CFluxion Interpreter Design

Ege Özkan

October 4, 2020

Abstract

This paper outlines the design of an implementation of the Fluxion 0.1 Language in the `C` language. The paper outlines the representations used for the Fluxion Datatypes, the process of Expressions and the evaluation using the rules of reduction in the Fluxion Specification, the process of differentiation, the algorithms behind symbolic integration, and some optimizations that will be performed on the interpreter.

1 Introduction

The C programming language may look like bad pick to implement Fluxion, wouldn't a language like LISP would be better for symbolic computation? Or perhaps Python due to its ease of use and readability? If the speed of Python is an issue, wouldn't Julia work? And finally, surely Java would be preferable due to ease of its multiplatform usability?

Although these are valid opinions, the choice of picking C is threefold: The first is its unmatched speed in terms of execution; the second, is that it can ran on almost anything (and be interfaced with anything, including most of the previously mentioned languages, ironically.); and third is its simplicity.

Most Computer Algebra Systems seem to use prewritten libraries, this implementation will aim to be as minimalistic as possible, as well as being powerful, C is a good pick for that.

2 Fluxion Data Types

Although C language does not have "proper" object oriantation, CFluxion uses a form of inheritence called "type puning", or more collaquially, struct inheritence. With this being said, all datatypes will "inherit" from the `FluxionType` struct.

FluxionType
marked: bool dataType: FluxionTypeName

Figure 1: `FluxionType` struct, base for all types.

The `dataType` argument is of type `FluxionTypeName`, this is an enum typedef, and is one of: `Integer`, `Fraction`, `Matrix`, `Vector`, `Set`, `Sequence`, `Enumerable`

2.1 Numeric Types

When dealing with numbers, CFluxion has three internal representations, all of which inherit the `FluxionNumeric` typedef struct.

FluxionNumeric
type: FluxionType* numberType: FluxionNumberType

Figure 2: `FluxionNumeric` struct, base for all numbers.

The `FluxionNumberType` is a typedef enum, with the possible values `Integer`, `Number` and `Irrational`.

`FluxionInteger` struct typedef holds numbers smaller than $2^{32} - 1$, they are small enough that the benefit of introducing a more complex representation is outweighed by the cost this datatype comes with when it comes to simple arithmatic.

FluxionInteger
numeric: FluxionNumeric* value: int32_t

Figure 3: `FluxionInteger` struct, for numbers that can fit `int32_t`.

This more complex data structure is called `FluxionNumber`. Instead of holding the exact value like `FluxionInteger` does, `FluxionNumber` has four dynamically resizing arrays (pointers) each holding integers of type `uint64_t`. These arrays are called `FluxionLongArray`, and this is an utility class.

These arrays, let us call them S_1, S_2, S_3 and S_4 are used to construct the actual numbers i thusly:

$$i = \frac{x}{y} + \frac{z}{q} i = \frac{\sum S_1}{\sum S_2} + \frac{\sum S_3}{\sum S_4} i \tag{1}$$

As can be seen, each list holds the prime factors of x, y, z and q.

We are saving one bit here by using `uint64_t`, however, we now need a way to express negativity, since negative numbers are allowed in Fluxion, this is achieved by putting a zero in S_2 or S_4. since zero cannot occur in these arrays, as that would create a divison by zero problem.

FluxionNumber
numeric: FluxionNumeric*
rNums: FluxionLongArray*
rDens: FluxionLongArray*
iNums: FluxionLongArray*
iDens: FluxionLongArray*

Figure 4: `FluxionNumber` struct, used for numbers.

Using a structure like this solves a couple of problems, first, it will avoid precision related issues ($0.1 + 0.2$ not equaling 0.3, for instance). Second, it will make it possible for large numbers to be expressed easilly. And finally, this will make reduction significantly easier down the line.

2.1.1 Irrational Numerals

Since Irrational numerals are also reserved for use of the Fluxion language, there also exists a third value for numerical objects in Fluxion outside those numbers $x \notin \mathbb{Q} \land x \in \mathbb{R}$. Most notably, three irrational numbers are reserved by the Fluxion standard, e, π and τ. These are represented by the `FluxionIrrational` class, there is also a helper enum typedef, `FluxionIrrationalType`, which is either `EULER`, `PI` or `TAU`.

FluxionIrrational
numeric: FluxionNumeric*
value: FluxionIrrationalType

Figure 5: `FluxionIrrational` struct, for e, π and τ.

2.2 Matrices

Matrices form an important part of mathemtics, and Fluxion, Matrices are represented using the base class `FluxionMatrix`, which has two subtypes, `Fluxion2Matrix` and `FluxionVector`. These subtypes are used for optimizing the memory usage of the classes.

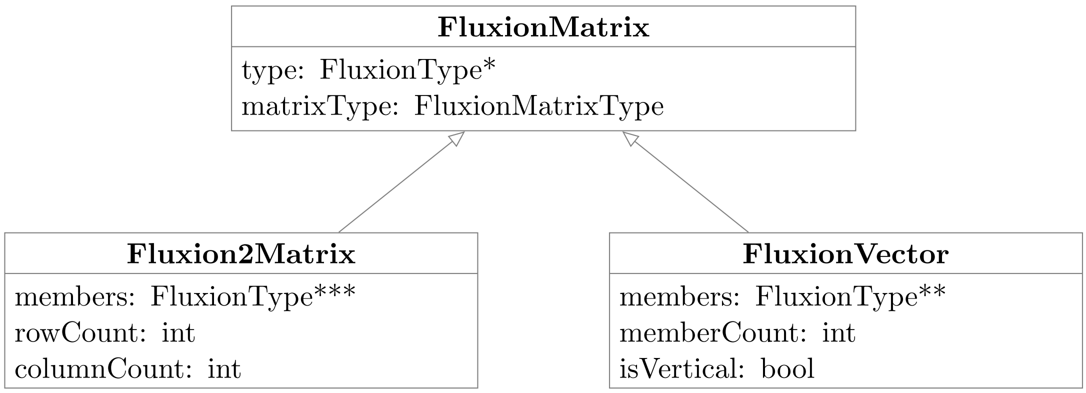

Figure 6: `FluxionMatrix` and its subtypes.

Observe that we can just use pointers here rather than any sort of dynamically allocated array since the size of vectors, and matrices, are known at compile time.

2.3 Sets & Sequences

Despite being the same type, infinite sets differ significantly from finite ones. Actually, infinite enumarables, differ from infinite sets too, despite technically being a subtype of them, so much so same type can be used to represent both them and Sequences, which are **not** sets. But sometimes, sets and sequences can be auto-cast to each other.

This interchangableness of the type causes a unique situation, despite Enumerables are subtypes of Sets and Sequences are another type entirely in the Fluxion standard, CFluxion uses an enum typedef `FluxionColType` for collections, that takes one of three values `InfiniteSet`, `FiniteSet` and `Sequance`. It also uses a base typedef for all of them `FluxionCollection`

Observe in Figure 7 that the generation rules for classes are normal Fluxion functions. `isSequenceOnly` is used for determining which operations are allowed on sequences. Also observe that the `FluxionFiniteCol` has a set number of members, as we also know the number of members of a sequence in compile time, *and* Fluxion's collections are immutable.

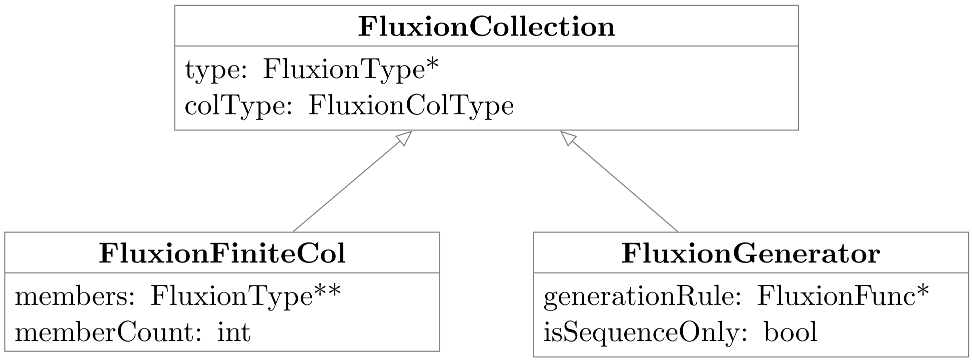

Figure 7: `FluxionCollection` and its subtypes.

2.4 Operations

Operations are, surprisingly, also types in Fluxion. The `FluxionOperation` struct typedef also inherits from `FluxionType`. And uses a helper enum typedef, `FluxionOpType`. Each enum represents a specific symbol. All operations are converted to a binary equivalent, for instance `- (x + 1)` is converted to `(-1) * (x + 1)`.

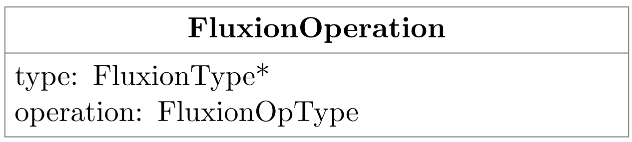

Figure 8: `FluxionOperation` struct, representing an operation.

2.5 Names & Function Calls

A Name is a variable name appearing inside an expression prior to binding. Function calls are also special names.

Although the argument count can be got from the actual function once the name is bound, the posibility of an erronous call means that the argument count must still be kept inside the body as well.

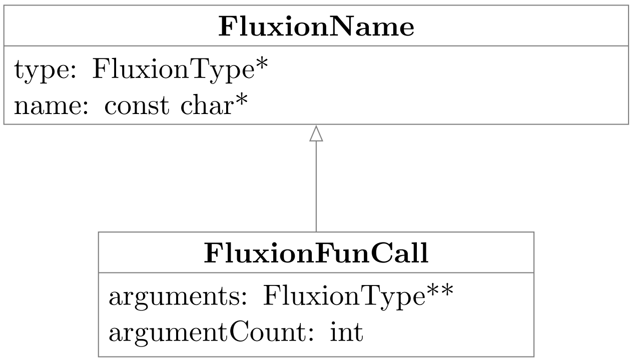

Figure 9: `FluxionName` and `FluxionCall`, representing variables.

2.6 Expression

Expressions are also `FluxionType`s, since they can appear inside function calls and alike. To be fair, this is actually because of the reverse, the Fluxion standard says the literals or variables of the other types by themselves are Expressions, but this is easier to implement.

The Expressions are, trees, more than that, they are binary trees. Since all of our operations are binary, and those types that are not operations can only apear in leaves of our tree.

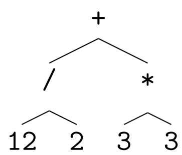

Figure 10: Expression tree for `12/2 + 3 * 3`

As can be seen in Figure 10, expression trees also adhere to the precedence of the operations. Each expression is evaluated by evaluating the operation on its left storing the result in a variable l, then evaluating the expression on its right storing the result in a variable r, and then performing the apporiprate operation on them (depending on the operation) and returning the result $f(l, r)$.

Expressions are stored using a tree structure with an associative array, where, the root node is at index 1, and for each node in index n, its right node is in $2n$, and its left node is in $2n - 1$.

Again, since each token will compile to a set number of oprations, we know exactly how many terms there are. There are a few things to make

FluxionExpression
type: FluxionType* terms: FluxionType** termCount: int

Figure 11: `FluxionName` and `FluxionCall`, representing variables.

sure while compiling an expression from its tokens. The first is to make sure the tree itself is balanced and second, is to make sure the order of precedence is preserved, the tree must be formed such that no operation of higher precedence will occur on a higher level of three than an operation with a lower precedence. To achieve these goals, we wil make two passes on the expression.

The first pass, we will extend any unary operation to its binary equivalent.