λ_{mmm}: THE INTERMEDIATE REPRESENTATION FOR SYNCHRONOUS SIGNAL PROCESSING LANGUAGE BASED ON LAMBDA CALCULUS

Tomoya Matsuura

Tokyo University of the Arts
Tokyo, Japan
me@matsuuratomoya.com

ABSTRACT

This paper proposes λ_{mmm}: a call-by-value simply-typed lambda calculus-based intermediate representation of a programming language for music that deal with synchronous signal processing, as well as a virtual machine and instruction set to run λ_{mmm}. Digital signal processing can be represented with a syntax that incorporates the internal states of delay and feedback into the lambda calculus. λ_{mmm} is a superset of the lambda calculus, which allows users to construct a generative signal processing graph and its execution in identical semantics. On the other hand, due to its specification, a problem was found that when dealing with higher-order functions, the users have to determine whether the execution is in the global environment evaluation or the DSP execution, and it is implied that the multi-stage computation can solve the issue.

1. INTRODUCTION

Many programming languages for sound and music have been developed, but only a few have strongly formalized semantics. One language that is both rigorously formalized and practical is Faust[1], which combines blocks with inputs and outputs with five primitive operations - parallel, sequential, split, merge, and recursive connection. By having basic arithmetics, conditional and delay as primitive blocks, any type of signal processing can be written in Faust. In a later extension, a macro based on a term rewriting system has been introduced, allowing users to parameterize blocks with an arbitrary number of inputs and outputs[2].

This strong abstraction capability through formalization enables Faust to be translated to various backends, such as C, C++, Rust and LLVM IR. On the other hand, BDA lacks theoretical and practical compatibility with common programming languages. Although it is possible to call external C functions in Faust, those functions are assumed to be pure functions without a heap memory allocation and deallocation. Therefore, while it is easy to embed Faust in another language, it is not easy to call another language from Faust.

In addition, a macro for Faust is an independent term rewriting system that generates BDA based on a pattern matching. Therefore, the arguments for pattern matching are implicitly required to be an integer, which sometimes causes compile errors despite the distinction between real and integer types does not exist in BDA. These implicit typing rules are not intuitive for novice users.

Proposing a computational model for signal processing based on the more generic computational models, such as a lambda calculus has the potential to interoperate between many different general purpose languages on run-time, and also facilitate the appropriation of existing optimization methods and the implementation of compilers and run-time.

Currently, it has been proved that BDA can be converted to a general-purpose functional language in the form of using arrow, a higher-level abstraction of monads[3]. However, higher-order functions on general-purpose functional languages are often implemented on the basis of dynamic memory allocation and release, which makes it difficult to use them in host languages for real-time signal processing.

Also, Kronos[4] and W-calculus[5] are examples of attempts at lambda calculus-based abstraction while being influenced by Faust. Kronos is based on the theoretical foundation of System-$F\omega$, a variation of lambda calculus in which the type itself can be abstractized (a function that takes the type as an input and returns a new type can be defined). In Kronos, a calculation of type corresponds to the signal graph generation and a calculation of value corresponds to the actual processing. A special primitive operation is only delay in Kronos, and the feedback routing can be represented as a recursive function application in a calculation of types.

The W-calculus has feedback operation as a primitive, along with the access to the value of the variable in the past(= delay). W-calculus limits the systems so that can represents to linear-time-invariant such as filters and reverbrators and defines a more formal semantics, aiming at automatic proofs of the linearity and identity of graph topologies.

Previously, the author designed a programming language for music *mimium* [6]. By adding basic operations of delay and feedback to lambda calculus, signal processing can be concisely expressed while having a syntax close to that of general-purpose programming languages (especially, the syntax of mimium is designed to be looks like Rust language).

One of the previous issues with mimium was the inability to compile the codes, which contain a combination of recursive or higher-order functions with stateful functions that contains delay or feedback because the compiler could not determine the data size of the internal state of the signal processing.

In this paper, I propose a syntax and semantics of λ_{mmm}, an extended call-by-value simply-typed lambda calculus, as a computation model that is supposed to be an intermediate representation for mimium. Also, I propose a virtual machine and its instruction set based on Lua's VM, to execute this computation model practically. Lastly, I discuss the problem and the potential, one that the current λ_{mmm} that users have to care whether the calculation happens in a global context or an acutal signal processing, another that the run-time interoperation between the other programming languages can be easier than the existing DSP languages.

2. SYNTAX

Definition of types and terms of the λ_{mmm} are shown in Figure 1.

$$
\begin{array}{llll}
\tau_p ::= R & [real] & v_p ::= r & r \in \mathbb{R} \\
\quad \mid N & [nat] & \quad \mid n & n \in \mathbb{N} \\
\tau ::= \tau_p & & v ::= v_p & \\
\quad \mid \tau \rightarrow \tau & [function] & \quad \mid cls(\lambda\, x.e, E) & \\
\end{array}
$$

Types Values

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

Terms

Figure 1: *Definition of Types, Values and Terms of the λ_{mmm}(Basic arithmetics are omitted).*

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

Listing 1: *Example of the code of one-pole filter in mimium.*

Two terms are added in addition to the usual simply-typed lambda calculus, $delay\, n\, e_1\, e_2$ that refers a previous value of e_1 to e_2 sample before (with maximum delay value n to limit memory size to finite), and $feed\, x.e$ abstraction that the user can refer the result of the evaluation of the e one unit time before as x during the evaluation of e itself.

2.1. Syntactic sugar of the feedback expression in mimium

mimium by the author has a keyword $self$ that can be used in function definition, that refers to the previous return value of the function. The example code of the simple one-pole filter function that mixes input and last output signal so as to a sum of gains of input and feedback should be 1, is shown in Listing 1. This code can be expressed in λ_{mmm} as Figure 2.

$$
\begin{array}{l}
let\ onepole = \\
\qquad \lambda x.\lambda g.\ feed\, y.\ x * (1.0 - g) + y * g\ in\ ...
\end{array}
$$

Figure 2: *Equivalent expression to Listing 1 in λ_{mmm}.*

2.2. Typing Rule

Additional typing rules to usual simply-typed lambda calculus are shown in Figure 3.

$$
\frac{\Gamma, x : \tau_a \vdash e : \tau_b}{\Gamma \vdash \lambda x.e : \tau_a \rightarrow \tau_b} \qquad \frac{\Gamma \vdash e_1 : N \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash delay\ e_1\ e_2 : \tau}
$$

T-LAMBDA T-DELAY

$$
\frac{\Gamma, x : \tau_p \vdash e : \tau_p}{\Gamma \vdash feed\, x.e : \tau_p} \qquad \frac{\Gamma \vdash e_c : R\ \Gamma \vdash e_t : \tau\ \Gamma \vdash e_e : \tau}{\Gamma \vdash if\ (e_c)\ e_t\ e_e\ : \tau}
$$

T-FEED T-IF

Figure 3: *Excerpt of the typing rules for λ_{mmm}.*

As primitive types, there are a real number type to used in most of signal processing and a natural number type that is used for the indice of delay.

In the W-calculus, which is a direct inspiration to designing λ_{mmm}, function types can takes only tuples of real numbers and return tuples of real numbers. This means that higher-order functions cannot be written. While this restriction is reasonable as a design choice for a language for signal processing since higher-order functions require data structures that require dynamic memory allocation, such as closures, for their implementation, it also lacks the generality of the lambda calculus.

In λ_{mmm}, the problem of memory allocation for closures is left to the implementation of the runtime in the Section4, and higher-order functions are allowed. However, the $feed$ abstraction does not allow function types as its input and output. Allowing the return of function types in the $feed$ abstraction means that it is possible to define functions whose processing contents may change time-to-time. While this may be interesting theoritically, there are currently no practical cases in real-world signal processing, and it is expected to further complicate implementations.

3. SEMANTICS

The excerpt of operational semantics of the λ_{mmm} is shown in Figure 4. This big-step semantics is a conceptual explanation of the evaluation that, when the current time is n, the previous evaluation environment t samples before can be referred to as E^{n-t} , and that when the time < 0, the evaluation of any term is evaluated to the default value of its type (0 for the numeric types).

Of course, if we tried to execute this semantics in a straightforward manner, we would have to redo the calculation from time 0 to the current time every sample, with saving all the variable environments at each sample. In practice, therefore, a virtual machine is defined that takes into account the internal memory space used by $delay$ and $feed$, and the λ_{mmm} terms are converted into instructions for that machine before execution.

4. VM MODEL AND INSTRUCTION SET

A model for the virtual machine and its instruction set to run λ_{mmm} is based on the VM for Lua version 5[7].

When executing a computational model based on lambda calculus, the problem is how to handle a data structure called a closure that captures the variable environment where the inner function is defined, to refer the outer variables from the inner function context. If the dictionary data of names and values of variables are paired

$$\frac{E^n \vdash e_2 \Downarrow v_d \; n > v_d \; E^{n-v_d} \vdash e_1 \Downarrow v}{E^n \vdash \; delay \; n \; e_1 \; e_2 \Downarrow v}$$

E-DELAY

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

E-LAM

$$\frac{E^{n-1} \vdash e \Downarrow v_f \; E^n, x \mapsto v_f \vdash e \Downarrow v}{E^n \vdash \; feed \; x.e \Downarrow v}$$

E-FEED

$$\frac{E^n \vdash e_c \Downarrow n \quad n > 0 \; E^n \vdash e_t \; \Downarrow v}{E^n \vdash \; if(e_c) \; e_t \; else \; e_e \Downarrow v}$$

E-IFTRUE

$$\frac{E^n \vdash e_c \Downarrow n \quad n \leqq 0 \; E^n \vdash e_e \; \Downarrow v}{E^n \vdash \; if(e_c) \; e_t \; else \; e_e \Downarrow v}$$

E-IFFALSE

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

E-APP

Figure 4: *Excerpt of the big-step semantics of* λ_{mmm}.

with inner function, implementation of the compiler (intepreter) is simple, but run-time performance is limited.

On the contrary, a runtime performance can be improved by performing a process called closure transformation (or lambda lifting), which analyses all the names of outer variables referred by the inner function and transforms the inner function by adding argument so that the variables can be referred explicitly, but the compiler implementation of the transformation is relatively complex.

The Lua VM takes an intermediate approach between these two by adding the VM instructions `GETUPVALUE` / `SETUPVALUE`, which allows the outer variables to be referred dynamically at runtime. The implementation of compiler and VM using *upvalue* is simpler than closure conversion, while at the same time preventing execution performance degradation, as outer variables can be referred via the call stack rather than on the heap memory unless the closure object escapes from the context of the original function[8].

Also, upvalue helps interoperations between other programming languages, as Lua can be easily embedded through C language API and when implementing external libraries in C, programmer can access to upvalues of Lua Runtime not only the stack values in C API.

4.1. Instruction Set

VM Instructions for λ_{mmm} differs from the Lua VM in the following respects.

1. Since mimium is a statically typed language unlike Lua, instructions for basic arithmetics are provided for each type.
2. The call operation is separated into the normal function call and the call of closure due to its static typing similarly, and also to handle higher-order statefull functions(See 4.2 for details).
3. If statements are realised by a combination of two instructions, `JMP` and `JMPIFNEG`, whereas the Lua VM uses a dedicated `TEST` instructions.
4. Instructions related to for loop, the `SELF` instruction used for object-oriented programming and the `TABLE`-related instructions for metadata references to variables are omitted in mimium as they are not used.
5. Instructions related to list-related data structures are also omitted in this paper, as the implementation of data structures such as tuples and arrays was omitted in the description of the λ_{mmm} in this paper.

Instructions in λ_{mmm} VM are 32bit data with operation tag and 3 operands. Currently, a bit width for the tag and each operands are all 8 bit[1].

The VM of λ_{mmm} is a register machine like the Lua VM (after version 5), although the VM has no real register but the register number simply means the offset index of the call stack from the base pointer at the point of execution of the VM. The first operand of most instructions is the register number in which to store the result of the operation.

The list of instructions is shown in Figure 5 (basic arithmetic operations are partly omitted). The notation for the instruction follows the Lua VM paper [7, p.13]. From left to right, the name of operation, a list of operands, and pseudo-code of the operation. When using each of the three operands as unsigned 8 bits, they are denoted as `A B C`. When used with a signed integer, prefix `s` is added, and when the two operand fields are used as one 16 bits, an suffix `x` is added. For example, when B and C are merged and treated as signed 16 bits, they are denoted as `sBx`.

In pseudo-code describing an functionality, `R(A)` means that data is moved in and out through the register (call stack) at the point of base pointer for current function + `A`. `K(A)` means that it retrieves the `A`-th number in the static variable field of the compiled program. `U(A)` means that referring `A`-th upvalue of the current function.

In addition to Lua's Upvalue operation, 4 operations related to internal state variables over time, `GETSTATE`, `SETSTATE`, `SHIFTSTATE` and `DELAY` are added to compile $delay$ and $feed$ expressions.

4.2. Overview of the VM structure

The overview of a data structure of the virtual machine, the program and the instantiated closure for λ_{mmm} is shown in Figure 6. In addition to the normal call stack, the VM has a storage area for managing internal state data for feedback and delay.

[1]Reason for this is that it is easy to implemented on `enum` data structure on Rust, a host language of the latest mimium compiler. Operands bitwidth and alignment may be changed in the future.

```
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 arithmetics continues for each primitive types...
```

Figure 5: *Instruction sets for VM to run λ_{mmm}.*

This storage area is accompanied by data indicating the position from which the internal state is retrieved by the GETSTATE / SETSTATE instructions. This position is modified by SHIFTSTATE operation back and forth. The actual data in the state storage memory are statically layed out at compile time by analyzing function calls that include references to self, call of delay and the functions which will call such statefull functions recursively. DELAY operation takes 2 inputs, B for an input and C for the delay time in samples.

However, in the case of higher-order functions that receive a function as an argument and return another function, the layout of the internal state of the given function is unknown at the compilation, so an internal state storage area is created for each instantiated closure separately from the global storage area held by the VM instance itself. The VM have an another stack to keep the pointer to state storage. Each time CALLCLS used, VM pushes a pointer to the state storage of instantiated closure to the state stack and, at the end of the closure call, VM pops out the state pointer from the stack.

Instantiated closures also hold the storage area of upvalues. Until the closure exits the context of parent function (such a closure is called "Open Closure"), upvalues holds a negative offset on the stack at the ongoing execution. This offset value can be determined at compile time, the offset is stored in the function prototype in the program. Also, not only local variables, upvalue may refer to parent funtion's upvalue (this situation can happens when at least 3 functions are nested). So the array of upvalue indexes in the function prototype holds a pair of tag whether it is local stack value or further upvalue and its index (negative offset of stack or parent function's upvalue index).

For instance, if the Upvalue indexes in the program were like [upvalue(1),local(3)], GETUPVALUE 6 1 means that, take 3 from the upvalue indexes 1 and get value from R(-3) over the base pointer and store it to R(6).

When the closure escapes from the original function with RETURN instruction, inserted CLOSE instruction
the RETURN instruction moves actual upvalues from the stack into somewhere on the heap memory. This upvalues may be referred from multiple locations when using nested closures, and some form of garbage collection needed to free memory after they are no longer referred.

In the current specification, the paradigm is call-by-value and reassignment expression does not exist, therefore, SETUPVALUE instruction does not exist in λ_{mmm} VM. This difference also make a difference to the implemention of open upvalue in the closure because the open upvalue should be shared memory cell which maybe recursively converted into memory cell of closed value when the CLOSE instruction is called.

4.3. Compilation to the VM instructions

Listing 2 shows an basic example when the mimium code in Listing 1 is compiled into VM bytecode. When self is referred, the value is obtained with the GETSTATE instruction, and the internal state is updated by storing the return value with the SETSTATE instruction before returning the value with RETURN from the function. Here, the actual return value is obtained by the second GETSTATE instruction in order to return the initial value of the internal state when time=0.

For example, when a time counter is written as | | {self + 1}, it is the compiler's design choice whether the return value of time=0 should be 0 or 1 though the latter does not strictly follow

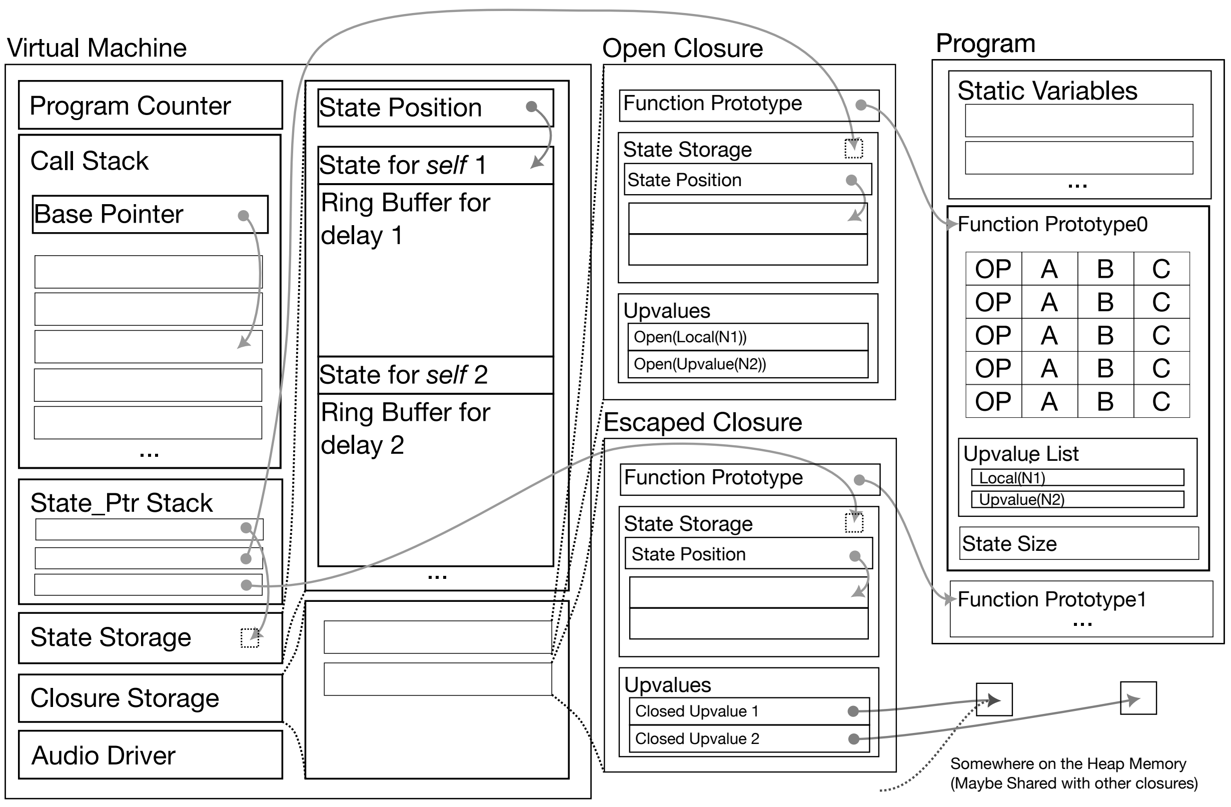

Figure 6: *Overview of the virtual machine, program and instantiated closures for λ_{mmm}.*

```
CONSTANTS:[1.0]
fn onepole(x,g) state_size:1
MOVECONST 2 0   // load 1.0
MOVE      3 1   // load g
SUBF      2 2 3 // 1.0 - g
MOVE      3 0   // load x
MULF      2 2 3 // x * (1.0-g)
GETSTATE  3     // load self
MOVE      4 1   // load g
MULF      3 3 4 // self * g
ADDF      2 2 3 // compute result
GETSTATE  3     // prepare return value
SETSTATE  2     // store to self
RETURN    3 1
```

Listing 2: *Compiled VM instructions of one-pole filter example in Listing 1*

the semantics E-FEED in Figure 4. If the design is to return 1 when time = 0, the second `GETSTATE` instruction can be removed and the value for the `RETURN` instruction should be `R(2)`.

A more complex example code and its expected bytecode instructions are shown in Listing 3 and Listing 4. The codes define delay with a feedback as `fbdelay`, the other function `twodelay` uses two feedback delay with different parameters, and `dsp` finally uses two `twodelay` function.

Each after the referring to `self` through `GETSTATE` instruction, or call to the other statefull function, `SHIFTSTATE` instruction inserted to move the position of state storage forward to prepare the next non-closure function call. Before exiting function, the state position is reset to the same position as that the current function context has begun by `SHIFTSTATE` (A sum of the operand for `SHIFTSTATE` in a function must be always 0). Figure 7 shows how the state position moves by `SHIFTSTATE` operations during the execution of `twodelay` function.

By describing an internal state as a relative position in the state storage, the state data can be expressed as a flat array, which makes the implementation of the compiler simple, not like a tree structure that need to analyze a call tree from the root to generate as in the previous implementation of mimium. This is similar to upvalue makes the implementation of the compiler simpler by describing free variables as relative positions on the call stack.

Listing 5 shows an example of a higher-order function `filterbank` that takes another function `filter` that takes an input and a frequency as an argument, duplicates `n` of filter, and adds them together. Note that in the previous specification of mim-

```
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)
}
```

Listing 3: *Example code that combines self and delay without closure call.*

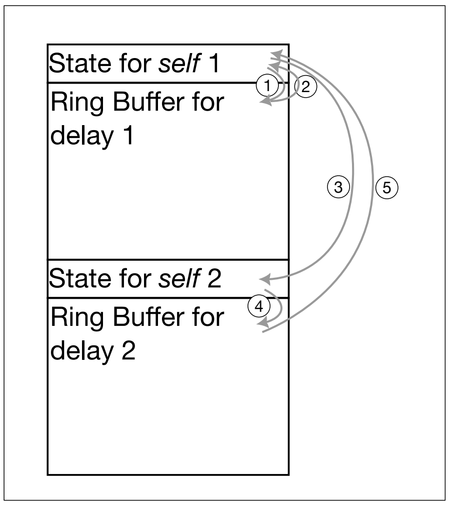

Figure 7: *Image of how the state position moves while executing* `twodelay` *function in Listing 4.*

ium in [6], the binding of new variable and destructive assignment were the same syntax (`x = a`) but the syntax for the variable binding has changed to use `let` keyword. Also, because the semantics is call-by-value paradigm, reassignment syntax never be used in the current implementation.

The previous mimium compiler could not compile code that takes a function containing such a state as an argument because the tree of all internal states was statically determined at compile time, but the VM in the λ_{mmm} can manage it dynamically. Listing 6 shows translated VM instruction of the code. Recursive calls of the first line of code in `filterbank` and the functions given as arguments or obtained via upvalue like `filter` are called with the `CALLCLS` instruction instead of the `CALL` instruction. The `GETSTATE` and `SETSTATE` instructions are not used in this function because the internal state storage is switched when the `CALLCLS` is interpreted.

5. DISCUSSION

As seen in the example of the filterbank, in λ_{mmm}, signal graph can be parametrically generated in an evaluation of global context, compared to that Faust uses a term-rewriting macro and Kronos uses a type-level computation as in the Table 1.

```
CONSTANTS:[0.7,2,0.8,400,800,0,1]
fn fbdelay(x,fb,dtime) state_size:2
MOVE        3 0   //load x
GETSTATE    4     //load self
SHIFTSTATE  1     //shift Spos
DELAY       4 4 2 //delay(_,_,_)
MOVE        5 1   // load fb
MULF        4 4 5 //delayed val *fb
ADDF        3 3 4 // x+
SHIFTSTATE -1     //reset SPos
GETSTATE    4     //prepare result
SETSTATE    3     //store to self
RETURN      4 1   //return previous self

fn twodelay(x,dtime) state_size:4
MOVECONST   2 5 //load "fbdelay" prototype
MOVE        3 0
MOVE        4 1
MOVECONST   5 0 //load 0.7
CALL        2 3 1
SHIFTSTATE 2    //2=state_size of fbdelay
MOVECONST   3 5 //load "fbdelay" prototype
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 -2
RETURN      3 1

fn dsp (x)
MOVECONST   1 6 //load "twodelay" prototype
MOVE        2 0
MOVECONST   3 3 //load 400
CALL        1 2 1
SHIFTSTATE 4    //4=state_size of twodelay
MOVECONST   2 6
MOVE        2 3 //load "twodelay" prototype
MOVE        3 0
MOVECONST   3 4 //load 400
CALL        2 2 1
ADD         1 1 2
SHIFTSTATE -4
RETURN      1 1
```

Listing 4: *Compiled VM instructions of feedback delay example in Listing 3*

```
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)
}
```

Listing 5: *Example code that duplicates filter parametrically using a recursive function and closure.*

The ability to describe both the generation of parametric signal processing and its content in a single semantics will make it easier for novice users to understand the mechanism of the language. Also, the single semantics may facilitate run-time interoperation with other general-purpose languages.

On the other hand, there is the problem that the single semantics causes λ_{mmm} to behave differently from the behavior expected in a normal lambda calculus.

	Parametric Signal Graph	Actual DSP
Faust	Term Rewriting Macro	BDA
Kronos	Type-level Computation	Value Evaluation
λ_{mmm}	Evaluation in Global Context	Evaluation of `dsp` Function

Table 1: *Comparison of the way of signal graph generation and actual signal processing between Faust, Kronos and λ_{mmm}.*

5.1. Different behaviour depending on the location of let binding

By having functions that have internal states which change over time in mimium, when higher-order functions are used, there is a counterintuitive behavior compared to general functional programming languages.

Listing 7 is an example of the incorrect code slightly modified from the filterbank example in Listing 5. The difference between Listing 5 and Listing 7 is that the recursive calls in the filterbank function are written directly, or once bound with `let` expression out of the inner function. Similarly, in the `dsp` function that will be called by the audio driver in mimium, the difference is whether the filterbank function is executed inside `dsp` or bound with `let` once in the global context.

In the case of normal functional language, if all the functions used in a composition do not contain destructive assignments, the calculation process will not change even if the variable bound by `let` were manually replaced with its term (beta reduction), as in the conversion from Listing 5 to Listing 7.

But in mimium, there are two major stages of evaluation, 0: the code is evaluated in the global environment (concretizing the signal processing graph) at first, and 1: the dsp function is repeatedly executed (actual signal processing) and the function may in-

```
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)
MOVECONST 2 1 //load itself
MOVE      3 0 //load n
MOVECONST 4 1 //load 1
SUBF      3 3 4
MOVECONST 4 2 //load inner_then
CALLCLS   2 2 1 //recursive call
MOVE      3 0
MOVECONST 4 2 //load 0
SUBF      3 3 4
JMPIFNEG  3 2
MOVECONST 3 2 //load inner_then
CLOSURE   3 3 //load inner_lambda
JMP       2
MOVECONST 3 3 //load inner_else
CLOSURE   3 3
CLOSE     2
RETURN    3 1
```

Listing 6: *Compiled VM instructions filterbank example in Listing 5*

```
fn bandpass(x,freq){
  //...
}
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(){ //called by audio driver.
  filterbank(3,bandpass)
}
```

Listing 7: *Wrong example of the code that duplicate filter parametrically.*

```
  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)
  }
```

Listing 8: *Example of filterbank function using multi-stage computation in a future specification of mimium.*

volve implicit internal state updates. Therefore, even though the code does not include destructive assignments, the recursive execution of the `filterbank` function is performed only once in Listing 5 for the evaluation of the global environment, whereas in Listing 7, every sample the dsp function is executed, the recursive function is executed and a closure is generated. Since the initialization of the internal state in the closure is performed at the time of closure allocation, in the example of Listing7, the internal state of the closure after the evaluation of `filterbank` is reset at each time step.

This means that the major compiler optimization techniques such as the constant folding and the function inlining can not simply be appropriated for mimium. Those optimizations should be done after the evaluation of a global context and before evaluating `dsp` function.

To solve this situation, introducing distinction whether the term should be used in global context evaluation (stage 0) and in the actual signal processing (stage 1) in type system. This can be realized with Multi-Stage Computation[9]. Listing 8 is the example of `filterbank` code using BER MetaOCaml's syntaxes `.<term>.` which will generate evaluated program to be used in a next stage, and `~term` which embed terms evaluated at the previous stage[10].

`filterbank` function is evaluated in stage 0 while embedding itself by using ~. This multi-stage computation code still has a same semantics in a generative signal graph generation and execution of the signal processing, in contrast to that Faust and Kronos.

5.2. A possibility of the foreign statefull function call

The data structure of closure in λ_{mmm} is a combination of functions and internal states, as shown in Figure 3. The fact that `filterbank` samples do not require any special handling of internal states also means that external signal processor (Unit Generator: UGen) such as oscillators and filters written in C or C++, for example, can be called from mimium in the same way as normal closure calls, and it is even possible to parametrically duplicate and combine external UGens. This is an advantage that is difficult to implement in Faust and other similar languages, but easy to implement on λ_{mmm} paradigm.

However currently, mimium is based on sample-by-sample processing and cannot handle buffer-by-buffer value passing. Since most native unit generators perform processing on a buffer-by-buffer basis, there are not many cases where external UGens are utilized in practice for now. However, in the λ_{mmm}, only $feed$ terms need to be processed sample-by-sample, so it is possible to distinguish functions that can only process one sample at a time from functions that can process concurrently at the type level. As the Multi-rate specification is being considered in Faust[11], it may be possible to read/write buffer between an external Unit Generator by having the compiler automatically determine the parts that can be processed as buffer-by-buffer.

6. CONCLUSION

This paper proposed λ_{mmm}, an intermediate representation for the programming languages for music and signal processing with the virtual machine and instruction set to run it. λ_{mmm} enables to describe generative signal graph and its contents in a single syntax and semantics. However, user have to be responsible to write codes that does not create escapable closures during the iterative execution of DSP, which will be difficult to understand for novice users.

In this paper, the translation from λ_{mmm} terms from VM instructions is explained by just showing examples of the code and its expected result of instructions as well as the semantics of VM is presented with pseudo-code of the behaviour. More formal semantics and translation process should be considered along with an introduction of the multi-stage computation.

I hope that this research will lead to more general representations of music and sound on the digital computer and more connections between the theory of languages for music and more general programming language theory.

7. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI (Grant No. JP19K21615). Also great thanks for many anonymous reviewers.

8. REFERENCES

[1] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[2] Albert Gräf, "Term Rewriting Extension for the Faust Programming Language," in *International Linux Audio Conference*, 2010.

[3] Benedict R Gaster, Nathan Renney, and Tom Mitchell, "OUTSIDE THE BLOCK SYNDICATE: TRANSLATING FAUST'S ALGEBRA OF BLOCKS TO THE ARROWS FRAMEWORK," in *Proceedings of the 1st International Faust Conference*, 2018.

[4] Vesa Norilo, "Kronos: A Declarative Metaprogramming Language for Digital Signal Processing," *Computer Music Journal*, vol. 39, no. 4, pp. 30–48, 2015.

[5] Emilio Jesús Gallego Arias, Pierre Jouvelot, Sylvain Ribstein, and Dorian Desblancs, "The W-calculus: A Synchronous Framework for the Verified Modelling of Digital Signal Processing Algorithms," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*. 2021, vol. 12, pp. 35–46, Association for Computing Machinery.

[6] Tomoya Matsuura and Kazuhiro Jo, "Mimium: A self-extensible programming language for sound and music," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, New York, NY, USA, Aug. 2021, FARM 2021, pp. 1–12, Association for Computing Machinery.

[7] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, "The Implementation of Lua 5.0," *JUCS - Journal of Universal Computer Science*, vol. 11, no. 7, pp. 1159–1176, July 2005.

[8] Robert Nystrom, *Crafting Interpreters*, Genever Benning, Daryaganj Delhi, July 2021.

[9] Walid Taha and Tim Sheard, "Multi-Stage Programming with Explicit Annotations," *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, vol. 32, no. 12, pp. 203–214, Dec. 1997.

[10] Oleg Kiselyov, "The Design and Implementation of BER MetaOCaml," in *Proceedings of the 12th International Symposium on Functional and Logic Programming*, Michael Codish and Eijiro Sumii, Eds., Cham, 2014, pp. 86–102, Springer International Publishing.

[11] Pierre Jouvelot and Yann Orlarey, "Dependent vector types for data structuring in multirate Faust," *Computer Languages, Systems & Structures*, vol. 37, no. 3, pp. 113–131, 2011.