mimium: a self-extensible programming language for sound and music

Tomoya Matsuura
me@matsuuratomoya.com
Graduate School of Design, Kyushu University
Fukuoka, Fukuoka, Japan

Kazuhiro Jo
jp@jp.org
Faculty of Design, Kyushu University
Fukuoka, Fukuoka, Japan

Abstract

We propose a programming language for music named *mimium*, which combines temporal-discrete control and signal processing in a single language. *mimium* has an intuitive imperative syntax and can use stateful functions as Unit Generator in the same way as ordinary function definitions and applications. Furthermore, the runtime performance is made equivalent to that of lower-level languages by compiling the code through the LLVM compiler infrastructure. By using the strategy of adding a minimum number of features for sound to the design and implementation of a general-purpose functional language, *mimium* is expected to lower the learning cost for users, simplify the implementation of compilers, and increase the self-extensibility of the language. In this paper, we present the basic language specification, semantics for simple task scheduling, the semantics for stateful functions, and the compilation process.

mimium has certain specifications that have not been achieved in existing languages. Future works suggested include extending the compiler functionality to combine task scheduling with the functional paradigm and introducing multi-stage computation for parametric replication of stateful functions.

CCS Concepts: • **Applied computing** → **Sound and music computing**; • **Computer systems organization** → *Real-time languages*; • **Theory of computation** → Timed and hybrid models.

Keywords: programming language, computer music, functional programming, signal processing

1 Introduction

1.1 Background

Programming environments for computer music, which are *software packages that enable the use of a digital computer to create music programmatically*[8], have been continuously developed since the early days of computers, such as Max[18], Puredata[19] and SuperCollider[11].

Among these programming environments, conventional languages contain multiple layers internally for discrete event control and signal processing by composing Unit Generator(UGen): a fundamental component of signal processing and description of UGen[8].

In conventional environments, there is a limitation in terms of extensibility in the lower-level description of signal processing. The user can describe signal processing by combining many built-in UGens, such as filter or oscillator, provided in the language. If the user wants to define a new UGen that cannot be expressed by combining existing UGens (for example, a nonlinear oscillator), they must use general-purpose languages such as C, to describe the algorithm in such environments.

For this type of problems, languages focusing on describing UGens, such as Faust[17], Kronos[14], Soul[27], Vult[21], and Gen˜(an embedded language on Max) were developed. For example, Faust can be used to output UGen binaries for Max, Puredata, and SuperCollider via C++ code or can be used as an original UGen in Max by compiling Faust code on memory using LLVM. These languages do not have a scheduler for high-level event control. In languages such as Faust, such discrete values are defined as external values controlled outside the program (for example, via GUI, MIDI, OSC).

The advantage of using multiple languages is that it maintains a balance between efficiency (in terms of coding by user) and generality of possible expressions. In addition, it allows the user to choose the level of complexity according to the task[8].

Multi-language paradigms, however, actually lead to other problems. For example, sometimes, the user must use slightly different operators for similar expressions, which may reduce the efficiency of the programming process for beginners. For instance, while describing the addition of two inputs in Max and Puredata, the user must choose the right one from two different objects according to its data type between [+] for control and [+ ~] for audio. SuperCollider also requires the user to select multiple methods for the same *SinOsc* object that generates a sine wave, such as *SinOsc.kr* when the required time-domain resolution is slow (e.g., LFO) and *SinOsc.ar* when it is used as an audio signal, depending on the processing load required.

In addition, practically, while a multi-language paradigm can strike a balance between generality of expression and efficiency of programming, the learning cost is high as users must learn separate languages for each domain. Considering that domain-specific-languages (DSLs)

have high training costs [26], if the languages can be unified without losing generality and efficiency, the training costs can be reduced.

Improving self-extensibility is one of the important topics in the design of programming languages for sound and music. Dannenberg argues that introducing ready-made solutions to a language specification will ultimately limit the expressiveness of the language itself. The language should therefore increase its expressiveness, and it is better to develop individual solutions as libraries on the language[2].

In fact, ChucK[30], Extempore[25] and Kronos[14] partially addressed the two aforementioned problems.

ChucK allows users to define their own UGen in the ChucK language itself using a language extension ChuGen[22]; however, as ChucK itself is a virtual machine-based interpreter language implemented on C, its runtime performance is inferior to that of UGen written in C++ for the same processing method. Furthermore, the data type of the input/output for UGen is distinct from a general numeric type, and the user must use the ChucK operator (=>) to represent connections between UGens.

In the Lisp-based live-coding environment Extempore, users can compile native binaries during runtime on a dedicated language called xtlang through the compiler-infrastructure LLVM[7], and the entire code including signal processing equivalent to UGen can be written within the Extempore environment while maintaining high runtime performance. It is, however, necessary to use two different languages: a dynamically typed language(Scheme) for control processing and a statically typed language(xtlang) for signal processing.

Kronos Meta-Sequencer[15], an extended specification of Kronos that was developed to unify Score, Orchestra(Composition of UGens) and Instrument(Description of UGen) languages through the preparation of syntactic sugars that combine the design pattern of Temporal Recursion[24] and IO Monads; however, Kronos can also be seen as a two-layered design of a dynamically typed meta-language that generates statically typed program[16, p34].

1.2 Introducing *mimium*

Granted the above background, we introduce *mimium* (***mi**nimal-**m**usical-med**ium***)[1], a full-stack music programming language, which can describe everything from low-level signal processing to discrete event processing in unified semantics.

Table 1 shows a comparison of the language specifications of *mimium* and existing languages. *mimium* realizes discrete-time event description and signal processing in unified semantics and achieves high execution speed via JIT compilation equivalent to UGen written in lower-level languages such as C++. The user does not need to be aware of hardware management such as memory allocation and release, which are determined statically during compilation.

In the following section, we describe the detailed language design and implementation of the running environment of *mimium*.

First, we introduce the basic syntax, showing that there awareness of hardware such as memory management is not required, and that type inference allows users to omit type annotations for variables. Next, we present the general architecture of *mimium*'s running environment (compiler and runtime), showing that *mimium* code can be immediately compiled into native binaries through LLVM and executed without losing run time performance, even for signal processing.

In addition, we describe two characteristic features of *mimium* that allow describing continuous signal processing and discrete control processing in unified semantics. The first is the syntax for a deterministic task scheduling at the sample level and the implementation of the scheduler. The second is a description of the semantics used to define the UGen for signal processing on the language and its compilation process, comparing it to the existing paradigm in terms of the data structure of a pair of functions and internal state variables.

In the discussion section, we address two problems: (1) although *mimium* can describe discrete control and signal processing in unified semantics, the way it describes discrete processing is more likely to be imperative, and the functional paradigms used for signal processing are very different from one another, and (2) the current implementation cannot express a parametric replication of stateful functions for signal processing unlike Faust and Kronos. We will also explain the possibility of using multi-stage computation as a solution to the aforementioned problems.

2 Design of *mimium*

2.1 Basic syntax and semantics

The basic syntax of *mimium* is based on the Rust language[6], which can be written like a general imperative programming language.

The main reason for this is that the syntax of the Rust is similar to conventional languages, and the relatively short reserved words are suitable for fields that perform quick prototyping like music. It also has the side effect of being close to the existing syntax of the language, which makes it easy to reuse the syntax highlighting from existing languages.

[1] https://github.com/mimium-org/mimium

Table 1. Comparison of specifications between computer music languages.

	Pd/SC	ChucK	Extempore	Faust & Vult	Kronos	mimium
Event Scheduler and Semantics for it	◯	◯	◯	-	◯	◯
Sample-Accurate Scheduling	-	◯	◯	-	◯	◯
Fundamental UGen Definition	-	◯	◯	◯	◯	◯
JIT Compilation of DSP Code	-	-	◯	◯	◯	◯
Functional Representation of Internal State	-	-	-	◯	◯	◯

```
1  //double slash for comments.
2  //assignment to the variable is also a
       declaration of new variable
3  mynumber = 1000
4  // Currently, all variables are mutable
5  mynumber = 2000
6
7  //type specification is optional
8  myvariable:float = 10
9  type FilterCoeffs = (float,float,float,float,
       float) //type alias definition
10
11 mystring = "somefile.wav" // string literal
       values are used for file loading and
       debugging purposes.
12
13 //array type value constructor
14 myarr = [1,2,3,4,5,6,7,8,9,10]
15 //access to the array. arr_content should be 1
16 arr_content = myarr[0]
17 myarr[4] = 20 //assignment to array. myarr
       becomes [1,2,3,4,20,6,7,8,9,10].
18
19 mytup = (1,2,3) //tuple type constructor
20 one,two,three = mytup // unpacking tuple.
21
22 // basic function definition.
23 fn add(x,y){
24   return x+y
25 }
26
27 add = |x,y|{ x+y }//equivalent definition of
       function with inline version.
28
29 //block,brace-wrapped multiple statements is an
        expression globally.
30 z = { x = 1
31       y = 2
32       return x+y } //z should be 3.
33
34 //conditional and recursive function
35 fn fact(input){
36   if(input>0){
37     return 1
38   }else{
39     return input * fact(input-1)
40   }
41 }
42 // if-else statement can be used as expression
       like below
43 fact = |input|{ if(input>0) 1 else input * fact
       (input-1) }
```

Listing 1. Basic syantax of *mimium*.

Listing 1 shows the list of the basic syntax. A formal definition of the language is written in Appendix A. Declaring a variable is done automatically by assigning some value to a variable with a new name withiin a scope of function. When declaring a variable, value type can be explicitly specified by providing the type name after a colon. When the type name is omitted, it can be inferred from context. Data types include void (an empty-value type), numeric (no distinction between integer/decimal type and internally a 64-bit float by default), and string as primitive types as well as aggregate types such as function, tuple, and array types. User-defined type aliases can also be declared.

The basic syntax includes function definitions, function calls, and conditional using if-else statements. *mimium* also incorporates the functional paradigm, allowing if statements to be used as expressions that can return values directly. This is achieved by having a syntax that allows multiple statements (assignment syntax or function execution) enclosed in a {} to be used as an expression that provides the value of the *return* expression of the last line (*return* can also be omitted). Similarly, function definitions are defined as syntax sugars for the assignment syntax of anonymous functions.

mimium is a statically-typed language, which means that the types of all variables and functions are determined during compilation. Type inference is based on Hindley-Milner inference systems (currently monomorphic).

In addition, for faster DSP processing, memory allocation and deallocation are determined statically at compile-time, and the runtime has no garbage collection.

2.2 Basic DSP in *mimium*

In *mimium*, when the user defines a function named *dsp*, it becomes an entry point to exchange audio input and output with an audio driver. The example is in Listing

2. In this case, the type of the *dsp* function must be a function type that takes *a tuple of any number of floats* and also returns *a tuple of any number of floats*. Each element of the tuple corresponds to input & output channels of the audio driver. The example of Listing 2 is a code that receives two channels of input from the audio driver, mixes them, and returns duplicated signals for the left and right channels.

```
fn dsp(input:(float,float))->(float,float){
  left,right = input
  out = (left+right)/2
  return (out,out)
}
```

Listing 2. Example of *dsp* function which merges stereo inputs and returns the same signal to each output channels.

The built-in functions in *mimium* include basic arithmetic operations, mathematical functions such as trigonometric and exponential functions defined in libc's *math.h*, built-in stateful functions such as *delay* and *mem* (one-sample delay), *loadwav* function for loading wav files using libsndfile[9], and *print* function for debugging. The filters and oscillators can all be defined as libraries by combining these functions.

2.3 Architecture

Figure 1 shows the architecture of a compiler and runtime of *mimium*.

The structure of the compiler is similar to that of a general functional language, based on the implementation in mincaml [28] and implemented on C++.

Text data of source codes is first parsed into an abstract syntax tree, and after removing the syntax sugar, the AST becomes transformed into a lambda calculus-based tree structure. Then, type inference and type checking are performed to determine all variable types. The AST is converted with the type information into a single-static-assignment form imperative intermediate representation where all variables are assigned only once. Considering that nested function definitions are still allowed at this stage, a closure conversion is performed to remove free variables from the function definition.

State variable detection for *mimium*'s unique specification of stateful function (described in Section 3.2) is performed between the closure transformation and the lower-level code (LLVM IR) generation. The transformer outputs the state variables used by the function as data in a tree structure (*State Tree* in the figure) with the node of the called stateful function names and the type of the state variables of the function, taking the *dsp* function as the entry point of the signal processing. Finally, the LLVM IR is generated based on the closure transformed IR and the State Tree.

The runtime consists of three parts: the execution engine, which receives the LLVM IR and compiles it into a native binary in memory; the audio driver, which handles input/output communication with the audio device; and the scheduler, which keeps information about the function and the logical time of the specified execution time. The audio driver currently uses RtAudio[23], a cross-platform library for C++ that abstracts audio devices through the operating system's API. The execution engine passes the *dsp* function, which is the entry point for signal processing, to the audio driver. The audio driver, in turn, commands the scheduler to advance the logical time. The scheduler is responsible for executing tasks as well as responding to requests from the execution engine to register tasks and obtain the internal time.

Only two functions of the LLVM IR compiled in *mimium* depend on the runtime system. One for registering tasks and another for getting the internal time. Almost[2] all other code is compiled on memory and executed; therefore, it can have the same execution speed as processing written in low-level languages such as C.

3 Characteristic semantics in *mimium*

3.1 Scheduling with @ operator

```
ntrigger = 1
fn setN(val:float){
    ntrigger = val
}
fn playN(duration:float)->void{
  setN(1)
  setN(0)@(now+duration)
}
fn Nloop(period:float)->void{
     playN(50)
     nextperiod = if(random()>0) period/2 else
    period
     Nloop(period)@(now+nextperiod)
}
Nloop(12000)
```

Listing 3. Example of Temporal Recursion

To describe events that occur discretely in the temporal direction in *mimium*, we used a design pattern called temporal recursion, which was introduced in Impromptu[24](a prior work of Extempore) and used in several languages such as Overtone[1] and Kronos Meta-Sequener[15]. The design pattern describes repetitive event processing as a function that calls itself recursively with a time delay.

A concrete example is shown in Listing 3. When a numeric value is given after the @ operator following the function call, the function is not executed immediately.

[2]Actually, some built-in functions, such as print, call pre-compiled C libraries to simplify the implementation.

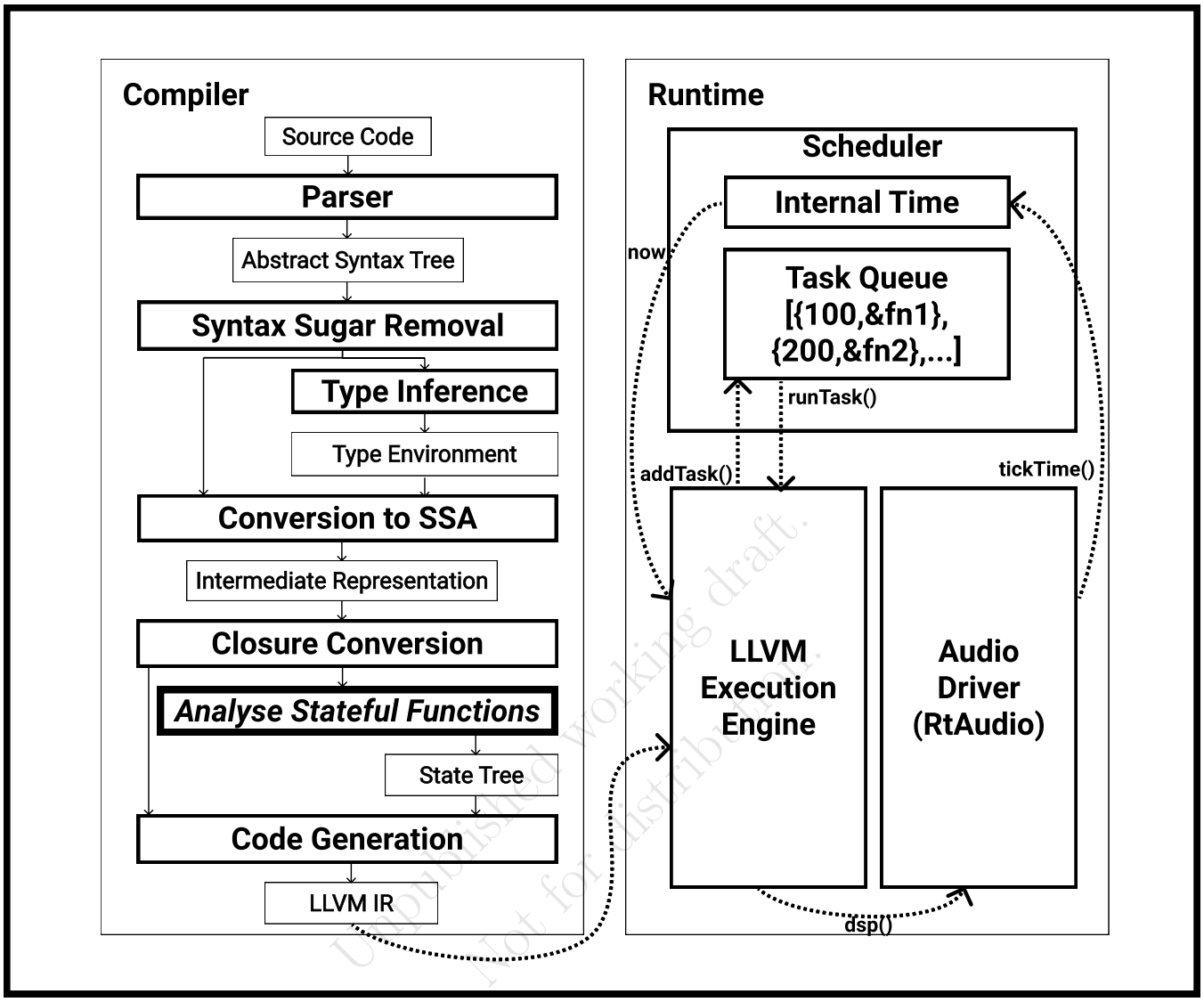

Figure 1. Architecture of *mimium* compiler and runtime.

Instead, it is registered to a task queue with a priority keyed by time, and the execution context returns to the next statement. The runtime checks the task queue before processing each sample demanded on the audio driver clock, and if the key of the first task has reached the current time, it executes them first before processing the audio signal. The time is the absolute time as the runtime started executing each sample. The user can describe relative time using the keyword *now* to obtain the current logical time from the runtime as same way as prior works.

```
(define ntrigger 1)
(define setN
    (lambda (val)
        (!set ntrigger 1)))
```

```
(define playN
    (lambda (duration)
            (setN 1)
            (callback (+ now duration) 'setN 0)))
(define Nloop
  (lambda (period)
        (playN 50)
        (callback (+ (now) (if (random > 0) (/ period 2) period)) 'Nloop period)))
(Nloop 12000)
```

Listing 4. Equivalent code to Listing 3 in Extempore

The function can be executed at regular intervals by calling itself recursively with a time specification within a function. In the case of Listing 3, the variable *ntrigger*

is rewritten every time the function *Nloop* is called. Listing 4 is an equivalent code in Extempore to Listing 3. Extempore uses a special function called *callback* for temporal recursion, while *mimium* introduces a special operator @ to improve readability.

mimium uses synchronous scheduling based on logical time, similar to ChucK for the simplicity of implementation. The logical time based scheduling will result in inaccurate for processes that involve IO exchanges such as sending/receiving MIDI and OSC even once; however, if the process is closed only in the language, accurate processing can be guaranteed on a sample-by-sample basis; the implementation is simple, and the execution cost is relatively low.

In contrast, Extempore uses asynchronous scheduling by dividing the event scheduling threads. In the case of asynchronous scheduling, processes involving IO can be processed accurately in real-time; however, as the implementation depends on the OS task scheduler, it requires individual support for each OS, and the execution cost is relatively high.

3.2 Stateful function for signal processing

In this section, we describe appropriate semantics and data structures for expressing UGen.

3.2.1 Comparing semantics for UGen between data structures.

A UGen takes a series of input data, processes it in some way, and outputs it. At the first glance, this seems to be possible as a pure function, but in reality, we must use a data structure of a function and a set of variables.

For example, a pure function is sufficient if it only adds or multiplies the inputs; however, to represent signals that cannot be expressed as a map $f(t)$ to time t, such as some filters and nonlinear oscillators, UGen must have an internal state. Therefore, to represent UGen in a general-purpose language, it must be represented as a data structure that combines functions such as objects and closures with internal states; however, if the user wants to use multiple objects or closures, they must instantiate them once and then call the actual process.

mimium has semantics, which allow us to use UGen as if it is a normal function, without having to create a dedicated data type for signal processing. Further, we will see how to represent phasor, which is a sawtooth-wave-like UGen that increases from 0 to 1 at a constant rate and returns to 0 again, in objects, closures, Faust, and Vult, and then we present a semantics for UGen as a stateful function in *mimium* and its compilation.

Object. Object is a data structure that contains a set of member variables as well as a set of member functions (methods) that modify the variables and send messages to other objects. In the case of an object, the internal state is defined as a member constant. To use it, the user must instantiate it beforehand and then call the main processing method. Listing 5 shows the pseudo-code in C++.

```
class Phasor{
  double out_tmp=0;
  double process(double freq){
    out_tmp = out_tmp+freq/48000;
    if(out_tmp>1){
      out_tmp = 0;
    }
    return out_tmp;
  }
};
//Instantiation
Phasor phasor1;
Phasor phasor2;
double something(){
  //use instantiated objects
  return phasor1.process(phasor2.process(10) +
    1000);
}
```

Listing 5. The code of Phasor written with object in C++.

Closure. Closure is a feature available in languages with a lexical scope that allows function definitions within functions. For example, the user can define function A that defines multiple local variables $a, b, c \ldots$ and returns another function B that refers to the variables a, b, c as free variables. Function A is a higher-order function that returns function B, and executing A is equivalent to creating an instance in an object.

The problem with using closures to describe signal processing is that it becomes difficult to determine the lifetime of variables at compile time. Languages that can use closures are often implemented with a garbage collection for automatic memory allocation and release, but it is difficult to bring GC to DSP languages[3] where functions are executed 48000 times per second in real-time. SuperCollider implements a GC that can work on real-time systems, and Extempore solves this problem by requiring the user to specify the lifetime with manual memory management. It means that either the user or the developer must bear the implementation cost.

The following example(Listing 6) is a pseudo-code in JavaScript [3].

```
//pseudo-code in javascript
function makeFilter(tmpinit){
  let out_tmp = tmpinit;
  let process = (freq) => {
```

[3]It is difficult to use this for signal processing practically as JS works with GC, but we used JS to show an example because it is imperative, easy to read, and closure can be used.

```
    out_tmp = out_tmp+freq/48000;//referring free variable out_tmp
    if(out_tmp>1){
      out_tmp = 0;
    }
    return out_tmp;
  }
  return process;
}
//instantiation
let phasor1 = makePhasor(0);
let phasor2 = makePhasor(1);
function something(){
  return phasor1(phasor2(10) + 1000);
}
```

Listing 6. The pseudo-code of Phasor written with Closure in Javascript.

Functional Representation. In the description of signal processing in Faust and Kronos, a minimum set of functions with internal states represented by delays and a one-sample delay implicit in recursive connections are prepared as built-in functions to enable algebraic combinatorial expressions in UGen without reading and writing temporary variables.

As shown in Listing 7, unlike objects and closures, there is no need to instantiate them first, and temporary variables for the phasor are automatically allocated for each function call after compilation.

What is symbolized in these languages is, in Faust, a unit generator with input and output (a constant is a function with no input and one output, + operator is a function with two inputs and one output, and so on), and an input/output list of a processor in Kronos. In these languages, the symbols do not correspond to data on a specific memory address as in ordinary languages. For this reason, it is difficult to use these languages as self-extensible systems.

```
phasor(freq) = +(freq/4800) ~ out_tmp
  with{
        out_tmp = _ <: select2(>(1),_,0);
 };
// no need to instantiate.
something = phasor(phasor(10)+1000);
```

Listing 7. The code of Phasor written with Faust.

In the Vult language[21], if the user declares a variable with the keyword *mem* and not the usual variable declaration *var* in a function definition, the destructively changed value will be kept over time series so that it can represent the internal states of the UGens. This feature allows the user to represent the connection of a UGen with an internal state as if it were a normal function application and does not need to be instantiated in advance as in Faust.

```
fun phasor(freq){
  mem out_tmp;//"mem" variable holds its value over times
  out_tmp = out_tmp+freq/48000;
  if(out_tmp>1){
    out_tmp = 0;
  }
  return out_tmp;
}
fun something(input){
// no need to instantiate.
  return phasor(phasor(10)+1000);
}
```

Listing 8. The code of Phasor written with Vult.

In both Faust and Vult, functions with an internal state can be expressed directly without first instantiating them. Instead, the initialization of the internal state is determined at the time of function definition, and the initial value cannot be determined via a constructor when creating an instance.

In other words, by taking advantage of the fact that the initial value of the internal state is almost always zero or an array of zeroes, which constitutes domain-specific knowledge in signal processing, functions with internal states can be expressed in the same syntax as normal function definition and application.

By expressing all the stateful functions with a limited number of built-in stateful functions (delays, table lookups) and feedback connections as in Faust, stateful functions can be mixed with the normal function application grammar, eliminating the need to create an instance of the function once; thereby removing redundancy from the code.

3.2.2 DSP coding in *mimium*. Based on these assumptions, in *mimium*, stateful functions can be used as UGens. They are called in the same semantics of normal function application $f(x)$ as in Vult. Using a limited number of built-in stateful functions such as delay as in Faust, the user can write stateful functions with little awareness of variable management, while maintaining that what is symbolized as data on the memory, similar to the case of general-purpose languages.

In addition, the user can use the keyword *self* in the function definition to refer to the return value returned by the function in the previous sample.

self is a reserved word that can only be used in function definitions. *self* is initialized with 0 and allows us to get the previous return value of the function. Listing 9 is the simplest use of *self*, a function that increments from 0 to 1 per sample.

```
fn counter(){
    return self+1
```

```
}
```

Listing 9. The code of sample counter in *mimium*.

By applying this method, we can define the UGen phasor, which we have seen as examples in objects and closures and as functions, as shown in Listing 10. In this example, the user does not need to declare variables in the function, and there is no need to instantiate when using the function. Additionally, the use of a recursive connection is closed within the unit of the function, unlike the representation of recursive connections as the infix operator ∼ in Faust.

```
fn phasor(freq){
  res = self + freq/48000 // assuming the
    sample rate is 48000Hz
  return if (res > 1) 0 else res
}
fn dsp(input){
    return phasor(440)+phasor(660)
}
```

Listing 10. The code of phasor in *mimium*.

Further, for users who are already familiar with the dataflow and functional paradigms, *mimium* provides the pipeline operator $|>$ as a syntax that makes it easier to interpret stateful functions as connections between processors. The pipeline operator is used in several functional language specifications such as F#[13] and allows programmers to rewrite a nested function call $h(g(f(arg)))$ as $arg|>f|>g|>h$. Listing 11 is an example of defining a sine wave oscillator using both regular function calls and pipeline operators.

The equivalent codes to Listing 11 are shown in Listing 12 in Faust and in Figure 2 in Max, which describes the flow of data from left to right in the same way as the graphical connection of UGen. In addition to the sequential composition operator in Faust(:), there are operators with similar functions in other languages, such as ChucK operator (=>) in ChucK language, but the difference is that *mimium*'s pipeline operators are semantically equivalent to function calls.

```
fn scaleTwopi(input){
  return input* 2 * 3.141595
}
fn osc(freq){ //normal function call
  return cos(scaleTwipi(phasor(freq)))
}
fn osc(freq){ //pipeline operator version
  return freq |> phasor |> scaleTwopi |> cos
}
```

Listing 11. Example of Pipeleine Operator in *mimium*

```
  scaleTwopi(input) = input * 2 * 3.141595;
  osc(freq) = freq : phasor : scaleTwopi : cos;
```

Listing 12. Example of Sequential Composition in Faust

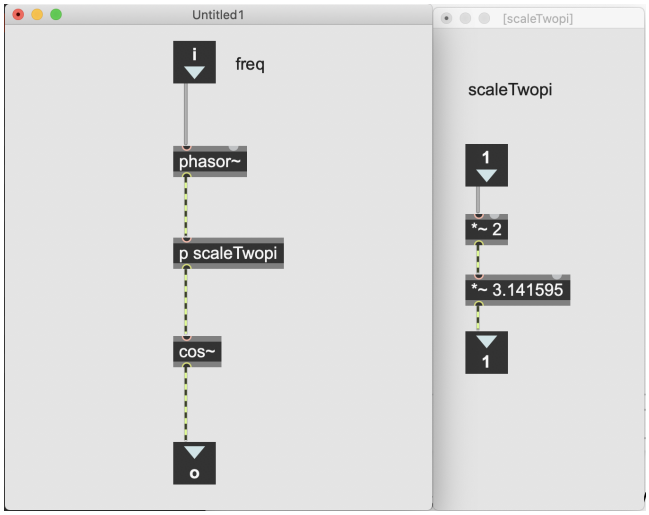

Figure 2. Example of dataflow syntax in Max

3.2.3 Compilation process of stateful functions.

Stateful functions are transformed into a combination of pure functions and state variables as arguments after closure transformation during compilation and before low-level code generation, as shown in the Figure 1.

The transformation is done as follows. First, all the function calls contained in *dsp* function definition are searched in order, and if the function is defined in *mimium*, the compiler further looks up its definition recursively to create a dependency tree of function calls.

Finally, if the function definition refers to *self* or the call of a built-in stateful function such as *mem* or *delay*, then the function becomes a stateful one, and the function that calls the stateful function is also determined as stateful.

After creating the tree, the function definition is rewritten such that the argument of the stateful function is a pointer to a tuple-type variable that lists all the state variables used in the function. The function call part should be rewritten in the same way, that is, to ensure that the state variables become explicit arguments.

As an example, the code that uses the built-in delay function and the two types of self is shown in Listing 13.

```
// delay is a built-in stateful function
fn fbdelay(input,time,fb){
    return delay(input+self*fb,time)
}
fn dsp(){
    // mix 2 feedback delay with different
    parameters
    src = random()*0.1
    out = fbdelay(src,1000,0.8)+fbdelay(src
    ,2000,0.5)
    return (out,out)
```

```
}
```

Listing 13. Example of Feedback Delay in *mimium* before state tree transformation.

The pseudo-code converted from this code to a form in which state variables are explicitly given as arguments is shown in 14.

```
// pseudo-code after lifting stateful function
fn fbdelay(state,input,time,fb){
  //unpack state variables
  self,delay_mem = state
  return delay(delay_mem,input+self*fb,time)
}
fn dsp(state){
    // unpack state variables
    s_fbdelay0, s_fbdelay1 = state
    src = random()*0.1
    out = fbdelay(s_fbdelay0,src,1000,0.8)+
    fbdelay(s_fbdelay1,src,2000,0.5)
    return (out,out)
}
```

Listing 14. Pseudo-code of Feedback Delay in *mimium* after state tree transformation.

4 Discussion

To summarize, *mimium* can describe temporal-discrete control and signal processing in unified semantics, including the definition of UGen as a stateful function, and the user can write code without being aware of the hardware. In addition, almost all of the code is compiled on memory through LLVM, so that the execution speed is equivalent to that of a low-level language. For writing discrete processing, the @ operator can be used to specify the time to execute a function, and by combining it with the temporal recursion design pattern, it is possible to abstract events that occur repeatedly in the time domain. For the description of signal processing, by hiding state variables and combining only feedback connections and limited built-in functions with states as in Faust, functions with internal states can be expressed in the same syntax as normal function definitions and applications.

4.1 Comparison to related works

Compared to the existing environment, *mimium* brings the following advantages: By taking an architecture that **adds minimal musical features and semantics to the specification and implementation of a general-purpose programming language, it keeps the implementation simple and allows the user to focus on musical tasks without losing the self-extensibility of the programming language**.

In fact, *mimium* can be used like a general-purpose scripting language when the source does not use scheduling or stateful functions. The compiler structure of *mimium* is the same as that of a general functional language except for the stateful function conversion part.

Extempore is similar to this approach in this aspect, allowing all description in a single environment; however, user must use two different language: Scheme and xtlang. xtlang requires the user to understand manual memory management and complex type signatures including pointers when defining UGen as a closure. Although a manual memory management is not always a negative point as Extempore is an environment for full-stack live programming that is not limited to music, it is generally essential to make hardware management such as memory and threads unnecessary or optional in the language specification, in terms of the language made for music, so that the user can focus on musical tasks as suggested by McCartney, the developer of SuperCollider, argues[11, p61].

Kronos (and Meta-Sequencer) is also similar language that focuses on self-extensibility. Kronos is more strict functional language based on System $F\omega$, and it is more expressive as it can describe generic signal processor by parameterizing inputs and outputs of processor as lists. Its internal representation, however, is a graph structure[16, p23] like Faust. An internal representation in *mimium* is AST and SSA-form IR, more like to IR of general programming languages.

4.2 Remaining Problems

The following issues remain in *mimium* when using it practically as a unified language for music. First, the way of describing discrete events using a task scheduler is much more like an imperative paradigm that is apart from the functional design pattern in signal processing.

When using @ operators, deferred functions are not executed immediately; this inevitably leads to the use of void-type functions with no return value and with side effects (destructive assignment of variables), following the form of imperative programming. Its programming style is far apart from the notation of connecting signal processing between the return value and the argument between functions.

The combination of closures and temporal recursion, as in Listing 15, would allow us to abstract discrete values as functions and confine side effects within the function; however, this is not possible in the current implementation because the lifetime of a local variable defined in a function is closed within that function definition. If the compiler can statically determine how long a variable captured in a closure can survive by performing

lifetime analysis[20], it would be possible to abstract discrete values without changing the language specification itself.

```
fn frp_constructor(period){
  n = 0
  modifier = |x|{
    n = x //capture freevar
    modifier(n+1)@(now+period)
  }
  modifier(0)@0
  get = ||{ n }
  return get
}
val = frp_constructor(1000)
event_val = val()
```

Listing 15. Example of encapsulating a temporal discrete value not realized in current implementation of *mimium*.

Another problem is that parametric replication of functions with states is not possible in the current implementation that can be realized in Faust using a pattern matching technique. Consider the example code in Listing 16 that inputs an arbitrary number of filters and adds the outputs together. Here, we assume that the function filter is a stateful function of some kind.

```
fn filterbank(N,input,lowestfreq,margin,Q,
    filter){
  if(N>0){
    return filter(input,lowestfreq+N*margin,Q)
        +  filterbank(N-1,input,lowestfreq,
    margin,Q,filter)
  }else{
    return 0
  }
}
```

Listing 16. Example of parametric replication of signal processor that cannot be realized in current implementation of *mimium*.

In the current implementation, the compiler cannot compile the code correctly because the compiler cannot determine how many instances of the state variable for the filter are needed statically. To solve this problem, partial application of the constant N to the function should be performed before the conversions of state variables.

In the future, the compiler will need to be modified to introduce a constant folding step between type inference and stateful function conversion.

The current semantics, furthermore, has a problem that the type system does not distinguish whether the argument is a constant or not. For example, if a function that returns some time-varying float is passed to N in function filterbank, it is allowed at the type checker level, but it fails at the constant folding stage. Semantically, this constant folding can be seen as describing two stages of computation in a single source code: one that determines the data flow of signal processing at the compile time, and the other that runs at run time.

This situation is similar to the paradigm of multi-stage computation such as templates/constexpr in C++ and MetaML[29]. Introducing a type system for multi-stage computation would solve the problem that the type checker cannot distinguish whether a variable is a constant or not, because it can distinguish the stage of computation (in this case compile-time and runtime) a variable belongs to.

In addition, because multi-stage computation can be used as an expressive macro[4], it is possible to build more specialized DSLs for specific expressions on *mimium*, just like developing DSLs built on top of SuperCollider, for instance, TidalCycles[12], FoxDot[5], and IXI[10], but in the same language system not like server/-client model.

In addition, *mimium*'s DSP is based on the sample-by-sample format similar to Faust, and it is not possible to write functions such as FFT and granular synthesis that process multiple samples as vectors at once. Considering that in Kronos, this can be achieved by adding a built-in function to convert the sample rate, *mimium* also requires new semantics for the block computation.

5 Conclusion and Future Work

In this paper, we have described the design and implementation of *mimium*, a new programming language for music. *mimium* is characterized by the fact that it combines the discrete processing in the time domain and the signal processing that has been a problem in music programming languages.

As a language specification, music and signal processing can be written without considering the hardware, as memory management is not required and type inference is available.

The design and implementation of *mimium* are based on general programming languages with minimum features for music such as @ operator and stateful functions such as UGen, to allow users to concentrate on musical tasks while ensuring that it is easily extensible on the language itself. In contrast, the current major research issues are the need to implement lifetime analysis of variables such that discrete events can be described functionally rather than imperatively, and the need to implement constant folding such that stateful functions can be parametrically replicated. In addition, the possibility of introducing the paradigm of multi-stage computation to increase type safety and self-extensionality was implied.

As a future application of the *mimium* language, we aim to use it not only as a creative tool for creating computer music, but also as an infrastructure for distributing musical works as programs. Faust, Soul, and Vult have played an infrastructural role by allowing the same DSP algorithm to be used across a wide range of platforms such as audio plug-ins, web applications, and hardware. While *mimium* incorporates discrete event processing into the language specification, the compiler implementation itself is relatively simple by design, and the runtime features are kept to a minimum, making it easy to reduce the binary size. In addition, the compiler, runtime, execution engine and audio driver within the runtime, are designed to be modular, to ensure that they can be flexibly reconfigured through the addition of various configurations such as DSP languages, for example, a web-based backend and an audio driver for file input/output.

In other words, *mimium* can enable the easy distribution of music generated by a program without fixing it via recording or rendering; therefore, it has the potential to serve as an infrastructure for codes as the musical medium.

To make it easy to use it as a practical tool for such applications, we are working on implementing environment variables (values that change depending on the execution environment even for the same code, such as the sample rate), enhancing IO such as MIDI and OSC support, enriching the library, and developing a mechanism to simplify code distribution, such as a package manager.

Additionally, a more formal definition of the languages and the type system, and consideration of a benchmark are the remaining issues.

Acknowledgments

A part of this study was supported by JSPS KAKENHI (Grant No. JP19K21615), 2019 Exploratory IT Human Resources Project (The MITOU Program) by IPA: INFORMATION-TECHNOLOGY PROMOTION AGENCY, Japan and all the contributions for the development of *mimium* including documentations and financial sponsorships[4].

References

[1] Samuel Aaron and Alan F. Blackwell. 2013. From Sonic Pi to overtone: Creative musical experiences with domain-specific and functional languages. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP* (2013), 35–46. https://doi.org/10.1145/2505341.2505346

[2] Roger B. Dannenberg. 2018. Languages for Computer Music. *Frontiers in Digital Humanities* 5 (nov 2018). https://doi.org/10.3389/fdigh.2018.00026

[3] Roger B Dannenberg and Ross Bencina. 2005. Design Patterns for Real-Time Computer Music Systems. In *ICMC 2005 Workshop on Real Time Systems Concepts for Computer Music*. https://www.researchgate.net/publication/242648768_Design_Patterns_for_Real-Time_Computer_Music_Systems

[4] Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)* 36, 10 (oct 2001), 74–85. https://doi.org/10.1145/507669.507646

[5] Ryan Kirkbride. 2016. FoxDot: Live coding with python and supercollider. In *Proceedings of the International Conference on Live Interfaces*. 194–198.

[6] Carol Klabnik, Steve and Nichols. 2020. The Rust Programming Language. https://doc.rust-lang.org/book/

[7] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75. https://llvm.org/pubs/2004-01-30-CGO-LLVM.html

[8] Victor Lazzarini. 2013. The Development of Computer Music Programming Systems. *Journal of New Music Research* 42, 1 (2013), 97–110. https://doi.org/10.1080/09298215.2013.778890

[9] Erik de Castro Lopo. 1990. libsndfile. http://www.mega-nerd.com/libsndfile/

[10] Thor Magnusson. 2011. The IXI Lang: A SuperCollider Parasite for Live Coding. *International Computer Music Conference Proceedings* 2011 (2011). http://hdl.handle.net/2027/spo.bbp2372.2011.101

[11] James McCartney. 2002. Rethinking the computer music language: SuperCollider. *Computer Music Journal* 26, 4 (dec 2002), 61–68. https://doi.org/10.1162/014892602320991383

[12] Alex McLean. 2014. Making programming languages to dance to: Live coding with tidal. In *FARM 2014 - Proceedings of the 2014 ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*. Association for Computing Machinery, New York, New York, USA, 63–70. https://doi.org/10.1145/2633638.2633647

[13] Microsoft. 2020. Functions - F# — Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/functions/#function-composition-and-pipelining

[14] Vesa Norilo. 2015. Kronos: A Declarative Metaprogramming Language for Digital Signal Processing. *Computer Music Journal* 39, 4 (2015), 30–48. https://doi.org/10.1162/COMJ_a_00330

[15] Vesa Norilo. 2016. Kronos Meta-Sequencer – From Ugens to Orchestra, Score and Beyond. In *Proceedings of the International Computer Music Conference*. 117–122.

[16] Vesa Norilo. 2016. *Kronos: Reimagining musical signal processing*. Ph.D. Dissertation. University of the Arts Helsinki.

[17] Y. Orlarey, D. Fober, and S. Letz. 2004. Syntactical and semantical aspects of Faust. *Soft Computing* 8, 9 (2004), 623–632. https://doi.org/10.1007/s00500-004-0388-1

[18] Miller Puckette. 2002. Max at seventeen. *Computer Music Journal* 26, 4 (2002), 31–43. https://doi.org/10.1162/014892602320991356

[19] Miller S. Puckette. 1997. Pure Data. In *International Computer Music Conference Proceedings*. Michigan Publishing, University of Michigan Library. http://hdl.handle.net/2027/spo.bbp2372.1997.060

[4] https://github.com/mimium-org/mimium#contributors

[20] Cristina Ruggieri and Thomas P. Murtagh. 1988. Lifetime analysis of dynamically allocated objects. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, Vol. Part F130193. Association for Computing Machinery, 285–293. https://doi.org/10.1145/73560.73585

[21] Leonardo Laguna Ruiz. 2020. Vult Language. http://modlfo.github.io/vult/

[22] Spencer Salazar and Ge Wang. 2012. CHUGENS, CHUBGRAPHS, CHUGINS: 3 TIERS FOR EXTENDING CHUCK. In *International Computer Music Conference Proceedings*. 60–63. http://hdl.handle.net/2027/spo.bbp2372.2012.010

[23] Gary P. Scavone. 2002. RtAudio: A Cross-Platform C++ Class for Realtime Audio Input/Output. In *Proceedings of the 2002 International Computer Music Conference*. Goteborg, Sweden.

[24] Andrew Sorensen and Henry Gardner. 2010. Programming With Time Cyber-physical programming with Impromptu. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '10*. ACM Press, New York, New York, USA.

[25] Andrew Carl Sorensen. 2018. *Extempore: The design, implementation and application of a cyber-physical programming language*. Ph.D. Dissertation. The Australian National University. https://doi.org/10.25911/5D67B75C3AAF0

[26] Diomidis Spinellis. 2001. Notable design patterns for domain-specific languages. *Journal of Systems and Software* 56, 1 (feb 2001), 91–99. https://doi.org/10.1016/S0164-1212(00)00089-3

[27] Julian Storer. 2019. SOUL_Overview.md. https://github.com/soul-lang/SOUL/blob/master/docs/SOUL_Overview.md

[28] Eijiro Sumii. 2005. MinCaml: A simple and efficient compiler for a minimal functional language. In *FDPE'05 - Proceedings of the ACM SIGPLAN 2005 Workshop on Functional and Declarative Programming in Education*. ACM Press, New York, New York, USA, 27–38. https://doi.org/10.1145/1085114.1085122

[29] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)* 32, 12 (dec 1997), 203–214. https://doi.org/10.1145/258994.259019

[30] Ge Wang, Perry R Cook, and Spencer Salazar. 2015. ChucK: A Strongly Timed Computer Music Language. *Computer Music Journal* 39, 4 (2015), 10–29. https://doi.org/10.1162/COMJ_a_00324

A EBNF definition of *mimium* language

The language specification of *mimium* (at the version 0.4.0) is shown below in EBNF notation. The precedence of operators is omitted but follows the order of precedence of general programming languages, and the @ operator is assumed to have the lowest precedence.

```
number ::= ?numbers?
symbol ::= ?all_alnum_and_underscore?
string ::= ?double_quote?<symbol>?double_quote?
rvar ::= <symbol>
lvar ::= <symbol> (":"<type>)?
lvar_args ::= <lvar> | <lvar_args> "," <lvar>
binop ::= "+"|"-"|"*"|"/"|"^"|"=="|
    "!="|">="|"<="|">"|"<"|"&&"| "||"
uniop ::= "-" | "!"
infix ::= <expr> <binop> <expr> | <uniop> <expr>
field ::= <expr>"."<symbol>
app ::= <expr>"(" <expr_args> ")"
lambda ::= "|" <lvar_args> "|" ("->" <type>)? <expr>
if ::=  "if" "("<expr>")" <expr> ("else" <expr>)?
block ::= "{"<statements>"}"
expr ::= "self"|"now"|<number>|<string>|<rvar>|<infix>|<field>|<app>|<lambda>|<if>|<block>
expr_args ::= <expr> | <expr_args> "," <expr>

statements ::= <statement> | <statements> ? linebreak? <statement>
statement ::= <app>|<schedule>|<fndef>|<assign>|<lettuple>|<return>|<typealias>
schedule ::= expr "@" expr
fndef ::=  "fn" <symbol> "("<lvar_args>")" ("->" <type>)? <block>
assign ::= <lvar> "=" <expr>
lettuple ::= <lvar_args> "=" <expr>
return ::= "return" <expr>

type ::= "void"|"float"|"string"|<symbol>|<tupletype>|<fntype>|<recordtype>
types ::= <type> | <types>","<type>
tupletype ::= "()" | "("<types>")"
fntype ::= <tupletype>"->"<type>
recordtype ::= "{"<typekeyvals>"}"
typekeyvals ::= <typekeyval> | <typekeyvals>","<typekeyval>
typekeyval ::= <string>":"<type>
typealias ::= "type" <symbol> "=" <type>

program ::= <statements>
```

Listing 17. EBNF definition of *mimium*