell contains the number 5 could be thought of as relationship independent information or as information that shows the relationship that the cell holds to other integer cells, namely that it is greater than all cells containing 4 or less and less than all cells containing 6 or more. The best visual programming languages will be the ones that allow the both visual (2D and even 3D) representations of information along with textual (1D) representations and allow the user/programmer to tailor the method of representation to the problem domain.
Bibliography


Acknowledgments

I would like to thank my thesis advisor, Dr. Kimura, for all his time and help in the writing of this thesis. The computation model presented in chapter 4 is the result of joint research between Dr. Kimura and me, to which he was the major contributor. The denotational semantics of the model are the result of ideas from his individual research.

Thanks also to the other members of my thesis committee, Drs. Ken Goldman and Gruia-Catalin Roman. Their comments not only helped make this thesis better, but also made me better educated. Dr. Goldman was particularly instrumental in my decision to write a thesis. I thank him for his time and help when I was making that decision.

I would also like to thank my current employer, Dr. John Stern, in the Department of Psychology of Washington University, for the many ways he has supported my education. My colleagues in the Department of Psychology also deserve thanks for their support of my education. Thanks also should go to the members of both the PenLab and Kumon Group in the Washington University Department of Computer Science for taking time to answer my questions.

Most importantly, I would like to thank my wife, Tricia, who has put up with the stresses and long hours entailed by my extended graduate school career.