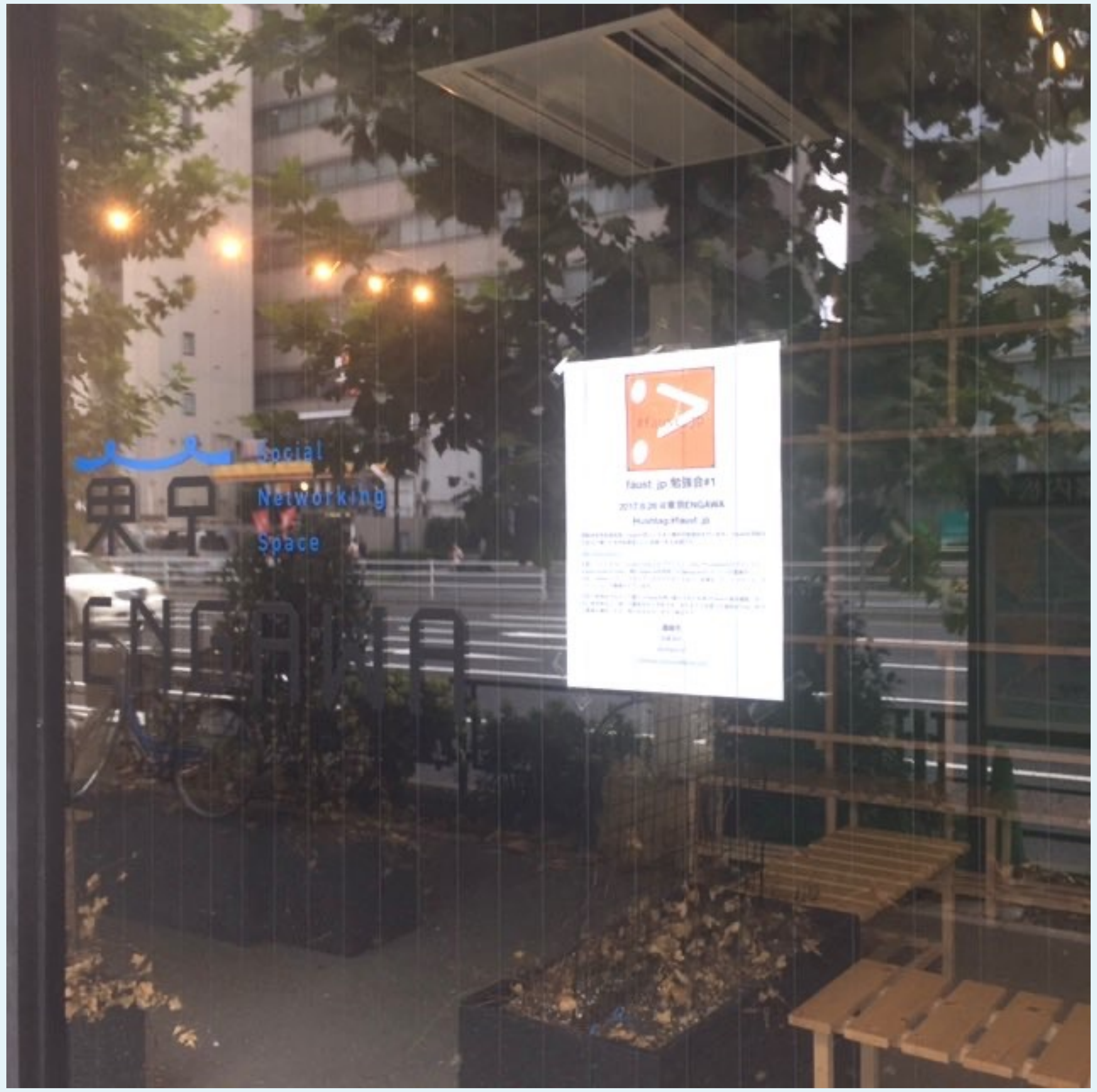
λ_{mmm}

-the Intermediate Representation for Synchronous Signal Processing Language Based on Lambda Calculus

2024-11-21 International Faust Conference 2024

MATSUURA Tomoya / Tokyo University of the Arts, Art Media Center(me@matsuuratomoya.com)



back in 2017. A first (and perhaps the last since today) Faust learning meeting in Japan

https://doi.org/10.5281/zenodo.13855342

Sorry, there were some errors in the typing rules and example codes on the paper!
Corrected version is currently uploaded on Zenodo.

Agenda

1. Background
2. Syntax of mimium and Lambda-mmm
3. Naive Operational Semantics of Lambda-mmm
4. VM and bytecode format for Lambda-mmm
5. Discussion

1. Background

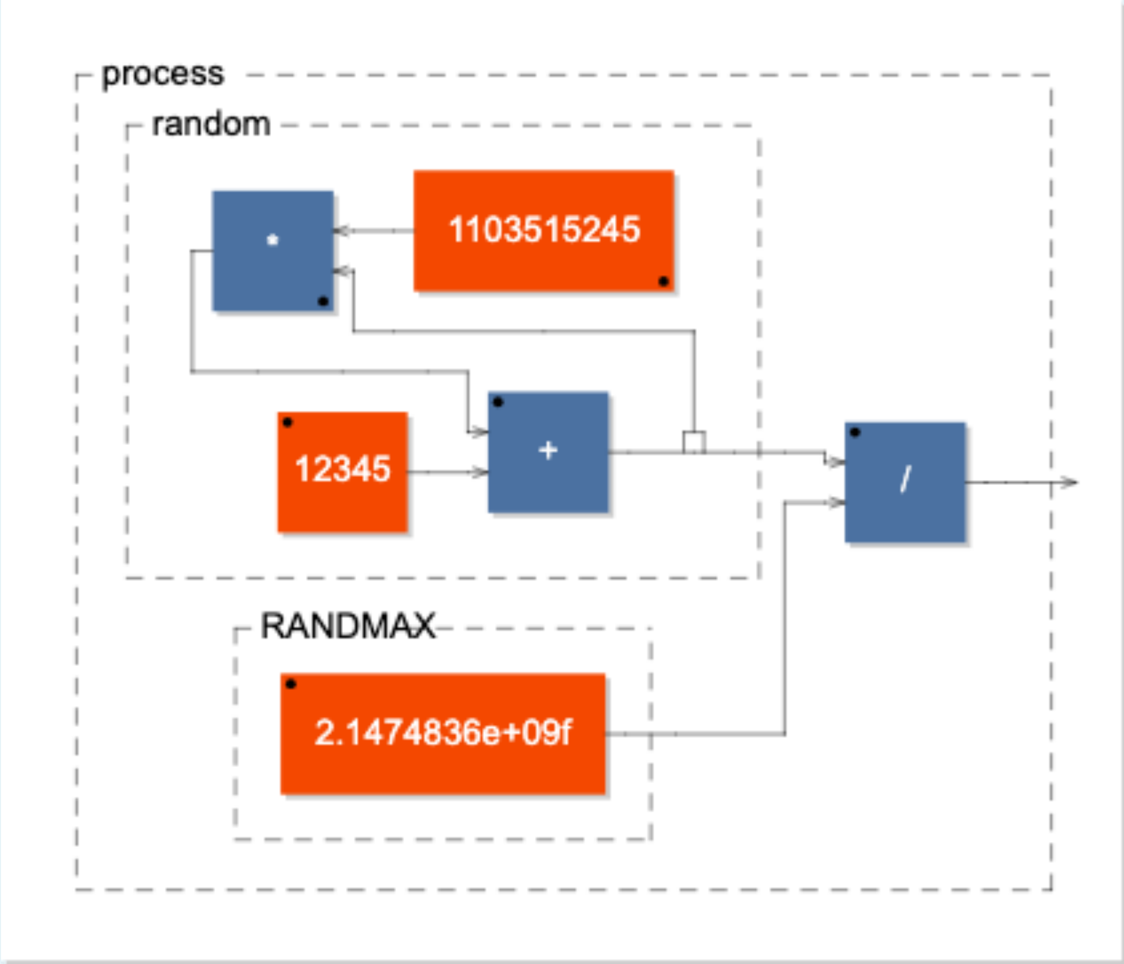
- Formalization of synchronous signal processing languages
- Need of formalization for mimium, lambda-calculus based DSP language

Background 1: Languages for Signal Processing

Faust

- Block Diagram Algebra: combining block with in/outs by 5 composition operators
    - parallel(,) sequential(:) split(<:) merge(:>) recursion(~)
- Primitive blocks: constant / arithmetics / delay / conditional*

FAUST
Functional
Audio
Stream



*Faust's conditional evaluate both branch and take either of the results

Pros and Cons in Faust

- + One algorithm can be translated into multiple platforms: C++/Rust/LLVM IR...
- **Lacks theoretical compatibility between other general systems like lambda-calculus**
    - - External function call from Faust must be pure
        - +- Easy to embed Faust to the host, Uneasy to call host's functions
- Term-Rewriting Macro is an independent system from BDA
    - +Can represent complex signal graph with pattern-matching
    - - Bad macro may causes an error because of in/out mismatch in BDA, but hard to understand the reason for the programmer
    - - Implicit distinction between signal(number) and compile-time constant integer

Idea: lambda calculus +
minimum primitives for the time operation

Idea: lambda calculus + minimum primitives for the time operation

Delay and Feedback

Background 2: mimium(2020~)

minimal **mu**sical medi**um** / ***mimi*(耳👂)+*medium***

https://github.com/tomoyanonymous/mimium-rs

mimium's syntax for feedback

mimium

```
fn onepole(x,g){
   x*(1.0-g) + self*g
}
```

can refer to the return value of 1 sample before

Faust

```
onepole(x,g) =  (1.0 - g) * x + g * _ ~ _;
```

or

```
onepole(x,g) =
self ~ _ with { self(y) = (1.0 - g) * x + g * y; };
```

(Simplified `si.smooth`)

Problems in the previous version of mimium

- No formal semantics
- Could not compile codes when the higher-order function is used with the stateful function: refers to `self` or `delay` somewhere in the call tree
  - = the allocation size of internal state for the feedback & delay cannot be determined at the compile time
- = Impossible to generate a signal graph parametrically
- →Re-design & implement the compiler from zero again

(Also, I was exhausted to write compiler in C++ and wanted to switch to Rust)

Prior works on lambda-based DSP language

- Kronos[Norilo 2015]
  - Based on System-Fω, Type-level computation corresponds to the signal graph generation
  - No formal semantics(compiler code is the reference)
- W-Calculus[Arias et al. 2021], strongly formalized with Coq
  - **No higher-order function / only for linear-time invariant systems**

W-calculus with loosening these restriction => λ_{mmm}

Prior works on lambda-based DSP language

- Kronos[Norilo 2015]
- Based on System-Fω (Type-level lambda abstraction can be used)
  - Type-level computation corresponds to the signal graph generation
  - Feedback is represented as a type-level recursive function application
- No formal semantics(compiler code is the reference)

Prior works on lambda-based DSP language

- W-Calculus[Arias et al. 2021], strongly formalized with Coq
- Introduces "feed" to the lambda calculus that represents feedback with 1 sample delay
- "onepole" example can be expressed like $\lambda x.\lambda g.\ feed\, y.\ x * (1.0 - g) + y * g$
- <u>**No higher-order function**</u>
  - Lambda abstraction can map from tuple of number, to tuple of number in the type system.

$$\frac{\Gamma, x : \mathrm{R}_a \vdash e : \mathrm{R}_b}{\Gamma \vdash \lambda x.e : \mathrm{R}_a \to \mathrm{R}_b}\ \textsc{Lam}$$

- Only `Expr + Expr` and `Constant * Expr` are allowed primitive operations for expressing linear time-invariant system (like basic filter and reverb)

Prior works on lambda-based DSP language

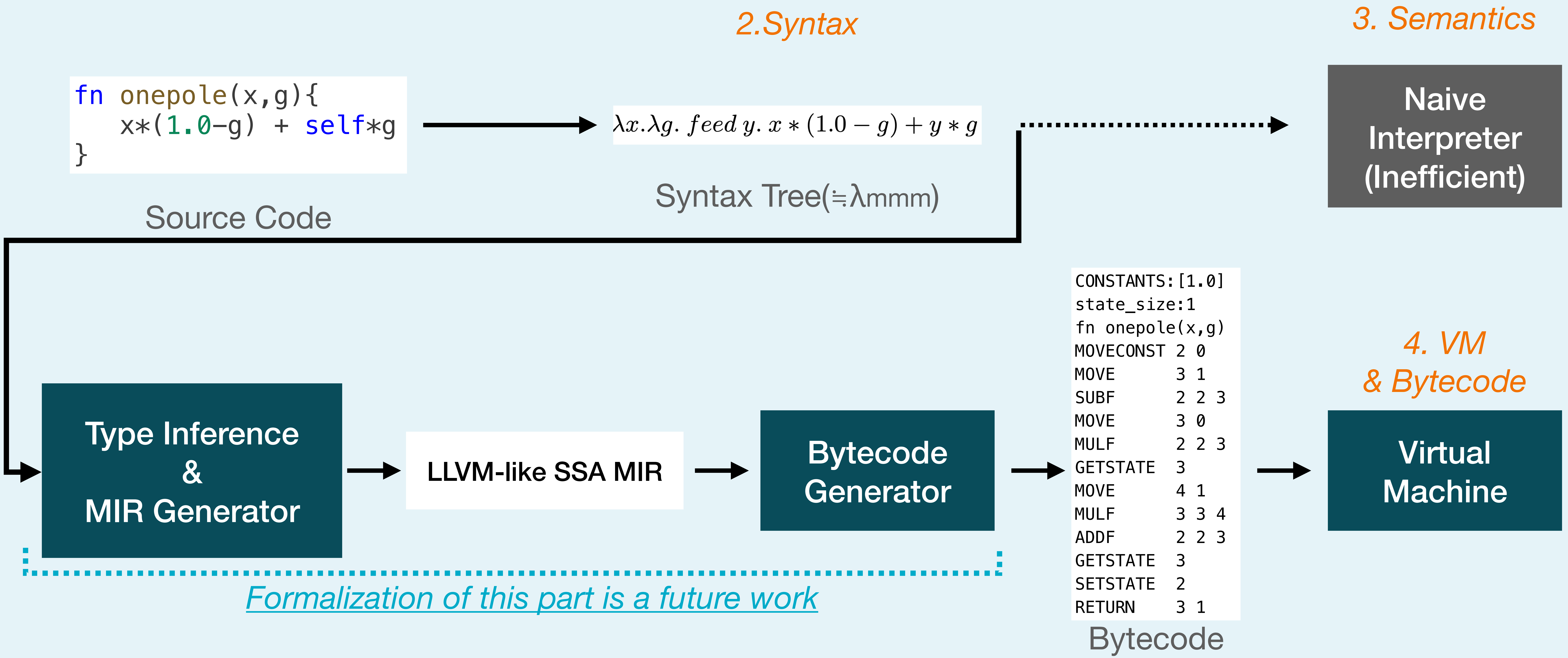
- W-Calculus[Arias et al. 2021], strongly formalized with Coq
- Introduces "feed" to the lambda calculus that represents feedback with 1 sample delay
- "onepole" example can be expressed like $\lambda x.\lambda g.\ feed\, y.\ x * (1.0 - g) + y * g$
- **<u>No higher-order function</u>**
  - Lambda abstraction can map from tuple of number, to tuple of number in the type system.

$$\frac{\Gamma, x : \mathrm{R}_a \vdash e : \mathrm{R}_b}{\Gamma \vdash \lambda x.e : \mathrm{R}_a \to \mathrm{R}_b}\ \textsc{Lam}$$

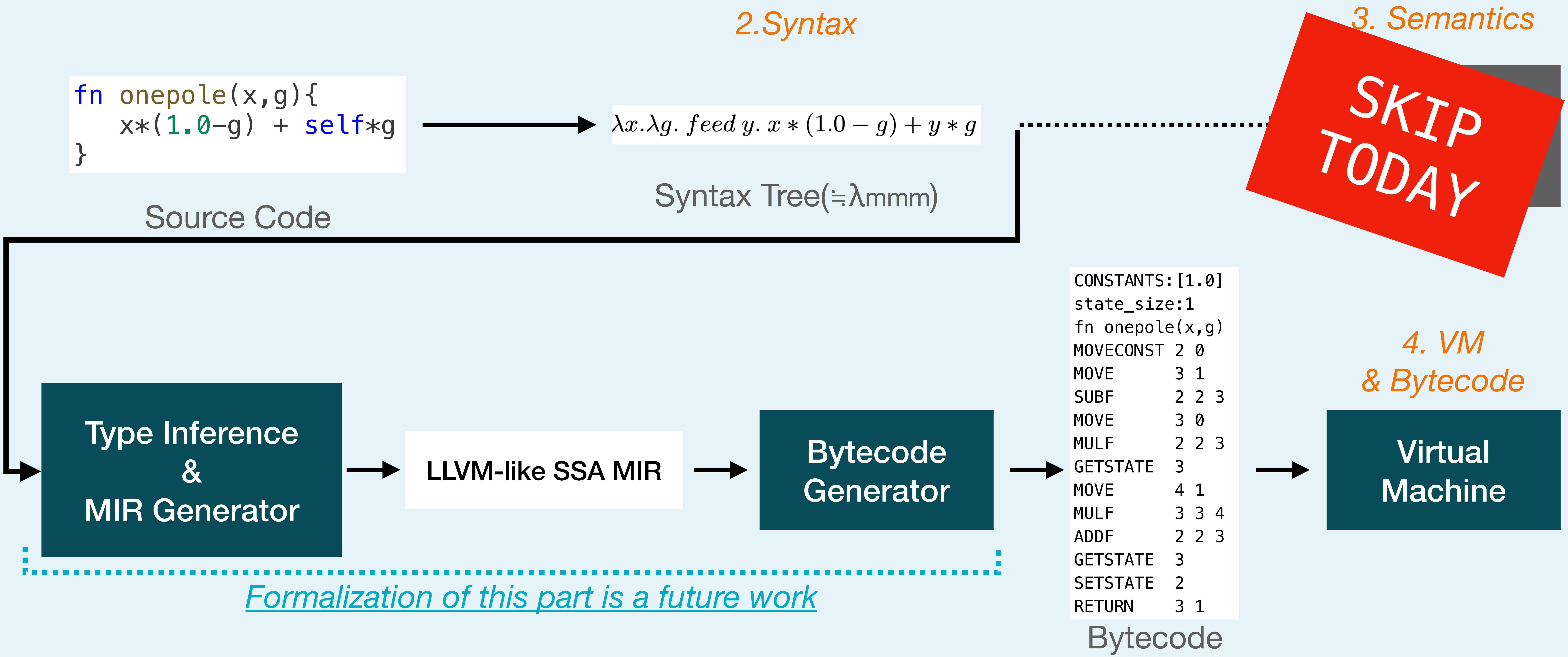
- Only `Expr + Expr` and `Constant * Expr` are allowed primitive operations for expressing linear time-invariant system (like basic filter and reverb)

W-calculus with loosening these restriction => λ_{mmm}

Scope of This Paper & Compiler Pipeline



Scope of This Paper & Compiler Pipeline



2. Syntax of λ_{mmm}

Syntax of λmmm

base on a simply typed, call by value lambda calculus

$$
\begin{aligned}
\tau_p &::= R && [real] \\
&\mid N && [nat] \\
\tau &::= \tau_p \\
&\mid \tau \to \tau && [function]
\end{aligned}
$$

Types

$$
\begin{aligned}
v_p &::= r \quad r \in \mathbb{R} \\
&\mid n \quad n \in \mathbb{N} \\
v &::= v_p \\
&\mid cls(\lambda x.e, E)
\end{aligned}
$$

Values

$$
\begin{aligned}
e ::= &\ x && x \in v_p && [value] \\
\mid &\ \lambda x.e && && [lambda] \\
\mid &\ let\ x = e_1\ in\ e_2 && && [let] \\
\mid &\ fix\ x.e && && [fixpoint] \\
\mid &\ e_1\ e_2 && && [app] \\
\mid &\ if\ (e_c)\ e_t\ else\ e_e && && [if] \\
\mid &\ delay\ n\ e_1\ e_2 && n \in \mathbb{N} && [delay] \\
\mid &\ feed\ x.e && && [feed] \\
\mid &\ \ldots
\end{aligned}
$$

Terms

(Aggregate types like tuple are omitted in this paper.)

Typing Rule(Excerpt)

$$\frac{\Gamma, x : \tau_a \vdash e : \tau_b}{\Gamma \vdash \lambda x.e : \tau_a \to \tau_b}$$

[T-LAM]

"Allows maps from any type to any type"

$$\frac{\Gamma \vdash n : N \quad \Gamma \vdash e1 : \tau \quad \Gamma \vdash e_2 : R}{\Gamma \vdash delay\, n\, e_1\, e_2 : \tau}$$

[T-DELAY]

"Time index must be real number"

$$\frac{\Gamma, x : \tau_p \vdash e : \tau_p}{\Gamma \vdash feedx.e : \tau_p}$$

[T-FEED]

"Feed must not return functional type"

$$\frac{\Gamma \vdash e_c : R \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_e : \tau}{\Gamma \vdash if\,(e_c)\, e_t\, e_e : \tau}$$

[T-IF]

"Use number instead of boolean for condition"

Typing Rule(Excerpt)

$$\frac{\Gamma, x : \tau_a \vdash e : \tau_b}{\Gamma \vdash \lambda x.e : \tau_a \to \tau_b}$$ [T-LAM]

$$\frac{\Gamma \vdash n : N \quad \Gamma \vdash e1 : \tau \quad \Gamma \vdash e_2 : R}{\Gamma \vdash delay\, n\, e_1\, e_2 : \tau}$$ [T-DELAY]

$$\frac{\Gamma, x : \tau_p \vdash e : \tau_p}{\Gamma \vdash feed x.e : \tau_p}$$ [T-FEED]

$$\frac{\Gamma \vdash e_c : R \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_e : \tau}{\Gamma \vdash if\,(e_c)\, e_t\, e_e : \tau}$$ [T-IF]

Only primitive types are allowed for feed to simplify implementation.

However, returning function in feed could be theoretically possible. (The function whose behavior changes sample-by-sample?)

3. Naive Operational Semantics of λ_{mmm}

Operational Semantics of λmmm

(Big-step style, Excerpt)

$$\frac{E^n \vdash e_1 \Downarrow v_1 \; n > v_1 \; E^{n-v_1} \vdash e_2 \Downarrow v_2}{E^n \vdash \; delay\, n\, e_1\, e_2 \Downarrow v_2}$$ [E-DELAY]

$$\frac{}{E^n \vdash \; \lambda x.e \Downarrow cls(\lambda x.e, E^n)}$$ [E-LAM]

$$\frac{E^{n-1} \vdash e \Downarrow v_1 \; E^n, x \mapsto v_1 \vdash e \Downarrow v_2}{E^n, x \mapsto v_2 \; \vdash \; feed\, x\, e \Downarrow v_1}$$ [E-FEED]

$$\frac{E^n \vdash e_c \Downarrow n \; n > 0 \; E^n \vdash e_t \; \Downarrow v}{E^n \vdash \; if(e_c)\, e_t\, else\, e_t \Downarrow v}$$ [E-IFTRUE]

$$\frac{E^n \vdash e_c \Downarrow n \; n \leqq 0 \; E^n \vdash e_e \; \Downarrow v}{E^n \vdash \; if(e_c)\, e_t\, else\, e_t \Downarrow v}$$ [E-IFFALSE]

$$\frac{E^n \vdash e_1 \Downarrow cls(\lambda x_c.e_c, E^n_c) E^n \vdash e_2 \Downarrow v_2 \; E^n_c, \; x_c \mapsto v_2 \vdash e_c \Downarrow v}{E^n \vdash \; e_1 \qquad e_2 \Downarrow v}$$ [E-APP]

This semantics stores evaluation context in each sample as E^n.

If referred to the environment of n<0, it returns 0.

In this semantics, the value from 0 to the present is recalculated every sample, and the variable environments are recreated and discarded each time.

4. VM to execute λ_{mmm}

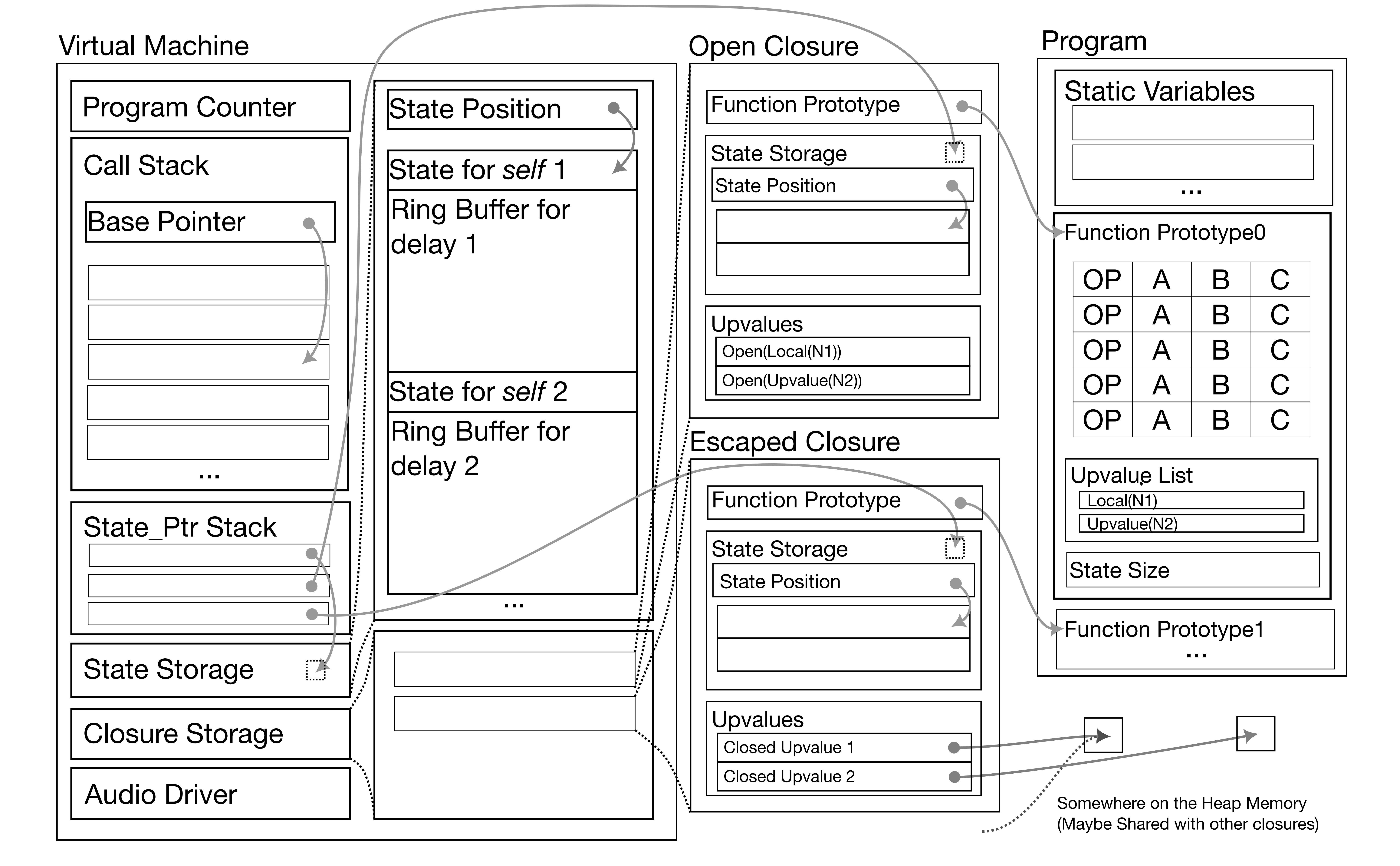
VM and Bytecodes for λ_{mmm}

- Based on Lua VM 5.0 (Register-machine but the register is represented as just the relative position on a call stack from a base pointer)
  - Resolves captured values of the closure by special instruction `getupvalue`
- Tuned for static typed language
  - e.g. Call to the global function and Call to the closure are different operation
  - Only closures are heap-allocated (currently managed by reference-counted GC)
- **Operations for getting/setting internal state variable for `self` and `delay`**

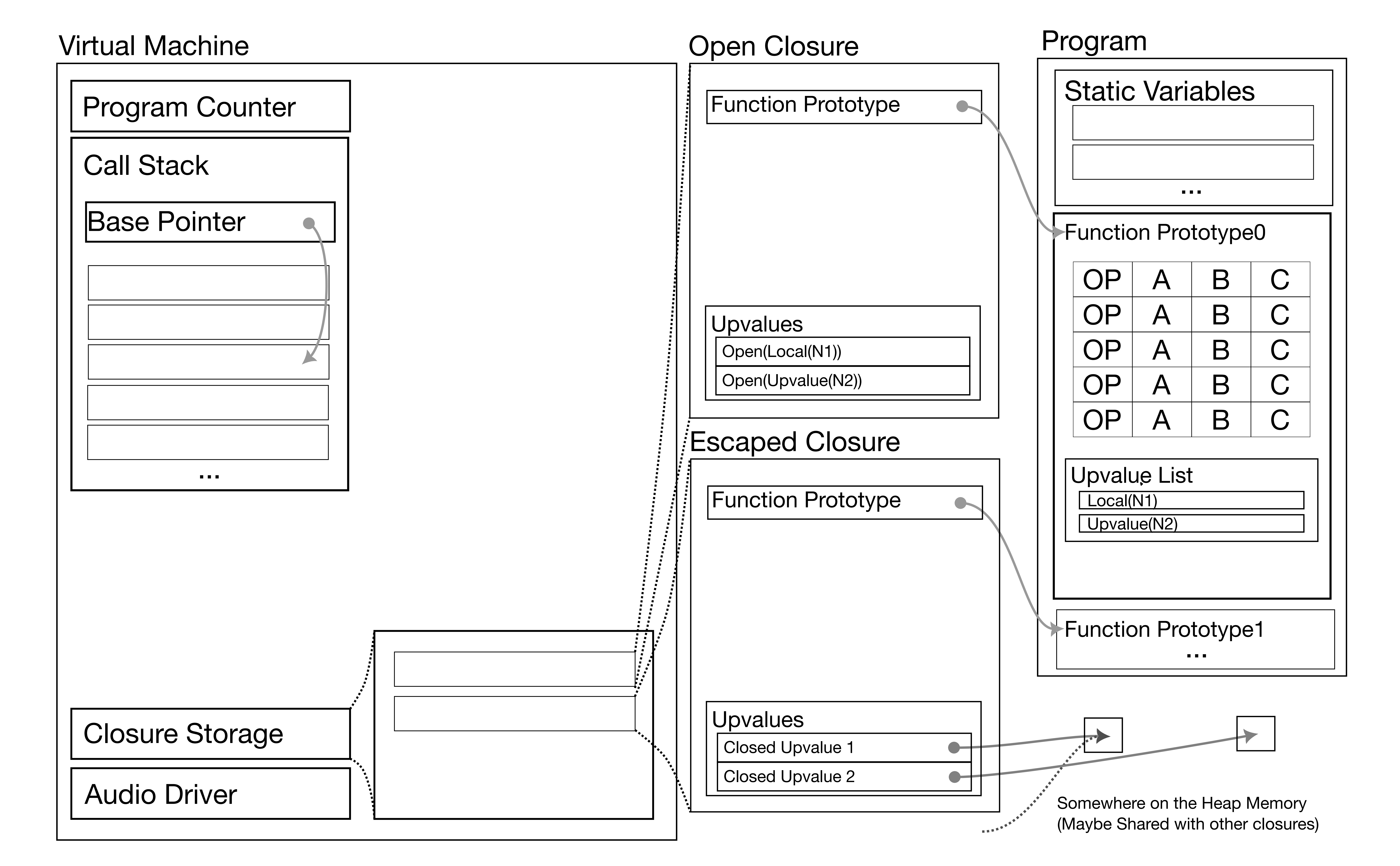
```
MOVE A B R(A) := R(B)
MOVECONST A B R(A) := K(B)
GETUPVALUE A B R(A) := U(B)
(SETUPVALUE does not exist)

GETSTATE* A R(A) := SPtr[SPos]
SETSTATE* A SPtr[SPos] := R(A)
SHIFTSTATE* sAx SPos += sAx
DELAY* A B C R(A) := update_ringbuffer(SPtr[SPos],R(B),R(C))
*(SPos,SPtr)= vm.closures[vm.statepos_stack.top()].state
(if vm.statepos_stack is empty, use global state storage.)
JMP sAx PC +=sAx
JMPIFNEG A sBx if (R(A)<0) then PC += sBx
CALL A B C R(A),...,R(A+C-2) := program.functions[R(A)](R(A+1),...,R(A+B-1))
CALLCLS A B C vm.statepos_stack.push(R(A))
               R(A),...,R(A+C-2) := vm.closures[R(A)].fnproto(R(A+1),...,R(A+B-1))
               vm.statepos_stack.pop()
CLOSURE A Bx  vm.closures.push(closure(program.functions[R(Bx)]))
               R(A) := vm.closures.length - 1
CLOSE A close stack variables up to R(A)

RETURN A B return R(A), R(A+1)...,R(A+B-2)
ADDF A B C R(A) := R(B) as float + R(C) as float
SUBF A B C R(A) := R(B) as float - R(C) as float
MULF A B C R(A) := R(B) as float * R(C) as float
DIVF A B C R(A) := R(B) as float / R(C) as float
ADDI A B C R(A) := R(B) as int + R(C) as int
...Other basic arithmetic continues for each primitive types...
```

(In the actual compiler, most of the operation have an additional operand to indicate word-size of the value to handle aggregate-type value)

Overview of the VM and Program



Simplified version when no stateful functions are used



Case: combining multiple delay with feedback

```
fn fbdelay(x,fb,dtime){
    x + delay(1000,self,dtime)*fb
}
fn twodelay(x,dtime){
    fbdelay(x,dtime,0.7)
      +fbdelay(x,dtime*2,0.8)
}
fn dsp(x){
    twodelay(x,400)+twodelay(x,800)
}
```

"fbdelay" uses delay with 1000 as a maximum samples , and self

"twodelay" uses "fbdelay" twice

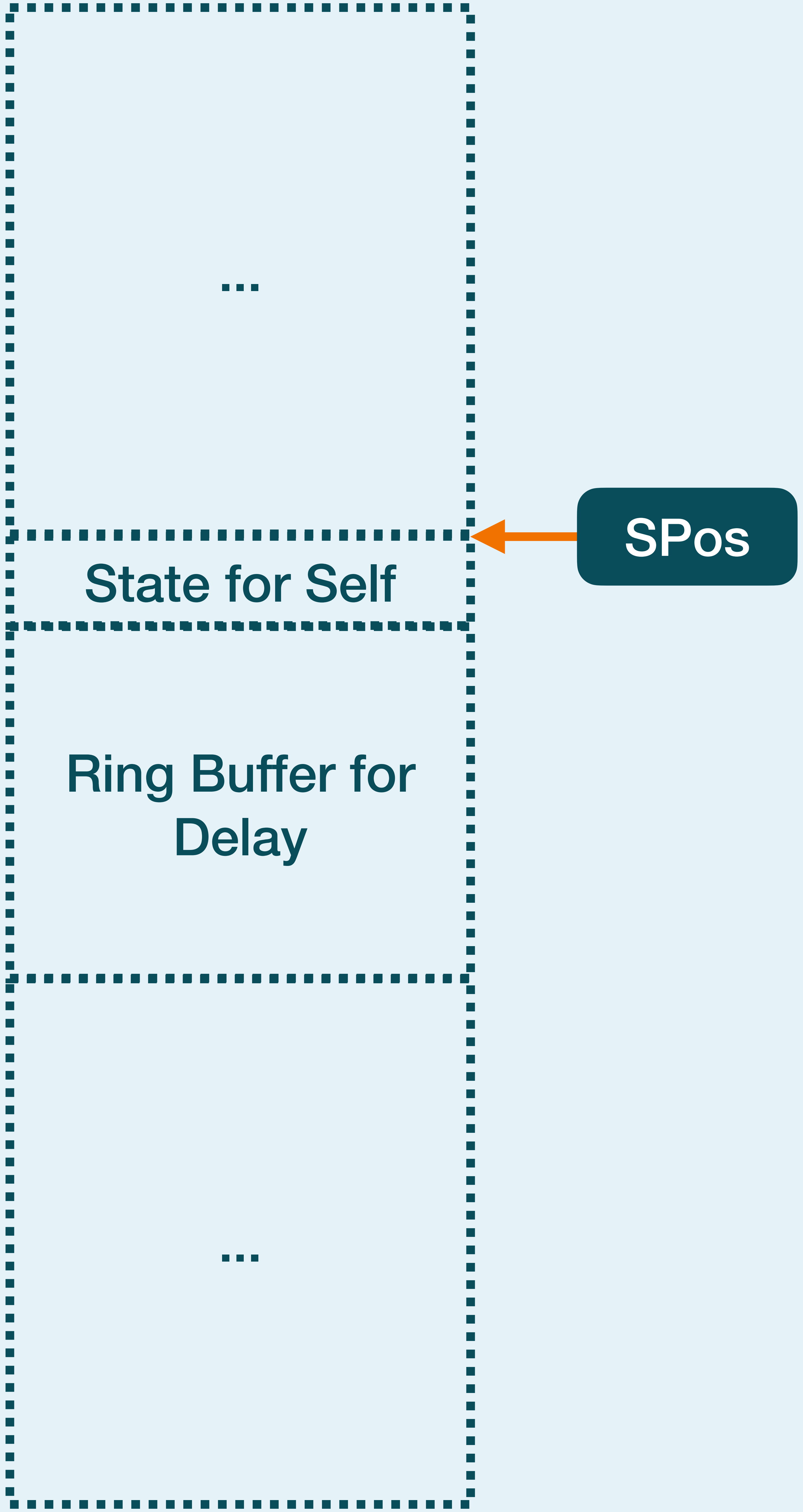
"dsp" uses "twodelay" twice

```
CONSTANTS:[0.7,2,0.8,400,800,0,1]
fn fbdelay(x,fb,dtime)
state_size:1004
    MOVE       3 0 //load x
    GETSTATE   4
    SHIFTSTATE 1
    DELAY      4 4 2
    MOVE       5 1
    MULF       4 4 5
    ADDF       3 3 4
    SHIFTSTATE -1
    GETSTATE   4
    SETSTATE   3
    RETURN     4 1
```

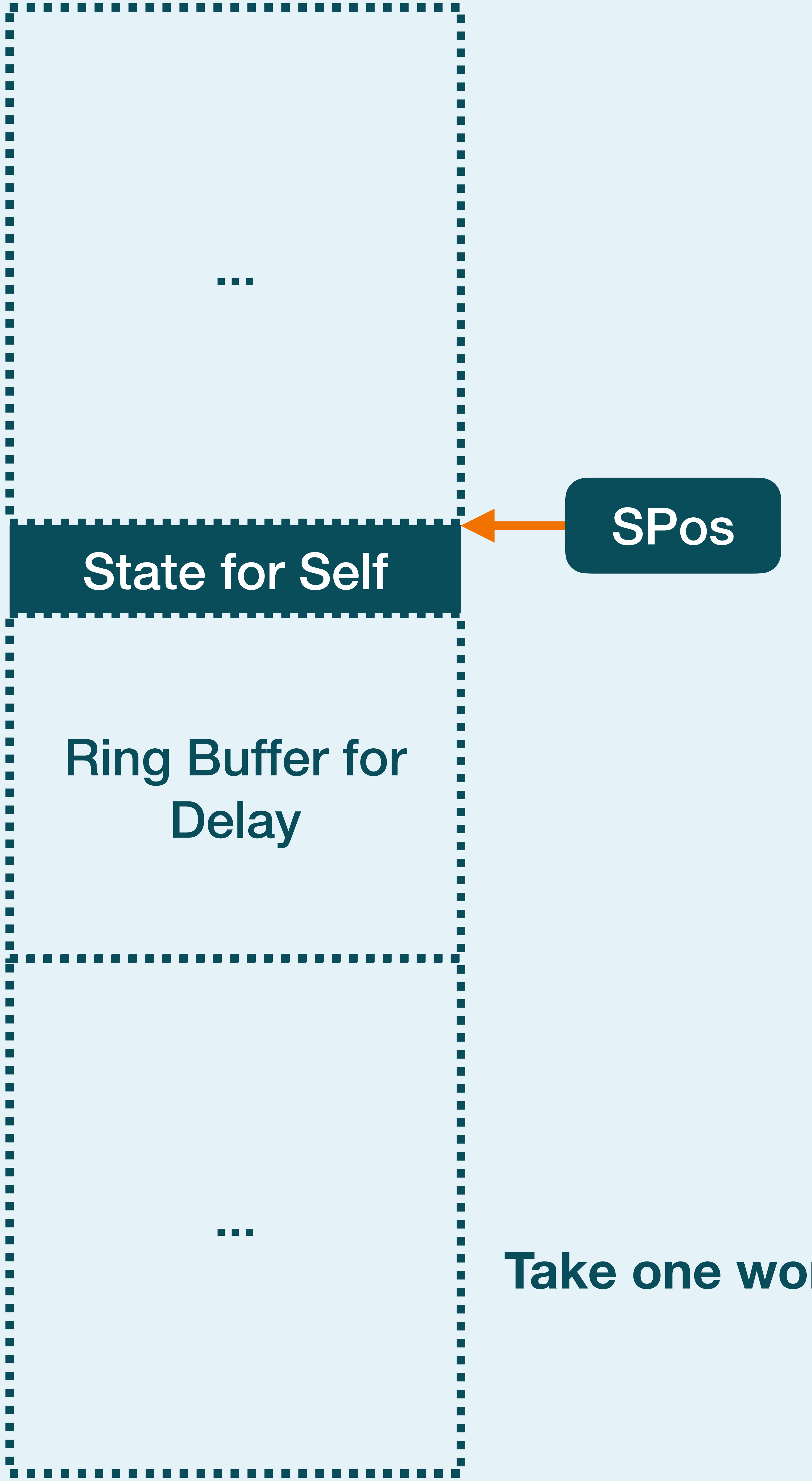
```
fn twodelay(x,dtime)
state_size:2008
    MOVECONST  2 5
    MOVE       3 0
    MOVE       4 1
    MOVECONST  5 0
    CALL       2 3 1
    SHIFTSTATE 1004
    MOVECONST  3 5
    MOVE       4 0
    MOVECONST  5 1 //load 2
    MULF       4 4 5
    MOVECONST  5 0 //load 0.7
    CALL       3 3 1
    ADDF       3 3 4
    SHIFTSTATE -1004
    RETURN     3 1
```

```
fn dsp (x)
state_size:4016
    MOVECONST  1 6 //load twodelay
    MOVE       2 0
    MOVECONST  3 3 //load 400
    CALL       1 2 1
    SHIFTSTATE 2008
    MOVECONST  2 6 //load twodelay
    MOVE       2 3
    MOVE       3 0
    MOVECONST  3 4 //load 400
    CALL       2 2 1
    ADD        1 1 2
    SHIFTSTATE -2008
    RETURN     1 1
```

Bytecode Representation of the "twodelay" Example

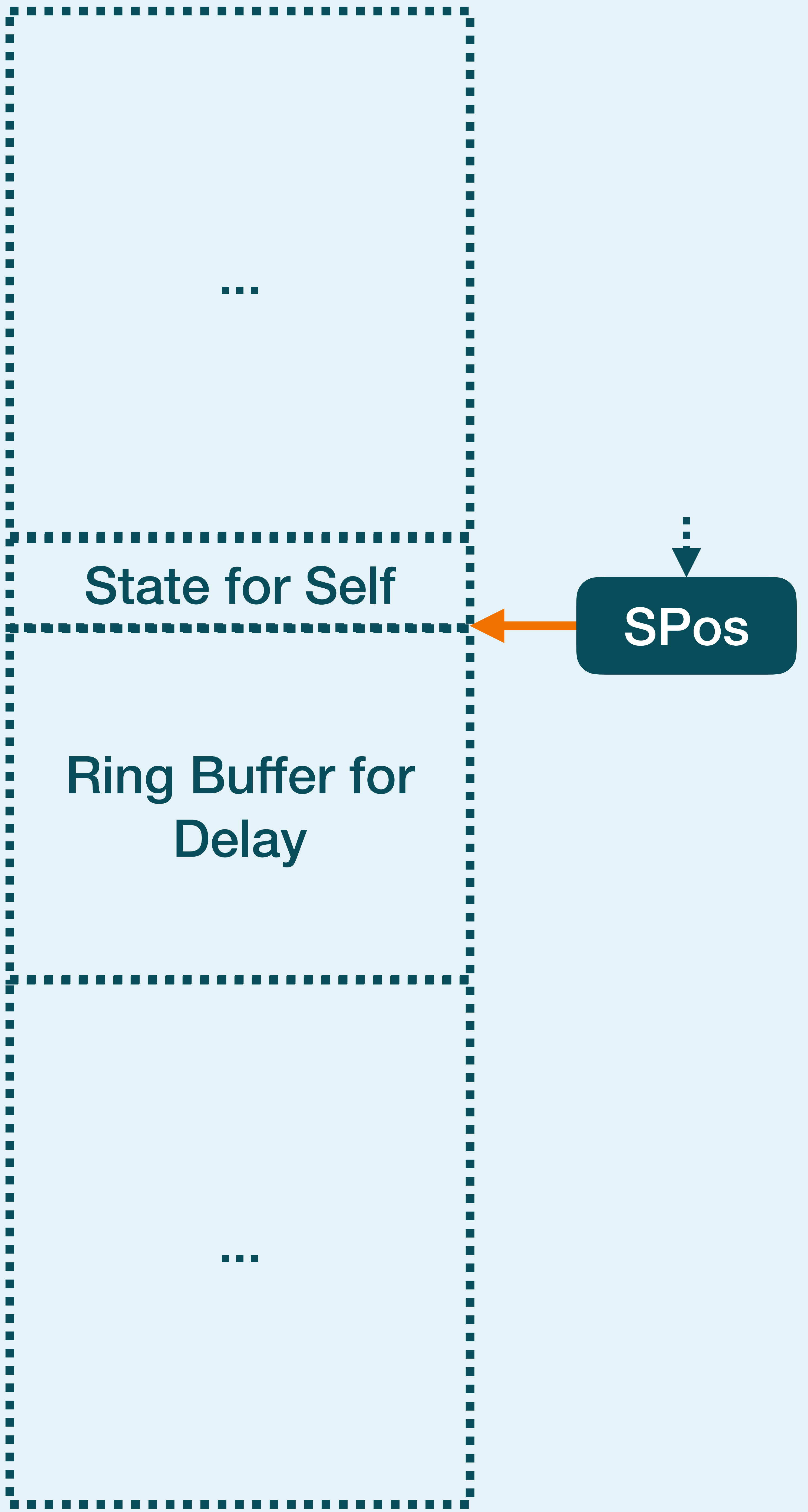



```
fn fbdelay(x,fb,dtime) state_size:1004
    MOVE       3 0 //load x
    GETSTATE   4
    SHIFTSTATE 1
    DELAY      4 4 2
    MOVE       5 1
    MULF       4 4 5
    ADDF       3 3 4
    SHIFTSTATE -1
    GETSTATE   4
    SETSTATE   3
    RETURN     4 1
```

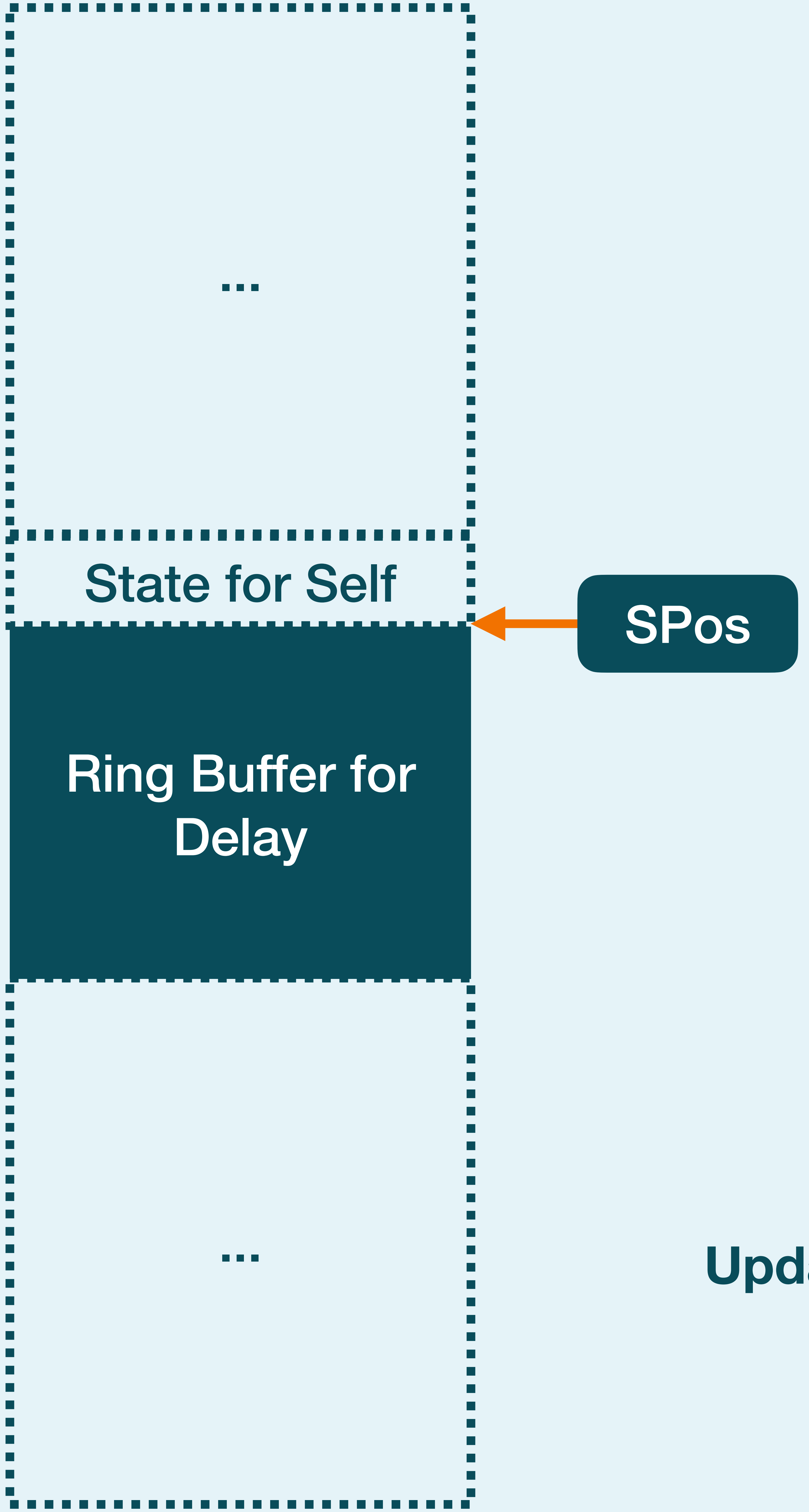



```
fn fbdelay(x,fb,dtime) state_size:1004
    MOVE       3 0 //load x
    GETSTATE   4
    SHIFTSTATE 1
    DELAY      4 4 2
    MOVE       5 1
    MULF       4 4 5
    ADDF       3 3 4
    SHIFTSTATE -1
    GETSTATE   4
    SETSTATE   3
    RETURN     4 1
```

Refer to the "self"
Take one word at SPos, and load to register 4

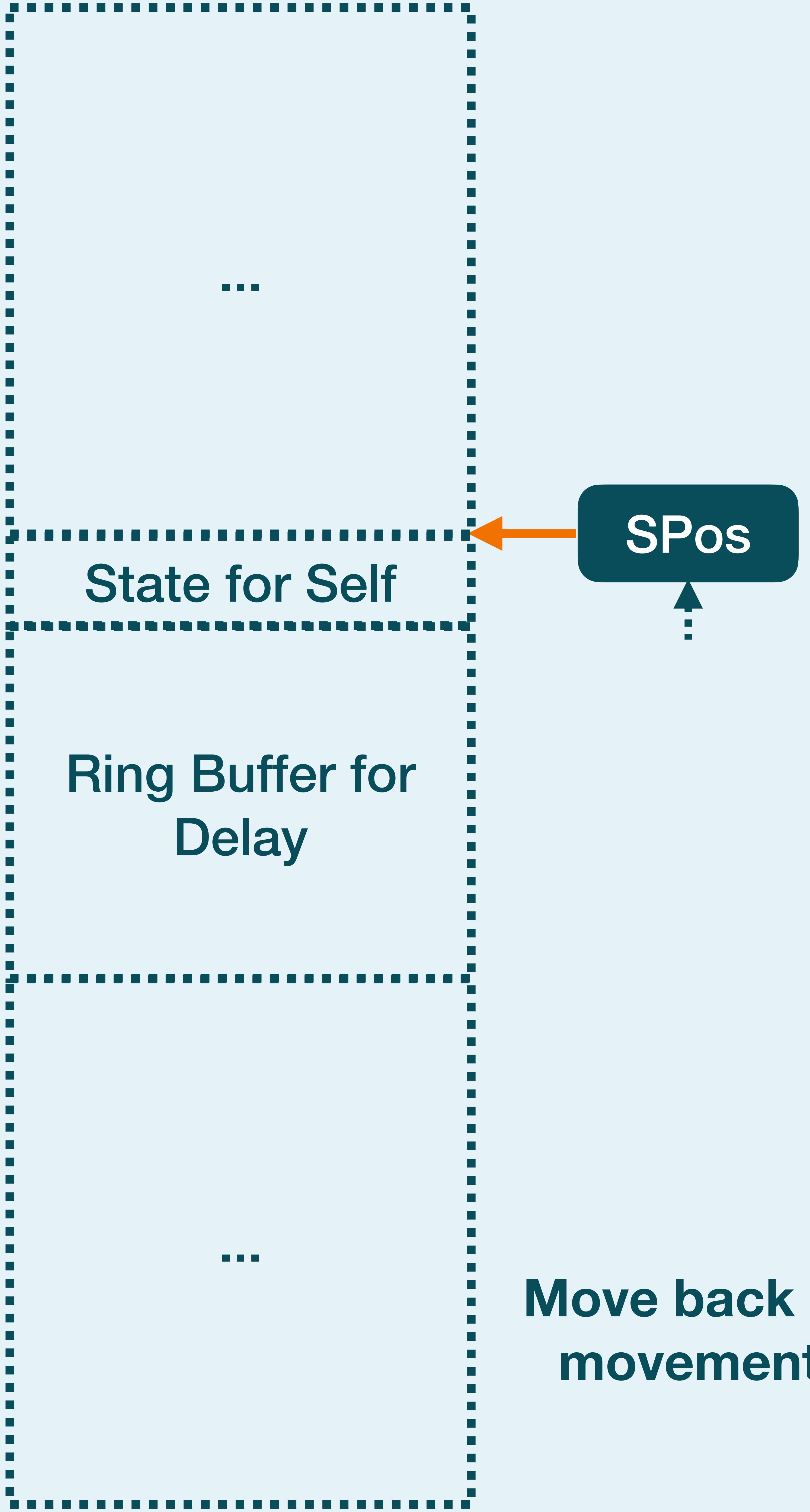



```
fn fbdelay(x,fb,dtime) state_size:1004
    MOVE       3 0 //load x
    GETSTATE   4
    SHIFTSTATE 1
    DELAY      4 4 2
    MOVE       5 1
    MULF       4 4 5
    ADDF       3 3 4
    SHIFTSTATE -1
    GETSTATE   4
    SETSTATE   3
    RETURN     4 1
```



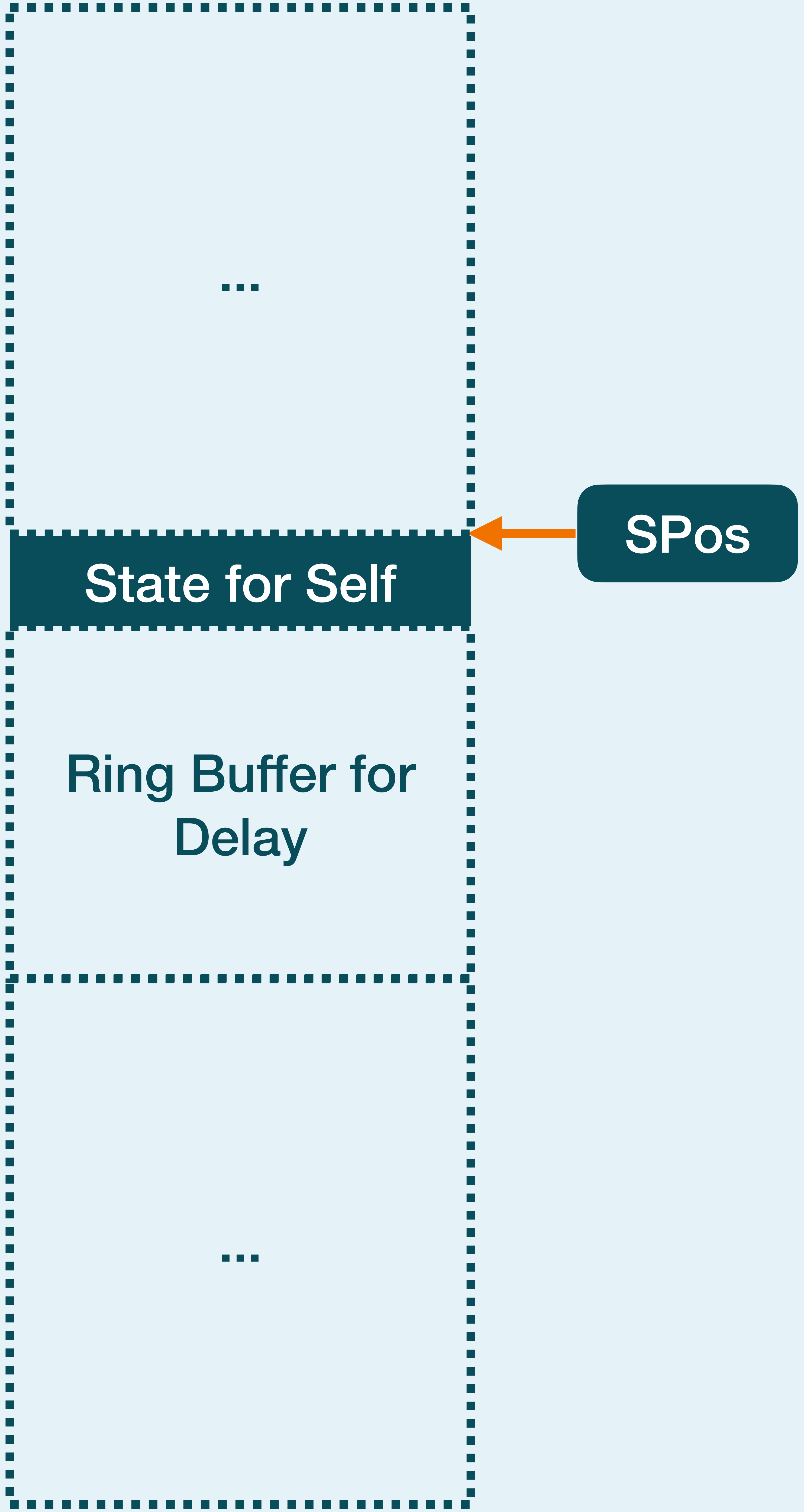

```
fn fbdelay(x,fb,dtime) state_size:1004
    MOVE       3 0 //load x
    GETSTATE   4
    SHIFTSTATE 1
    DELAY      4 4 2
    MOVE       5 1
    MULF       4 4 5
    ADDF       3 3 4
    SHIFTSTATE -1
    GETSTATE   4
    SETSTATE   3
    RETURN     4 1
```

Update a ring buffer at a SPos




```
fn fbdelay(x,fb,dtime) state_size:1004
    MOVE       3 0 //load x
    GETSTATE   4
    SHIFTSTATE 1
    DELAY      4 4 2
    MOVE       5 1
    MULF       4 4 5
    ADDF       3 3 4
    SHIFTSTATE -1
    GETSTATE   4
    SETSTATE   3
    RETURN     4 1
```

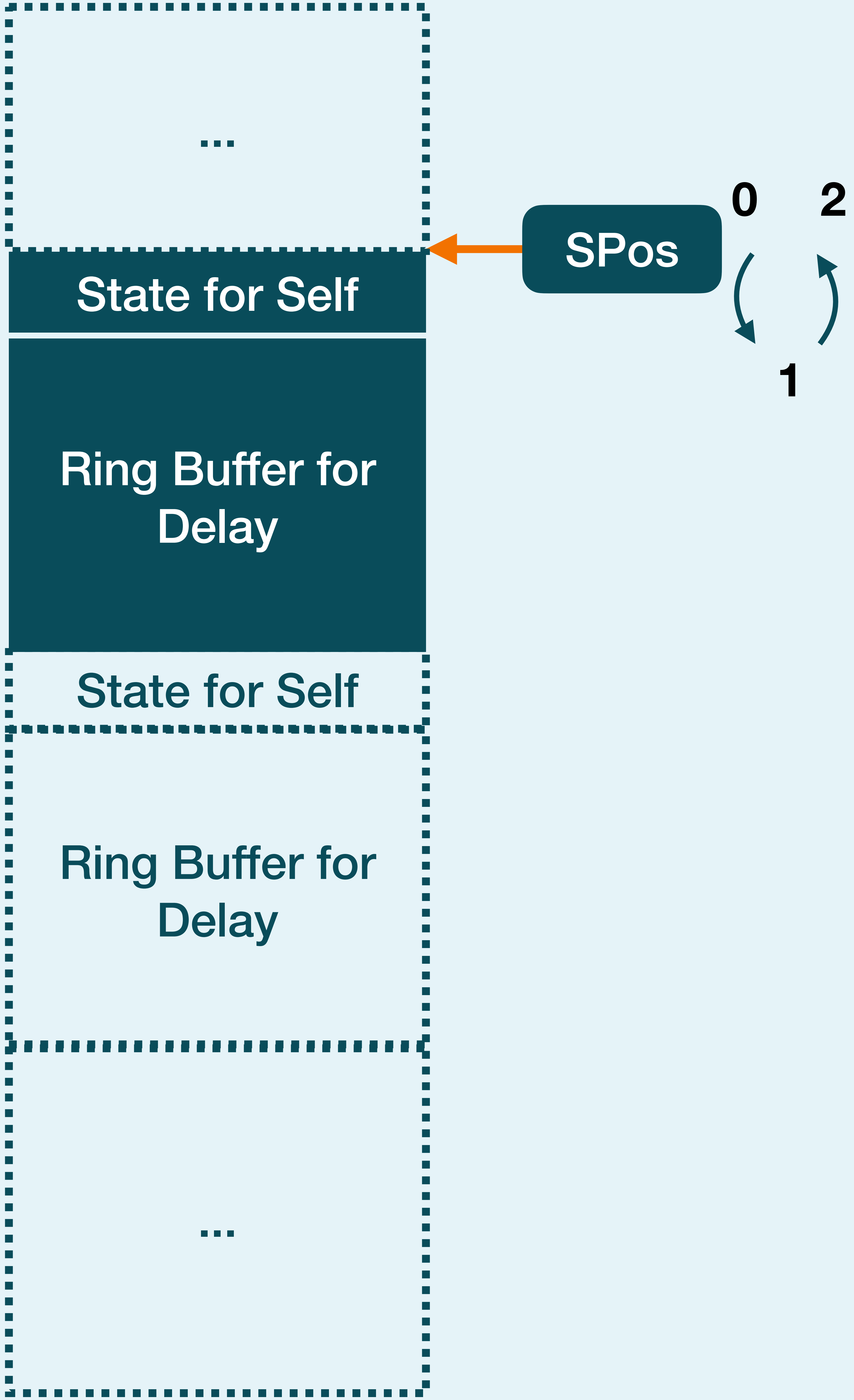
Move back Spos so that the sum of the Spos movement within the function should be 0



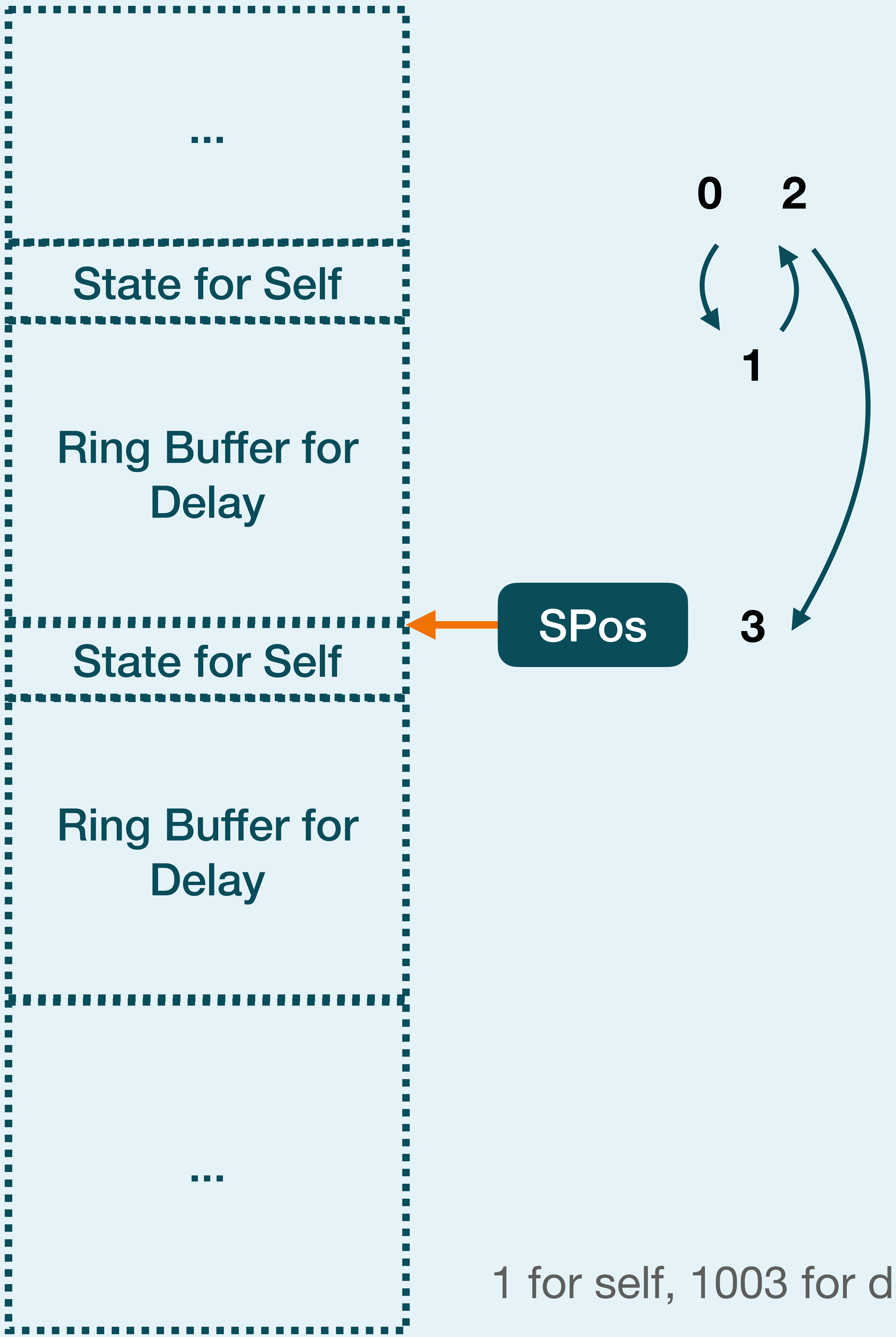

```
fn fbdelay(x,fb,dtime) state_size:1004
    MOVE       3 0 //load x
    GETSTATE   4
    SHIFTSTATE 1
    DELAY      4 4 2
    MOVE       5 1
    MULF       4 4 5
    ADDF       3 3 4
    SHIFTSTATE -1
    GETSTATE   4
    SETSTATE   3
    RETURN     4 1
```

If "self" is used, take the previous return value from Spos, write return value at this time to Spos, and return the previous value from function

Call to the first "fbdelay"



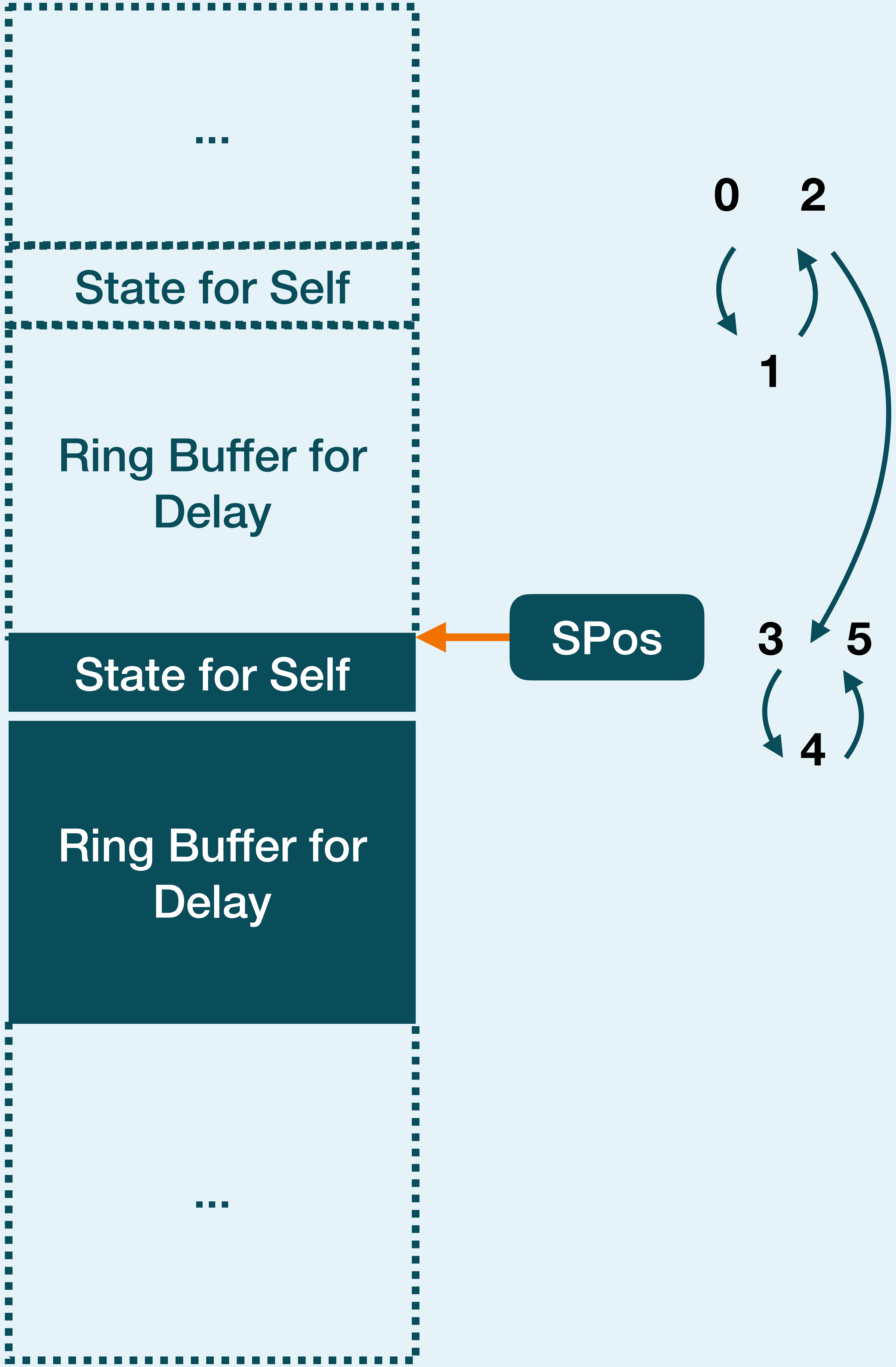

```
fn twodelay(x,dtime) state_size:2008
    MOVECONST  2 5
    MOVE       3 0
    MOVE       4 1
    MOVECONST  5 0
    CALL       2 3 1
    SHIFTSTATE 1004
    MOVECONST  3 5
    MOVE       4 0
    MOVECONST  5 1 //load 2
    MULF       4 4 5
    MOVECONST  5 0 //load 0.7
    CALL       3 3 1
    ADDF       3 3 4
    SHIFTSTATE -1004
    RETURN     3 1
```



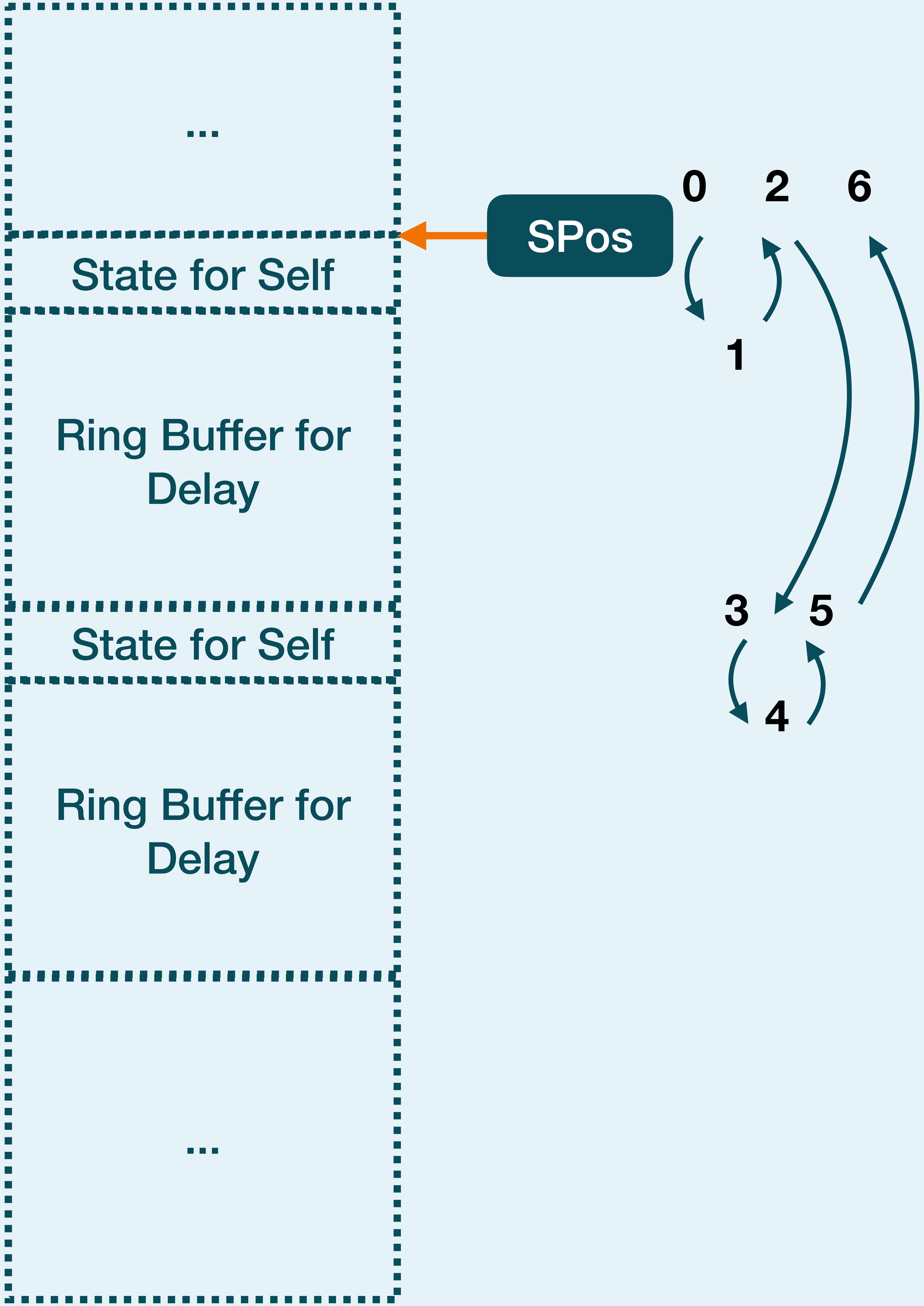

```
fn twodelay(x,dtime) state_size:2008
    MOVECONST  2 5
    MOVE       3 0
    MOVE       4 1
    MOVECONST  5 0
    CALL       2 3 1
    SHIFTSTATE 1004
    MOVECONST  3 5
    MOVE       4 0
    MOVECONST  5 1 //load 2
    MULF       4 4 5
    MOVECONST  5 0 //load 0.7
    CALL       3 3 1
    ADDF       3 3 4
    SHIFTSTATE -1004
    RETURN     3 1
```

1 for self, 1003 for delay(3 for read index, write index, buffer size) => 1004

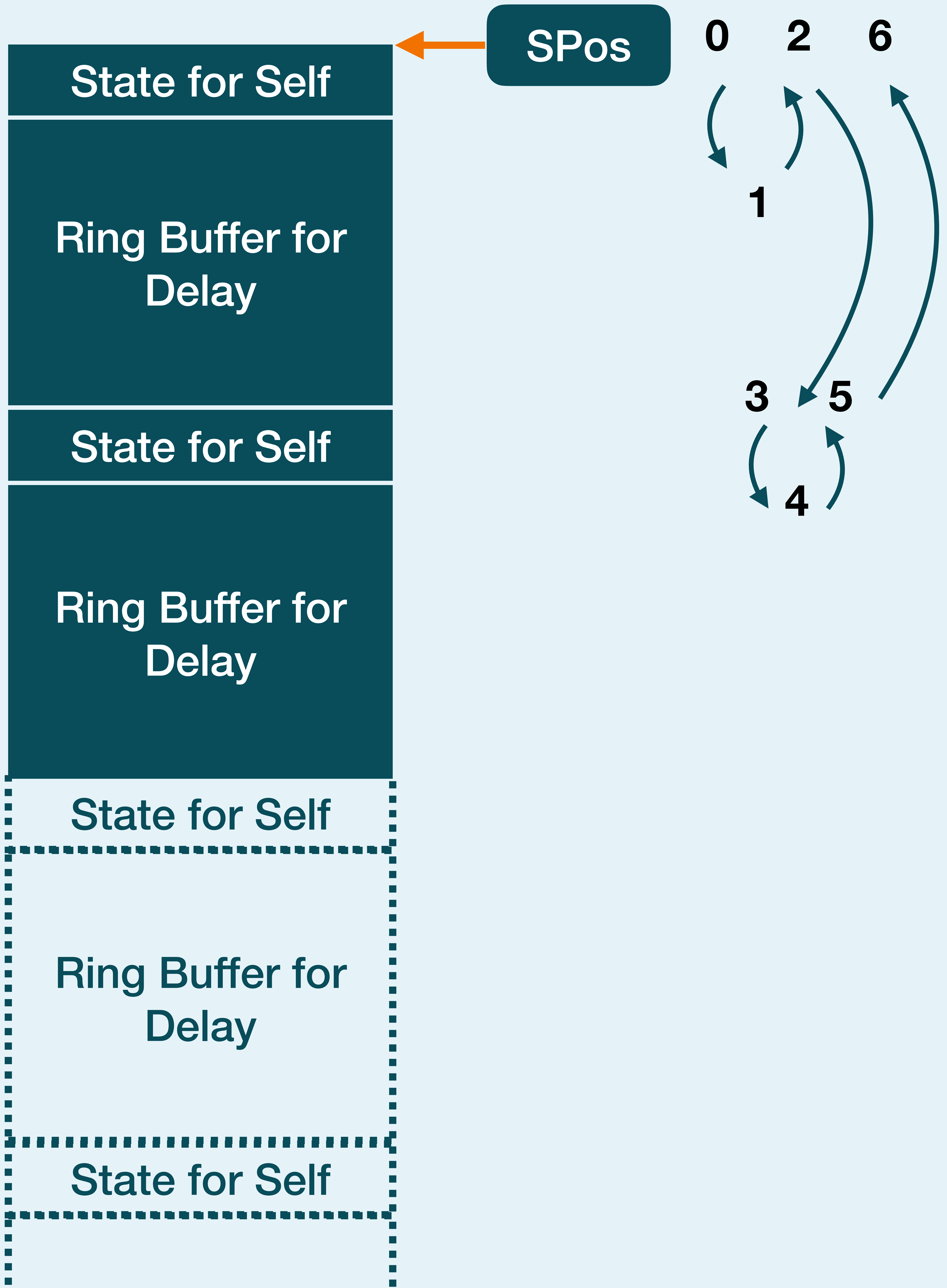
Call to the second "fbdelay"




```
fn twodelay(x,dtime) state_size:2008
    MOVECONST  2 5
    MOVE       3 0
    MOVE       4 1
    MOVECONST  5 0
    CALL       2 3 1
    SHIFTSTATE 1004
    MOVECONST  3 5
    MOVE       4 0
    MOVECONST  5 1 //load 2
    MULF       4 4 5
    MOVECONST  5 0 //load 0.7
    CALL       3 3 1
    ADDF       3 3 4
    SHIFTSTATE -1004
    RETURN     3 1
```

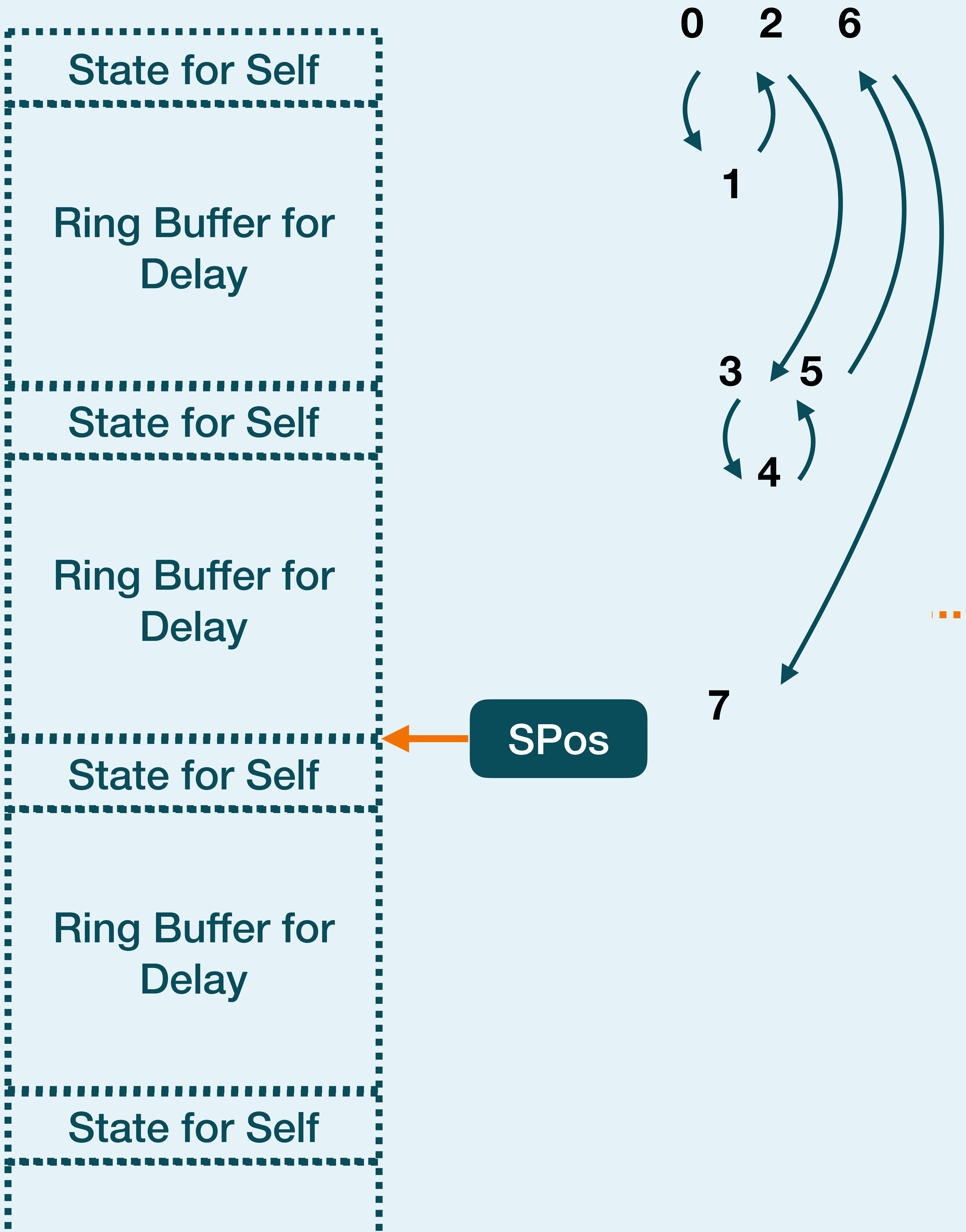



```
fn twodelay(x,dtime) state_size:2008
    MOVECONST  2 5
    MOVE       3 0
    MOVE       4 1
    MOVECONST  5 0
    CALL       2 3 1
    SHIFTSTATE 1004
    MOVECONST  3 5
    MOVE       4 0
    MOVECONST  5 1 //load 2
    MULF       4 4 5
    MOVECONST  5 0 //load 0.7
    CALL       3 3 1
    ADDF       3 3 4
    SHIFTSTATE -1004
    RETURN     3 1
```



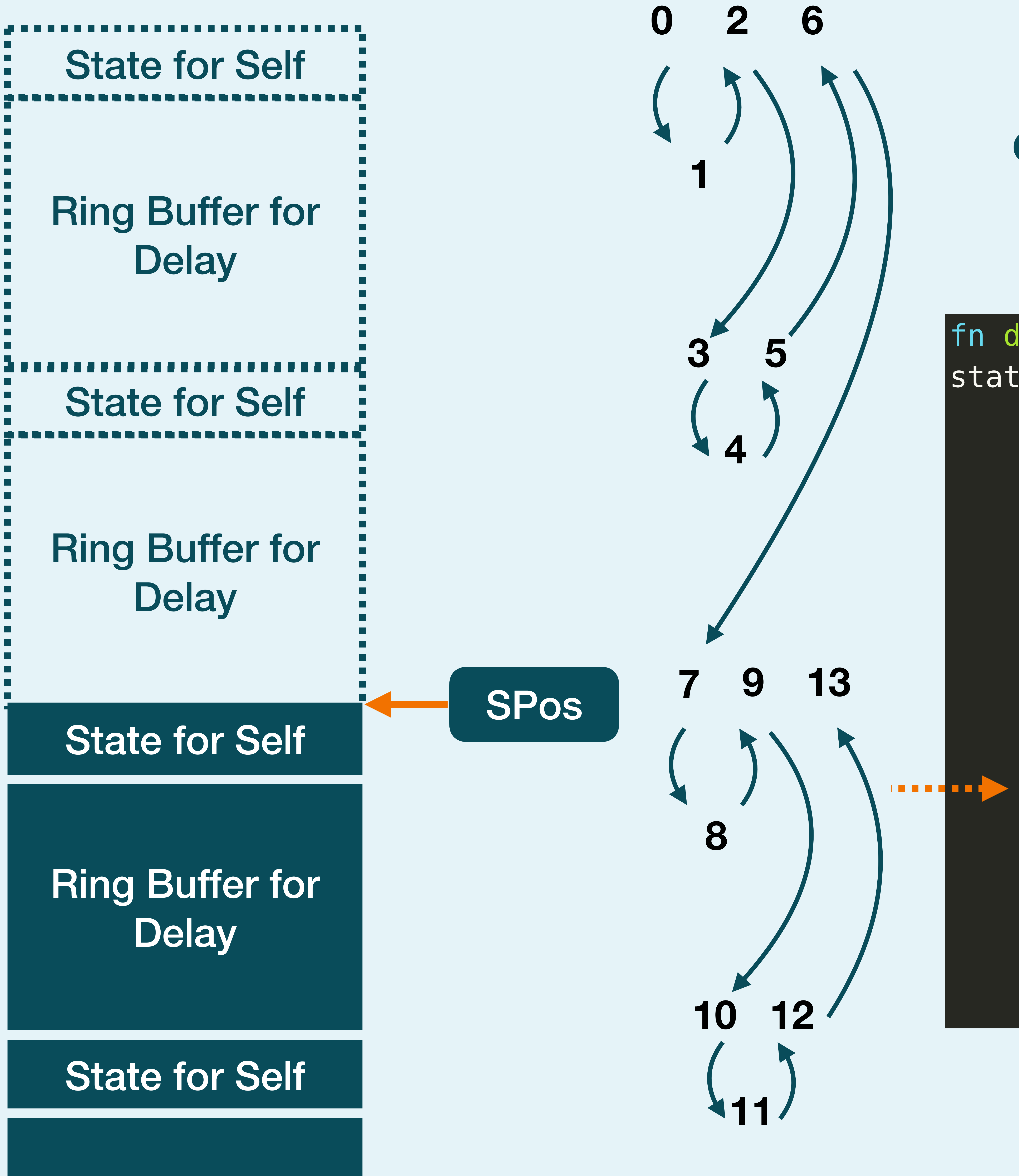
Call to the first "twodelay"

```
fn dsp (x)
state_size:4016
    MOVECONST  1 6 //load twodelay
    MOVE       2 0
    MOVECONST  3 3 //load 400
    CALL       1 2 1
    SHIFTSTATE 2008
    MOVECONST  2 6 //load twodelay
    MOVE       2 3
    MOVE       3 0
    MOVECONST  3 4 //load 400
    CALL       2 2 1
    ADD        1 1 2
    SHIFTSTATE -2008
    RETURN     1 1
```

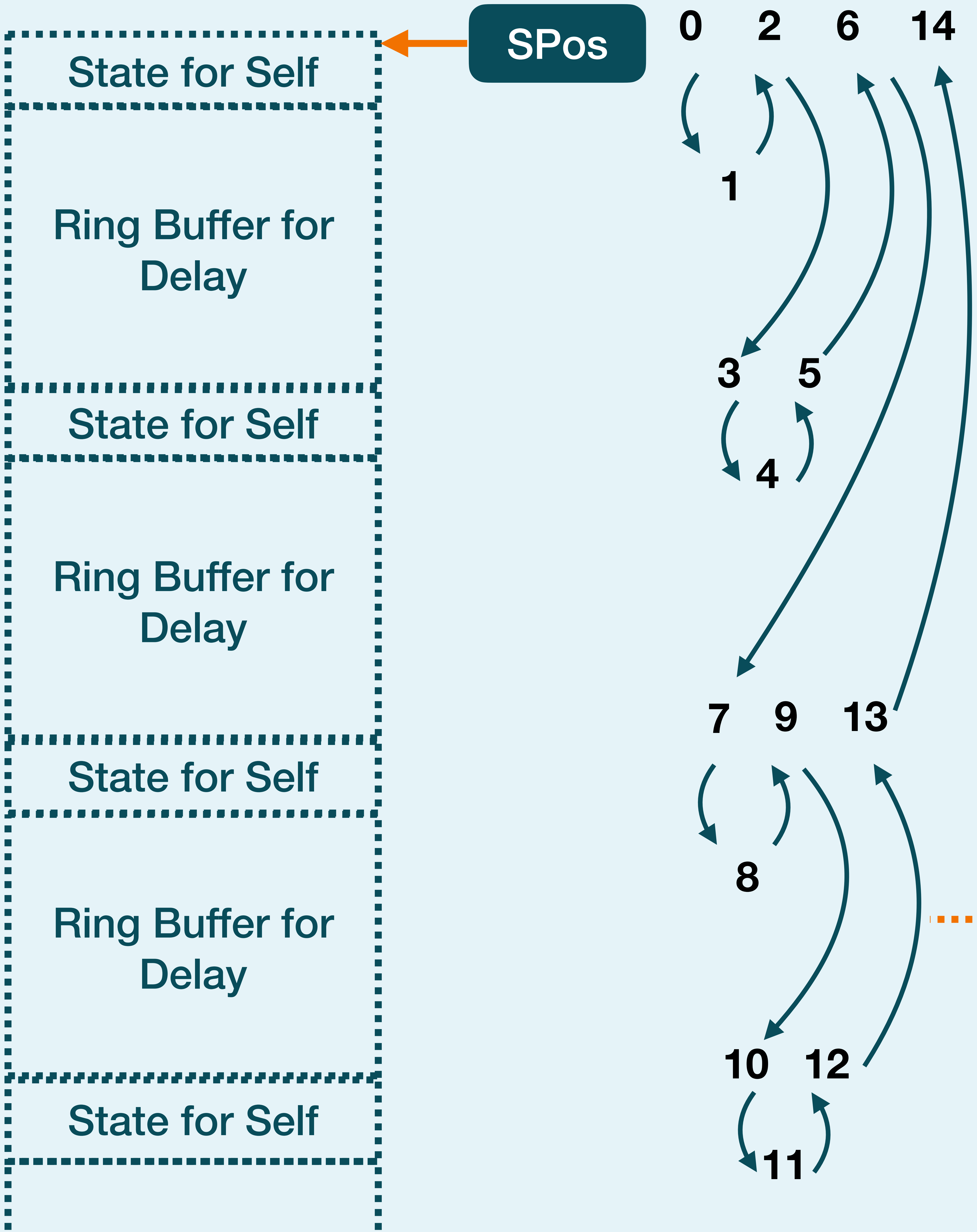



```
fn dsp (x)
state_size:4016
    MOVECONST  1 6 //load twodelay
    MOVE       2 0
    MOVECONST  3 3 //load 400
    CALL       1 2 1
    SHIFTSTATE 2008
    MOVECONST  2 6 //load twodelay
    MOVE       2 3
    MOVE       3 0
    MOVECONST  3 4 //load 400
    CALL       2 2 1
    ADD        1 1 2
    SHIFTSTATE -2008
    RETURN     1 1
```

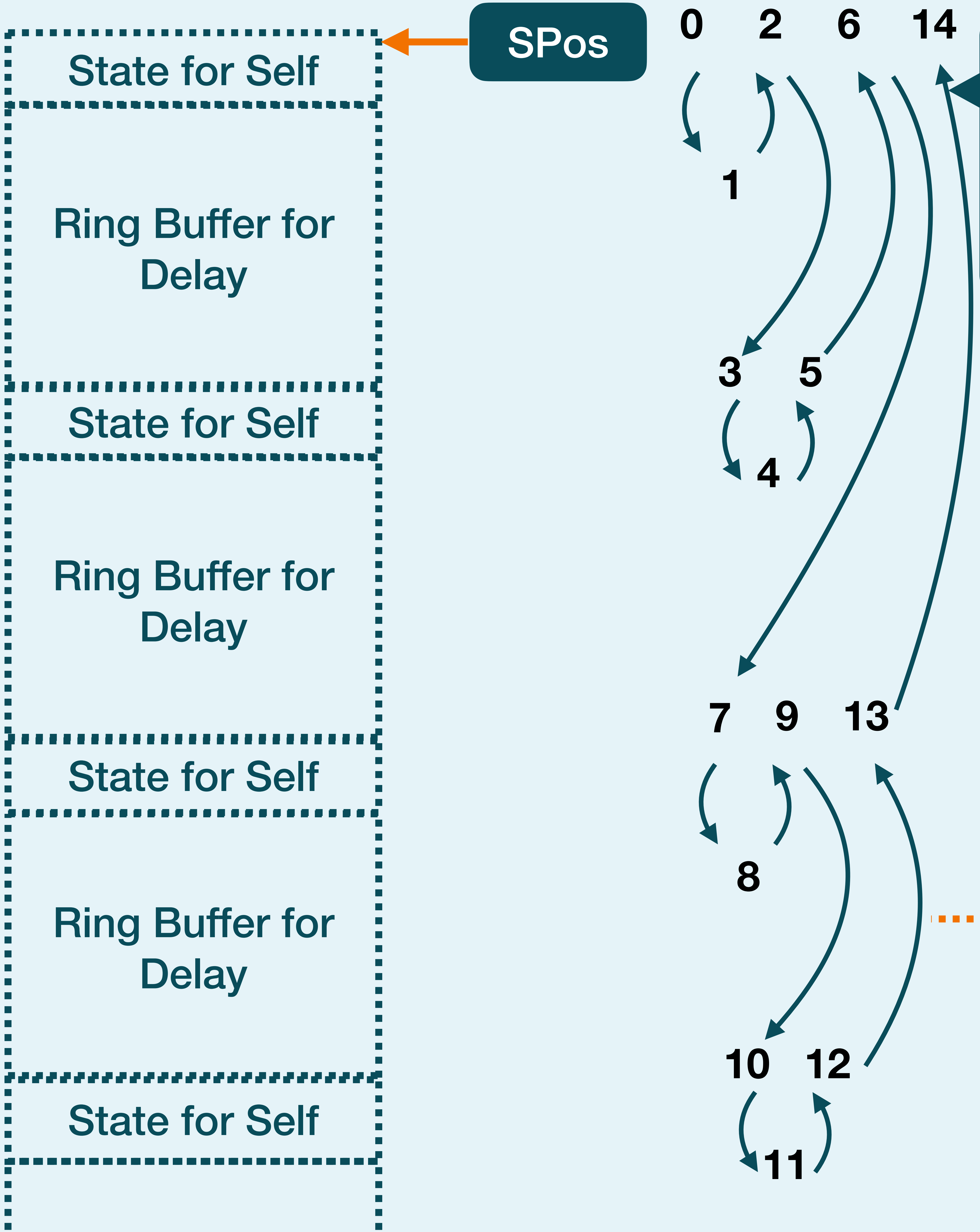
Call to the second "twodelay"




```
fn dsp (x)
state_size:4016
    MOVECONST  1 6 //load twodelay
    MOVE       2 0
    MOVECONST  3 3 //load 400
    CALL       1 2 1
    SHIFTSTATE 2008
    MOVECONST  2 6 //load twodelay
    MOVE       2 3
    MOVE       3 0
    MOVECONST  3 4 //load 400
    CALL       2 2 1
    ADD        1 1 2
    SHIFTSTATE -2008
    RETURN     1 1
```




```
fn dsp (x)
state_size:4016
    MOVECONST  1 6 //load twodelay
    MOVE       2 0
    MOVECONST  3 3 //load 400
    CALL       1 2 1
    SHIFTSTATE 2008
    MOVECONST  2 6 //load twodelay
    MOVE       2 3
    MOVE       3 0
    MOVECONST  3 4 //load 400
    CALL       2 2 1
    ADD        1 1 2
    SHIFTSTATE -2008
    RETURN     1 1
```



By having relative offsets, each functions do not need to care where they are called from

```
fn dsp (x)
state_size:4016
    MOVECONST  1 6 //load twodelay
    MOVE       2 0
    MOVECONST  3 3 //load 400
    CALL       1 2 1
    SHIFTSTATE 2008
    MOVECONST  2 6 //load twodelay
    MOVE       2 3
    MOVE       3 0
    MOVECONST  3 4 //load 400
    CALL       2 2 1
    ADD        1 1 2
    SHIFTSTATE -2008
    RETURN     1 1
```

Combination with Higher-Order Function

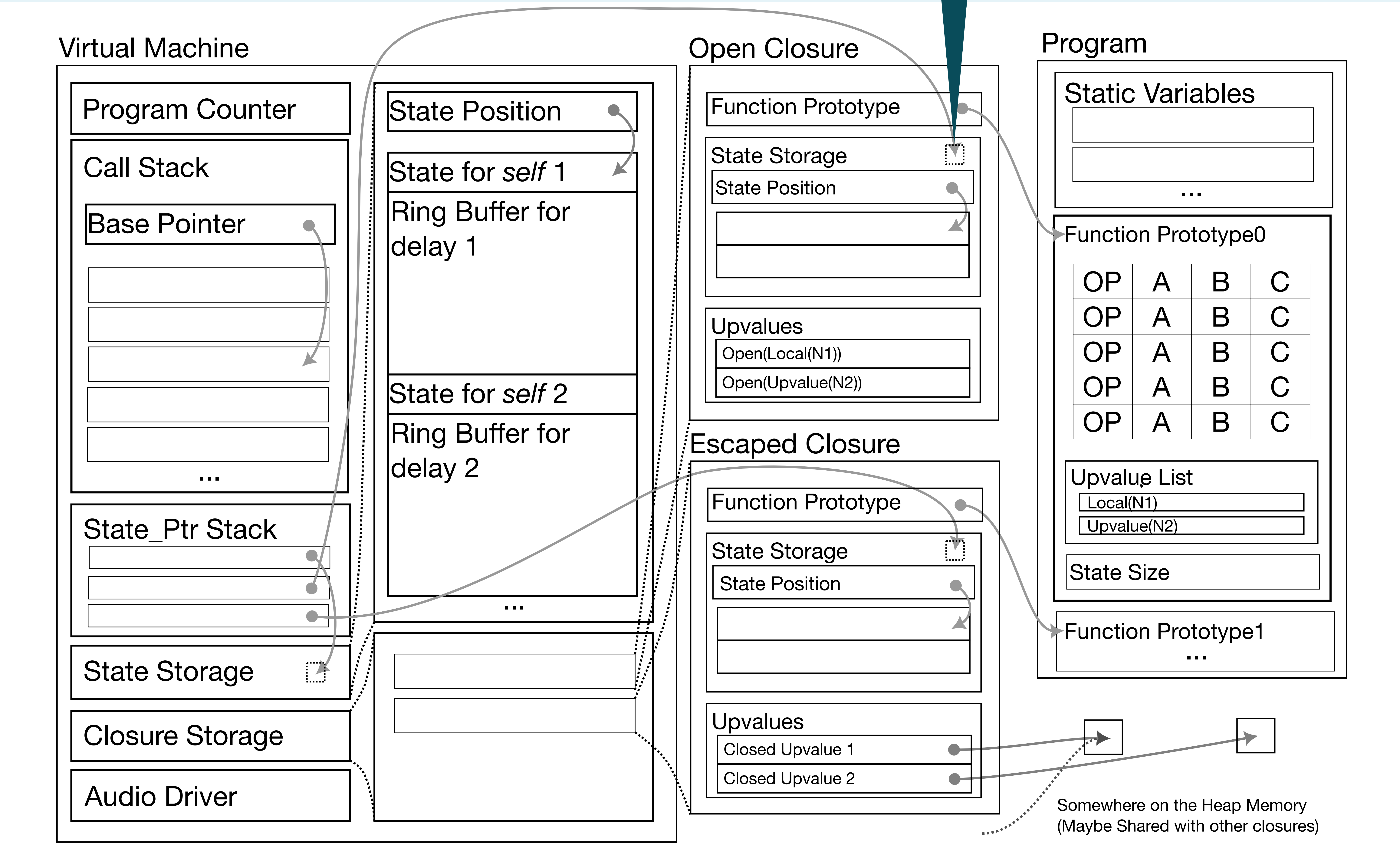
```
fn bandpass(x,freq){
      //...
    }
fn filterbank(n,filter_factory:()->(float,float)->float){
  if (n>0){
    let filter = filter_factory()
    let next = filterbank(n-1,filter_factory)
    |x,freq| filter(x,freq+n*100)
             + next(x,freq)
  }else{
    |x,freq| 0
  }
}
let myfilter = filterbank(3,| | bandpass)
fn dsp(){
      myfilter(x,1000)
}
```

Combination with Higher-Order Function

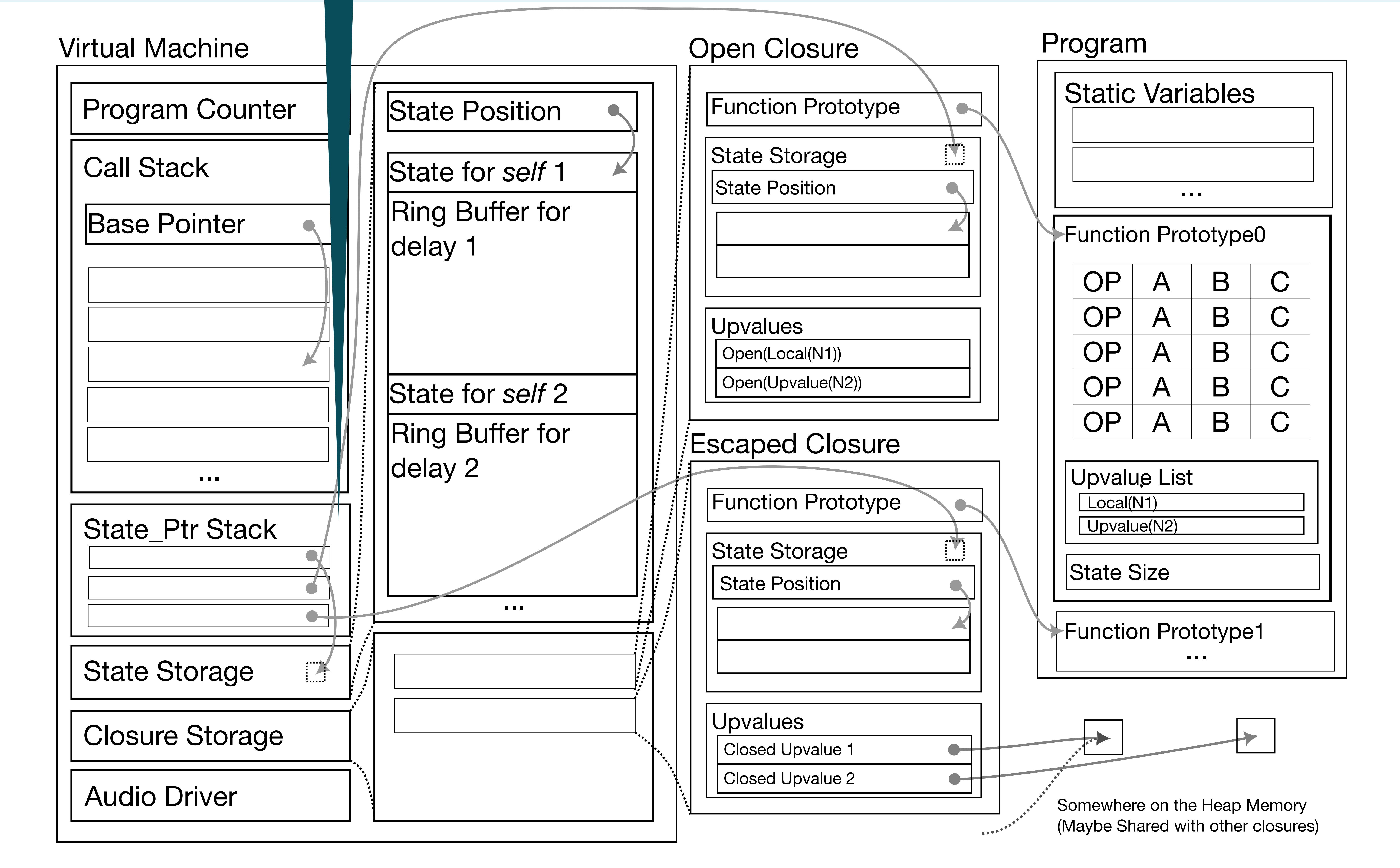
```
fn bandpass(x,freq){
      //...
    }
fn filterbank(n,filter_factory:()->(float,float)->float){
  if (n>0){
    let filter = filter_factory()
    let next = filterbank(n-1,filter_factory)
    |x,freq| filter(x,freq+n*100)
             + next(x,freq)
  }else{
    |x,freq| 0
  }
}
let myfilter = filterbank(3,| | bandpass)
fn dsp(){
      myfilter(x,1000)
}
```

The size of the internal state variable for "filter_factory" is not determined at a compile time.

When the closure is made with CLOSURE instruction, it allocates storage for internal state variables individually



When CALLCLS is used, VM pushes the pointer to closure's state storage to the stack, to switch which storage are used in GET/SET/SHIFTSTATE operations



```
CONSTANTS[100,1,0,2]
fn inner_then(x,freq)
    //upvalue:
[local(4),local(3),local(2),local(1)]
    GETUPVALUE 3 2 //load filter
    MOVE       4 0
    MOVE       5 1
    GETUPVALUE 6 1 //load n
    ADDD       5 5 6
    MOVECONST  6 0
    MULF       5 5 6
    CALLCLS    3 2 1  //call filter
    GETUPVALUE 4 4 //load next
    MOVE       5 0
    MOVE       6 1
    CALLCLS    4 2 1 //call next
    ADDF       3 3 4
    RETURN     3 1

fn inner_else(x,freq)
    MOVECONST  2 2
    RETURN     2 1
```

```
fn filterbank(n,filter_factory)
    MOVE      2 0 //load n
    MOVECONST 3 2 //load 0
    SUBF      2 2 3
    JMPIFNEG  2 12
    MOVE      2 1 //load filter_factory
    CALL      2 2 0 //get filter
    MOVECONST 3 1 //load itself
    MOVE      4 0 //load n
    MOVECONST 5 1 //load 1
    SUBF      4 4 5
    MOVECONST 5 2 //load inner_then
    CALLCLS   3 2 1 //recursive call
    MOVECONST 4 2 //load inner_then
    CLOSURE   4 4 //load inner_lambda
    JMP       2
    MOVECONST 4 3 //load inner_else
    CLOSURE   4 4
    CLOSE     4
    RETURN    4 1
```

There are no "GET/SET/SHIFTSTATE" operation here!

Combination with Higher-Order Function

```
fn bandpass(x,freq){
      //...
    }
fn filterbank(n,filter_factory:()->(float,float)->float){
  if (n>0){
    let filter = filter_factory()
    let next = filterbank(n-1,filter_factory)
    |x,freq| filter(x,freq+n*100)
             + next(x,freq)
  }else{
    |x,freq| 0
  }
}
let myfilter = filterbank(3,| | bandpass)
fn dsp(){
      myfilter(x,1000)
}
```

This works like a constructor of Unit Generator, in the object-oriented programming world

5. Discussion

- Comparison to the other languages
- Counterintuitive behavior of higher order functions
- Foreign stateful function call

Comparison to the other languages

	Parametric Signal Graph	Actual DSP
Faust	Term Rewriting Macro	BDA
Kronos	Type-level Computation	Value Evaluation
mimium	Global Context Execution	`dsp` Function Execution

Both are same semantics in the value level.

This will make it easier to understand for novice users **but...**

This code does not work:

```
fn filterbank(n,filter){
  if (n>0){
    |x,freq| filter(x,freq+n*100)
    + filterbank(n-1,filter)(x,freq)
  }else{
    |x,freq| 0
  }
}
fn dsp(){
  filterbank(3,bandpass)(x,1000)
}
```

This code does not work:

```
fn filterbank(n,filter){
  if (n>0){
    |x,freq| filter(x,freq+n*100)
    + filterbank(n-1,filter)(x,freq)
  }else{
    |x,freq| 0
  }
}
fn dsp(){
  filterbank(3,bandpass)(x,1000)
}
```

These part re-instantiates the closure with zero-initiallized state variables every samples

This code still does not work:

```
fn filterbank(n,filter){
  let next = filterbank(n-1,filter)
  if (n>0){
    |x,freq| filter(x,freq+n*100)
      + next(x,freq)
  }else{
    |x,freq| 0
  }
}
let myfilter = filterbank(3,bandpass)
fn dsp(){
      myfilter(x,1000)
 }
```

This code still does not work:

```
fn filterbank(n,filter){
  let next = filterbank(n-1,filter)
  if (n>0){
    |x,freq| filter(x,freq+n*100)
      + next(x,freq)
  }else{
    |x,freq| 0
  }
}
let myfilter = filterbank(3,bandpass)
fn dsp(){
      myfilter(x,1000)
 }
```

This code shares the same instance of the closure and updated multiple times at a sample

*This behavior could be fixed by changing the closure to be “deep-copied” when passed as an argument to HOF.

If the Multi-Stage Programming can be used:

```
fn filterbank(n,filter:&(float,float)->float)->&(float,float)->float{
  .< if (n>0){
    |x,freq| ~filter(x,freq+n*100)
      + ~filterbank(n-1,filter)(x,freq)
  }else{
    |x,freq| 0
  } >.
}
fn dsp(){
  ~filterbank(3,.<bandpass>.)(x,1000)
}
```

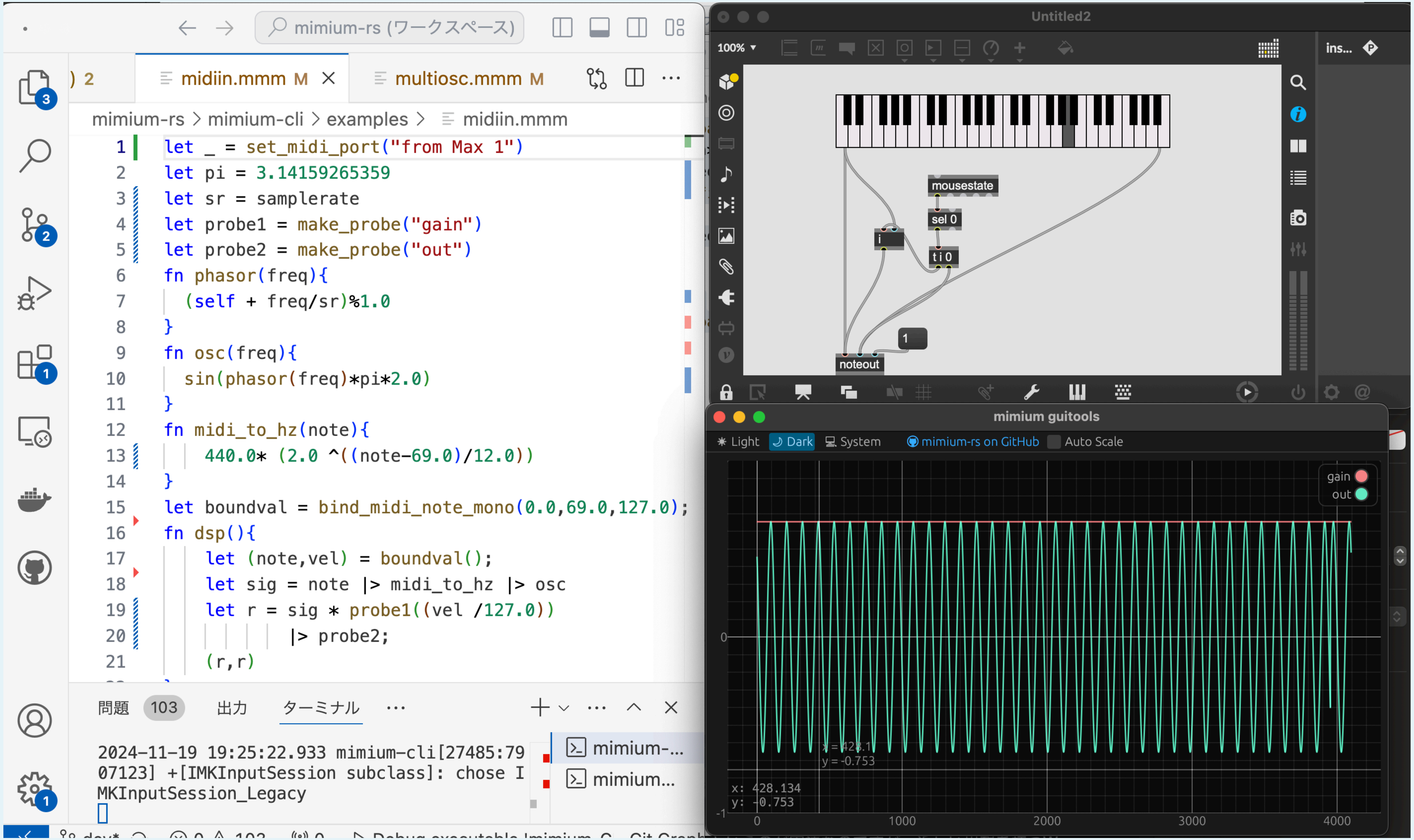
*This is a pseudo-code, based on the syntax of BER MetaOCaml

Considering on a multi-stage computation

- Question: When should we evaluate stage-0. At AST or Bytecode?
  - If the former, we have to implement two different evaluators.
  - If the latter, we have to translate multi-stage computation semantics into imperative world somehow. *I'm going to this choice currently
- Is the syntax of multi-stage computation really easy to understand for novices, than the type-level computation in Kronos or the term rewriting macro in Faust?

Foreign stateful function calls

- Because the closure works like Unit Generator in the OOP world, mimium can call UGen defined in the native code with small wrapper naturally.
  - though it will not work for vector-by-vector processing correctly.



In fact, some external modules like MIDI and Instant oscilloscope
(written in Rust) are used with higher-order function pattern

Wrap-up

- λmmm: an extended call-by value lambda calculus, that adds "delay" and "feed"
- Proposed VM and Instruction set for it
  - GET/SET/SHIFTSTATE to handle "delay" and "feed"
  - A closure instance holds a memory for state variables for "delay" and "feed" to handle a higher-order function with stateful functions.
- Resulted in unified semantics for both parametric signal graph generation and actual execution of the graph
  - This makes it easier to understand semantics but the users have to be responsible to distinct whether the function is evaluated in global context once or in "dsp" function iteratively
- Domain-Specific, but not loosing generality, self-extensibility and interoperability

Thank you for listening.

Development repository of mimium v2 (written in Rust)

https://github.com/tomoyanonymous/mimium-rs

email: me@matsuuratomoya.com
mastodon: social.matsuuratomoya.com/@tomoya