Wind: A Flow-based Programming Language

Joshua Weinstein

2018

Contents

0.1 Preface

This book is intended as an overview of the Wind programming language. It discusses the paradigm of flow-based programming, and the principles of it. The book discusses the advantages and disadvantages of flow-based programming, while then proceeding onto the syntax and usage of the Wind language. The book also, in detail, describes the C implementation of the language. The state management, the instructions, the translation of source code, and execution are all covered.

The Wind language is not a fully fledged, general purpose programming language. It is a language with a fundamental set of computation tools and components. Wind was developed with the following goals in mind.

Goals

1. An extremely light-weight language that is highly portable.
2. A programming language which does not use dynamic memory allocation.
3. A fluid, highly dynamically typed runtime.
4. A system that allows efficient transfers of immutable data.

The most unique element of Wind is that it is a "bare bones" language. It has no abstract syntax tree, no tokenizer, parser, or standard library. It reads and executes instructions directly from source code, which transition and alter several internal buffers. From a high level overview, these buffers pass around data to one another allowing the state of the data in the program to change. This choice of design permits the language to focus on manipulating and changing data, instead of managing and allocating resources.

Overall, the hope is for this book to serve as an inspiration or reference for computing and programming in a flow-based manner.

0.2 Dedication

This book is dedicated to my mother, Vida, who has guided me and raised me to be an incredible human being I am today. I would not be who I am today without her support.

0.3 About the Author

Joshua Weinstein Grew up in the sunny, green fields of Los Angeles, California. He went to the University of California Berkeley, where he studied Computer Science and Fine Art. He now works as a software engineer, specializing in distributed cloud systems and high performance search. In his spare time, he is an active contributor to the open source community.

Chapter 1

Flow-based Programming

Flow-based programming is a paradigm of programming that deals with the flow of data over one or more destinations. These destinations could in theory be any sort of structure. For this book, and the implementation of *Wind*, they are only considered buffers; fixed sized arrays controlled with pointers and indices.

This chapter will focus on an abstract perspective of flow-based programming. It will describe the fundamental concepts, such as nodes, data, and flows. It will also discuss the arrangements of how data flows can be constructed, and give some respective examples.

This chapter serves as an introduction to flow-based programming, and a preparation for the actual Wind language.

1.1 Data

In flow-based programming, data is considered any of the values or information that are computed or processed. Numbers, integers, strings, characters, lists, are all examples of data. Nearly all programming languages type the data that they contain and compute. At the raw level, data to a computer is simply sequences of 1's and 0's. Programming languages provide much more human readable forms of data. Integers or floats to hold numbers, strings to hold texts, and numerous data structures like a *list* to hold an ordered collection of data.

An important distinction in how data is typically assigned types is whether a type is mutable or immutable. An *immutable* type is one that cannot be altered or changed after it is created. In order to alter or change it, you must create a new copy of that immutable type from existing instances of the type. For example, an integer, or any number, is an immutable type. Numbers are created from operations performed on other numbers.

$$a = 1$$
$$b = a + 1$$

In the above example, a is a variable with the value of 1, while b is a variable with the value of the sum of a and 1. Neither of them can be changed. They might be able to be recreated under the same variable name, but this is not altering the existing value. Next, let's look at an example in C that deals with mutable and immutable data.

```
1 int a = 1;
2 int* b = malloc(sizeof(int));
3 *b = a;
```

Here, b is an integer allocated dynamically on the program's heap, while a is a stack allocated variable. The main difference between them is that, b is a *mutable* data value that can be changed freely after it has been created. Yet a is immutable. The last statement, in the third, line sets the value of b to a. This means b can be considered a mutable type.

However, a and b are not both integers. b is a *pointer* to an integer, specifically a block of heap memory the size of one integer. In comparison to dynamically typed languages, b can be thought of as a one element list. In flow-based programming, the same data types have both mutable and immutable characteristics. Which characteristics of data determine it's behavior is determined by the location and the *movement* of the data.

1.1.1 Data Movement

Generally, moving data in a program means the ownership of values and data are being passed around. Some languages define specific criteria for the movement of objects and memory, like C++. *Move semantics* usually refers to the process of moving dynamically allocated objects from one owner to another. Here is a simple example.

```
1 int* a = malloc(sizeof(int) * 2);
2 a[0] = 1;
3 a[1] = 2;
4 int* b = a;
```

The memory containing enough space for two integers, pointed to by a, is passed to b. After line four, both a and b point to or "hold" the same value. Accessing through either one yields the same behavior. Changing one of the integers of a changes it in b as well.

```
printf("a is %d %d\n", a[0], a[1]);
printf("b is %d %d\n", b[0], b[1]);
b[0] = 5;
printf("a is %d %d\n", a[0], a[1]);
```

Compiling this code will display the following in the terminal:

```
1 2
1 2
5 2
```

Moving the resource always involves sharing or copying a *pointer* or *reference* to some value. Movement never writes the value itself again in a new location. Thus, immutable data cannot be moved in the context of usual move semantics.

In flow-based programming, data is copied from one node or location to another. A reference to data is never moved from owner to owner. During that copy, the data is most often transformed in some way, such that the data written to a new location is not the same as the data in the old location. The following example illustrates how a flow-based program would transform and copy data.

Example: Two node transport

```
1
2 void enterNode(int* node, int arg) {
3       *node = arg;
4 }
5
6 void transferToNode(int* startNode, int* endNode, int modify) {
7       *endNode = *startNode + modify;
8       *startNode = 0;
9 }
```

Two functions above are shown, *enterNode* and *transferToNode*. Both of these functions demonstrate the acquisition and transfer of data between two nodes. It's important to note this is a very basic example. The nodes themselves here do not contain any information of how to "modify" data that they acquire and transfer. The *modify* parameter is simply used as an example of how the next node a data value is transferred to modifies each data point it receives. Take note again that no objects, pointers, or references enter the nodes, only literal data values.

1.2 Nodes and Locations

Nodes, destinations, and locations determine the path data will travel in a flow-based program. Nodes which receive, transform, and send data can take on an incredible amount of different forms. The form of a node is usually determined by the functionality or role it plays. While the structure of a node is quite variable, it must always abide by the following rules:

Rules of Flow Nodes:

1. *A node must possess some amount of readable and writable memory.*
2. *A node must have a valid memory address that can refer to itself.*
3. *A node must live longer than the data it handles.*

The first rule is the least ambiguous. All nodes must be containers, and be able to some non-zero amount of data. Specifically, nodes must own either an array or a pointer to a block of memory. If a node has to transport data along some path, it must be able to hold data temporarily at certain steps. For example, an `enum` value would not be a valid node, as even though it evaluates to an integer, it is not a container, nor does it possess the ability to own a block of memory.

Rule two is quite simple, a node must exist or be allocated in addressable memory. A *register* is not addressable, therefore a register cannot be a valid node. Constants defined with preprocessor statements like `#define` do not have a memory address and therefore cannot be used as nodes. Compile-time constants and values can be used as nodes if they are permitted for modification during runtime.

The last rule ensures that nodes will be able to support, transform and move data from the start of the program to the end of it. The key idea behind flow-based programming is that, nodes are the supporting component of data flow. Nodes need live and be reusable throughout data flowing through the program. Specifically, nodes should not be created dynamically as data is intended to be transported. When data enters a flow, all the nodes should already be initialized and ready to pass along data.

1.2.1 Node Memory

As explained earlier, nodes must possess or own modifiable memory. Nodes can store the incoming data they acquire in a variety of ways. An important factor to consider in how a node uses memory is what kind of data the node deals with. Further, if the data is all the same type, or if the node accepts multiple data types in the same pass.

First, let's look at a node which only holds a single value, and how it can be initialized, and deinitialized:

```
1  #include <string.h>
2
3  typedef struct
4  {
5          void* data;
6          size_t size;
7          int occupied;
8  } SingleNode;
9
10 void SingleNode_init(SingleNode* node, size_t size) {
11          node->size = size;
12          node->data = malloc(size);
13          node->occupied = 0;
14 }
15
16 void SingleNode_deinit(SingleNode* node) {
17          free(node->data);
18          memset(node, 0, sizeof(SingleNode));
19 }
```

Let's break down the above code. First, we see the `struct` describing the attributes of the node. It has a `void` pointer to a block of memory, a field indicating the size, and a field acting as boolean to check if the node currently possess a data value. This implementation allows any C type to be placed and written into the memory of the node. However, in order to access the data we

must cast the data pointer as some C type. While this isn't problematic, it urges the need for the types of data to be tracked externally to the node.

In this setup, initializing a node requires that the node itself, the scalar type, already be declared and stored somewhere. The pointer to the node body is passed in, initializing it with it's memory and size. Then, once the node is done being used in the program, or needs to be resized, it can be de-initialized, by freeing the memory and clearing the data in the node's body. We want to permit the node itself to be stored in a centralized location, rather than allocating the node itself with a call to `malloc()` each time it is initialized.

A downside to this approach is that it decouples what is essentially the metadata of the node, and the data of the node. It requires two steps to have a fully functional node. Next, let's look at nodes that combine both it's data and information in a single scalar body.

```c
#include <stdlib.h>

typedef struct
{
    size_t capacity;
    size_t len;
    unsigned char data[];
} BlockNode;

BlockNode* BlockNode_new(size_t size) {
       BlockNode* node = malloc(sizeof(BlockNode) + size);
       node->capacity = size;
       node->len = 0;
       return node;
}
```

Note: The above uses a flexible array parameter, a special array available since C99. Flexible array parameters must always be the last members of a struct, and cannot be directly used with the sizeof operator.

BlockNode is a node that uses a dummy pointer, also called a flexible array, to refer to a variably sized body of memory it carries at the end of itself. Calling `sizeof()` on BlockNode will not work directly, as every newly constructed node of this type exists with a differently sized block of memory. Therefore the size of a BlockNode is given by the sum of it's `capacity` property, and the sizeof operator. Likewise, the `len` property indicates the amount of space occupied in the node's memory body segment. This makes the BlockNode slightly more useful than the SingleNode previously discussed, as we can store differently sized data in the same node.

The *len* property also makes it fairly easy to write data into the node, as well as expand the size of the node if and when needed.

```c
// Wrapped in () to prevent macro bloat errors
#define BLOCKNODE_SIZEOF(node) \
             (sizeof(BlockNode) + node->capacity)

```

```
5  #define BLOCKNODE_EXPAND(node, extraSpace) \
6                do { \
7                node->capacity += extraSpace; \
8                size_t newSize = BLOCKNODE_SIZEOF(node); \
9                node = realloc(node, newSize);\
10               } while(0)
```

Note: In C, the function realloc() takes as arguments, an existing pointer to a dynamically allocated chunk of memory, and the new desired size for that memory. The function attempts to resize the memory chunk by simply extending it first, if the downstream addresses are unoccupied. However, the function can return a pointer with a different address than what was passed in, the event it needs to allocate and copy over the memory to a new address. Thus, a macro ideally should be used when dealing with a memory-embedded struct, as this alleviates the need to return expanded pointers to nodes.

We implemented two macros, `BLOCKNODE_SIZEOF` and `BLOCKNODE_EXPAND`. Each of them assists in expanding a BlockNode, allowing more new data to be written to it, if and when needed. However, in order for the node to acquire new data, it must increment it's *len* field and check if it currently has enough capacity. This behavior can be demonstrated with the following:

```
1  #include <string.h>
2
3  BlockNode* BlockNode_write(BlockNode* node, void* data,size_t n)
       {
4          if(n > (node->capacity - node->len)) {
5                  BLOCKNODE_EXPAND(node, (n + 10));
6          }
7          memcpy(node->data, data, n);
8          node->len += n;
9          return node;
10 }
```

Prior to the node acquiring new data, the capacity of the node needs to be checked. The remaining space for a BlockNode, or any real block of memory is given by:

$$space = capacity - length$$

Where *capacity* is the total allocated space of the memory segment, and *length* is the currently occupied boundary of the memory segment. The amount of space available is used to determine the procedure to write incoming data to the node. The cases below describe the course of action, given that n is a positive integer representing the size of the incoming data:

$$\begin{cases} write & n \leq space \\ expand \rightarrow write & n > space \end{cases}$$

Figure 1.1: Different actions taken for a BlockNode to acquire new data.

It's important to understand that although this type of node, a *BlockNode*, has to grow and expand to accumulate more data in it's own memory, nodes do not need to be implemented to grow dynamically. Nodes can use a fixed amount of memory, one that cannot be changed at runtime. These fixed-size nodes benefit from never having to check their space or expand themselves.

The type of nodes the *Wind* language uses are an even further optimization, nodes which only use static memory. In C, declaring an array with the `static` keyword tells the compiler to store that array in the resulting program's data segment. This means that the array can be available throughout the entire runtime, and uses absolute form of addressing. It does not need to wait for neither stack nor heap memory to be allocated for a node to be initialized. This implementation in Wind ill be discussed in a lter chapter.

1.2.2 Node Connections

Up until now, we have mainly described the types of nodes, their pros and cons, as well as their key behavior. Next, the manner in which nodes connect to each other, and the structure of these connections will be discussed. Despite the different ways one can connect nodes to form a *flow*, the main guideline is to not treat or handle nodes as objects. This implies that the nodes themselves and the states they might have should drive the movement of data. The transportation of data should not rely on using external data structures.

The first, most naive form of a node connection is a single direction link. This allows a one way from of travel from one node to another.

```
struct Node {
        size_t cap;
        size_t len;
        struct Node* next;
        unsigned char data[];
};
```

The *next* property, in this arrangement, is always `NULL` or a pointer to another node. This node connection functions very similar to a linked list. In this implementation though, more arbitrary data can be stored in the node, while a linked list is most often only storing individual values in the head of the nodes. Connections like these can similar be changed to produce doubly-connected nodes, or tree like node arrangements. However, nodes do not absolutely need to be linked by their addresses. Given that a population of nodes is constant or near constant over time, certain nodes can be programmed to pass data to other specific nodes.

Nodes can specialize themselves in a flow-based program by assuming specific roles. The roles allow nodes to form a more consistent architecture. This also makes a node arrangement adhere more closely to the lifetime rule of nodes. The longer and more constant a path of nodes exists, the less chance of errors or data leaks.

1.3 Flows

The final topic to introduce in flow-based programming are flows themselves. A flow is the path and direction data follows as it traverses from node to node in a flow-based program. In a previous section, node *connections* were discussed. The difference between a flow and a connection is that, a connection between nodes describes a localized, singular transition, while a flow describes the pathway of data through a program from input to output. To make this distinction, we can describe the simplest possible flow, a flow with no nodes at all. Let's take a look at it visually:

$$I \longrightarrow O$$

Figure 1.2: A flow with zero nodes.

where I is the *input* of the program, and O is the *output* of the program. In this flow, there are no nodes present to manipulate or act whatsoever on the incoming data. Thus, data is simply moved from input to output. There is no acquisitions or transfer of data, it's simply written right back out. From a code perspective, this would occur when command line arguments to a program are just written directly back to `stdout` or `stderr`. This is illustrated in the following program:

```
1 #include <stdio.h>
2
3 int main(int argc, const char** argv) {
4         for(int i=1 ; i<argc ; i++) {
5                 puts(argv[i]);
6          }
7   return 0;
8 }
```

This program loops over the arguments the program is executed with, and prints them as strings to `stdout`. The arguments are never even written to any dynamic or static memory within the program. Flow-based programs with no nodes have no real effect on any arguments or data they consume.

1.3.1 Single Node Flows

Flows with a single node have more control over their data then zero-node flows, yet are restricted to essentially one operation. A flow with a single node only has one "stop", before it reaches it's destination, the output of the program. Here is a diagram of a single node flow:

$$I \longrightarrow N_1 \longrightarrow O$$

Figure 1.3: A flow with one node.

Here, N_1 is the node in the flow. It is enumerated with 1 because in addition to being the only node it's also the first in the flow. In this arrangement, the

flow has a single point in which manipulation of data occurs. However, there is no rule in a flow that the total amount of data coming into the program must be the same as the amount coming out. One possible role of a single node flow is to filter unwanted data, and only allow data that meets some criteria to reach the output of the program. This can be shown as:

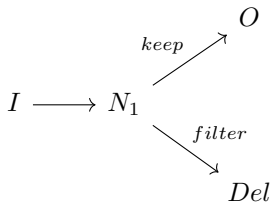

Figure 1.4: A flow with one node that filters.

Del is not a node, but rather a terminal destination that signifies the data is deleted, and will not reach the output of the program. This diagram accurately conveys the path in which data could take in a filtering situation. Yet, there are several operations in which the amount of data from input to output stays the same. For example, a single node flow can map each data value it's node acquires. It could also reduce all the data that passes through it's node to a single value. Overall, a singular node flow supports only one operation that can be performed on data.

1.3.2 Two Node Flows

Flows with two nodes in them offer a tremendous increase in utility and functionality over flows with a single node. First, having two nodes at the basic level allows for more than one operation to be performed on any incoming data. Such as the operations discussed previously, data can me mapped in one node and filtered in another, or mapped in both. The first and most direct advantage of a dual node flow is an additional operation.

With two nodes, the flow can also be arranged in a fashion where the first node is a filter, and data that does not pass the filter is passed to the second node, where it is mapped to the output. Data that does pass the filter test is simply carried to the output directly. This is called a *fork* node, and is illustrated below:

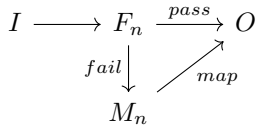

Figure 1.5: A dual flow node with a conditional mapping arrangement.

In the above diagram, F_n is a filtering node, and M_n is a mapping node.

When data arrives at F_n from the input, it is tested on the node's filter. If it passes, it goes straight to the output, O. If it does not pass, it is transferred to M_n, the mapping node. From there, data is mapped directly to the output. The data that travels through this flow forms a *one way path*. There is, one, and only one direction the data can travel. The data cannot backtrack, it has no way of moving back toward the input.

A dual node flow can also be arranged to act on data recursively. This setup is very similar to the filter-map approach. Except, data is continuously checked against the filter, then mapped if it does not fit the filter's criteria. Data in such a flow can only progress to the output if it initially or eventually passes the filter.

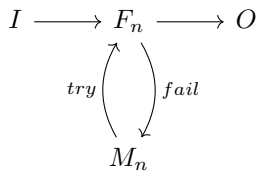

Figure 1.6: A dual flow node with a recursive loop.

In this flow, there is a recursive *loop*, where once data fails the filter, it is mapped, and then tested in the filter node again. This loop continues until the data passes the filter. The main concept here is that, the value of the data is different every time it passes through the filtering node. This procedure mimics the computational concept of recursion. The data is checked continuously against a base case, while if failing it, is incremented in some way.

Overall, using two or more nodes in a flow greatly increases the flow-based program's utility. The exact flow implementation that the Wind language uses will be discussed in a later chapter.

Chapter 2

Setup, Syntax, and Types

This is the wind chapter on syntax and types.