Fluidity Proposal Annexe

Part #1: The Basic Language Design

```
(a b).
((:) ~ ~) < (: ~).
```

Above is a small example of Fluidity code. Fluidity is a lot like Prolog, but instead of defining predicates directly, you define a program that enumerates all the correct values for a predicate, i.e., values that satisfy the predicates conditions. The code above generates the following valid expressions:

```
(a b)
((a) b b)
(((a)) b b b b)
! etcetera
```

The first line generates the valid expression `(a b)` as the base expression. The second line takes in two values with `(: ~)` and passes them into `((:) ~ ~)`, which then repeats over and over to generate an infinite list of valid expressions. Note that `!` denotes a comment until the next newline character. Also note that expression variables begin with a tilde, string variables begin with a colon, and atoms begin with nothing.

Let's now consider a more complicated example:

```
(A :x u ~) <
    (B x ()),
    (C u ~).
```

In this example there are two conditions which are conditions via conjunction denoted by a comma separating the two conditions. This produces the following valid expressions:

```
(A x u v) <
    (B (() x)),
    (C u v).
! etcetera
```

Note that `()` is considered an empty string variable. Perhaps we should define something useful.

```
(num 0).
(num (s ~n)) < (num ~n).
```

In this example we define the integers 0 and above. The following expressions are valid:

```
(num 0) ! equivalent to 0
(num (s 0)) ! equivalent to 1
(num (s (s 0))) ! equivalent to 2
! etcetera
```

Where `s` stands for successor. While this isn't practical for the programmer, and there are better solutions for defining natural numbers, it demonstrates that the program is responsible for extending itself to best fit the application domain. No requirements are assumed so that the qualities of an environment can be fit best by an implementation. If we then wanted to define the sum operator, we could do so like this:

```
(plus 0 ~n ~n) < (num ~n).
(plus (s ~1) ~2 (s ~3)) :-
    (plus ~1 ~2 ~3).
```

I'm sure you'll recognize how the first clause works, but the second may be more confusing.

What happens is that it will keep being called until the first value is zero, and then it will be backtracked to construct the third value.

Part #2: Complementing Valid Expressions

Complementing valid expressions is an extension to the axiomatic language designed by Walter W. Wilson described here (Note that different syntax is used). To demonstrate the complement of an expression, you use square brackets instead of parentheses to hold the expression. Let's look at an example:

```
(member ~ (:1 ~ :2)).
(sequence (:)).
(not_member ~x ~seq) <
    (sequence ~seq),
    [member ~x ~seq].
```

In this example we define two normal axioms, one that checks if a sequence contains a member and another that defines the set of all valid sequences.

Part #3: Exclusive Guarded Horn Clauses

Fluidity takes the concept of EGHCs (Exclusive Guarded Horn Clauses) from the P-Prolog system. EGHCs make Fluidity easier to run in parallel or a distributed environment. EGHCs are a rule when writing axioms that state each clause in an axiom (clauses that are not mutually exclusive in being committable) are run in parallel using or-parallelism. Clauses that are not mutually exclusive in being committable are run in parallel using or-parallelism. This does very little to modify the actual program structure but makes it clear precisely how the programs will be made automatically parallel.

Part #4: Addressing Blockchain Security

The Ethereum blockchain runs processes synchronously across a global network to ensure that all programs are audited by all Ethereum nodes and to ensure Ethereum can stay a trustless network. Under a standard parallel distributed execution environment, bad actors can insert false information where they are depended upon for partial computation. The solution is an ugly one, but one that maintains some of the scalability that comes with parallel Fluidity execution. For blockchains that are willing to sacrifice some security, the problem is void, but for blockchains that aren't, it is pointed out that with Fluidity, it is easier to verify computation than it is to redo the computation, which means not all nodes have to do the full computation, and instead can verify the computation of other untrusted nodes. Take, for example, the member axiom defined above. A node tasked with verifying the computation would require only the information to efficiently find the element if there were such an element. Similarly, with the plus axiom, we would only need to do the backtracking to verify the computation.

Part #5: Efficient Term Representation in Memory

It is no secret that storing large numbers as a nested structure where each level represents an increment of one is not a practical solution. In this section, we investigate how we can not only drastically reduce memory usage but convert terms into highly efficient data structures automatically.

```
(num 0).
(num (s ~n)) < (num ~n).
```

In the example above, we can demonstrate that for each value but 0, each clause is exclusively committable. Since the second clause will never return (num 0), we can assign that value to the literal value 0 in memory. For all other inputs, only one value will be unified against, so we can assign each of those values as an increment of one in relation to each other. Note that this does not necessarily mean that (num 0) would be assigned the value 0 but that it would be assigned a single integer value, and all the other values would be assigned distinct integer values following a pattern of repeatable translation. It should again be noted that this approach for representing natural numbers is not ideal and indeed, you can represent them more effectively with an axiom that maps each sequence of decimal numbers to a sequence of binary digits.

```
(num (0 1 1 0 1 0 1 0))
```

Depending on the implementation, this could have a very similar memory representation to the previous example. For other, more complicated structures, it remains possible to map their values to integers and data structures of integers or other structures, e.g., sets can sometimes be made into ribbon filters, many structures can utilize parallel hashmaps, etc. There is one more detail that must be considered; for efficient computation, we must be able to compute these axioms on their in-memory representations instead of their programmatic representations because often converting would be a highly expensive process. To solve this issue, for each axiom defined, a change structure, a concept from ILC (Incremental Lambda Calculus), will be automatically generated to map changes in the programmatic representation to changes in the in-memory representation and vice-versa. This also makes program verification significantly more efficient.

Part #6: Memory Management

Almost all languages that exist use some form of manual memory management or garbage collection. There are some exceptions; for example, Gnuprolog does not implement garbage collection or require manual memory management. When programming in a language like Java, one should expect that local variables inside a function be freed when the function stops executing and returns a value. The behaviour of a Fluidity program is very simple: when a value is unified, and no more unification is necessary, all existing memory for that predicate call is freed. When executing Fluidity programs, it is often necessary to employ a search mechanism for unification; as such, when that predicate starts to deconstruct that search data, it can be freed automatically; i.e., global stack space can be easily and efficiently reclaimed on backtracking. In more concrete terms, memory can be efficiently reclaimed even without a garbage collector.

Part #7: Conclusion

It has been demonstrated that an efficient implementation of Fluidity is theoretically possible and that in systems where synchronous execution was believed to be necessary for blockchain security, Fluidity can improve scalability through parallel processing without sacrificing blockchain security. It has also been demonstrated that an intentional interpreter is likely not necessary for efficient representation of Fluidity terms and that while Fluidity will likely never execute as efficiently as the EVM in a synchronous environment, it can theoretically outperform it exponentially in an asynchronous environment.