Poing Impératif: Compiling Imperative and Object Oriented Code to Faust

Kjetil Matheussen
Norwegian Center for Technology in Music and the Arts. (NOTAM)
k.s.matheussen@notam02.no

Abstract

This paper presents a new compiler called *Poing Impératif*. Poing Impératif extends Faust with common features from imperative and object oriented languages.

Imperative and object oriented features make it easier to start using Faust without having to immediately start thinking in fully functional terms. Furthermore, imperative and object oriented features may enable semi-automatic translation of imperative and object oriented code to Faust.

Performance seems equal to pure Faust code if using one of Faust's own delay operators instead of the array functionality provided by Poing Impératif.

Keywords

Faust, functional programming, imperative programming, object oriented programming, compilation techniques.

1 Introduction

Poing Impératif is a new compiler that extends Faust with imperative and object oriented features.[1]

The input code is either Poing Impératif code (described in this paper), or pure Faust code. Pure faust code and Poing Impératif code can be freely mixed. Pure Faust code goes through Poing Impératif unchanged, while Poing Impératif code is translated to pure Faust code.

1.1 About Faust

Faust [Orlarey et al., 2004] is a programming language developed at the Grame Institute in Lyon. Faust is a fully functional language especially made for audio signal processing. Faust code is compact and elegant, and the compiler produces impressively efficient code. It is simple to compile Faust code into many types of formats such as LADSPA plugins, VST plugins, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, etc. Faust also offer options to automatically take advantage of multiple processors [Orlarey et al., 2009; Letz et al., 2010] and generate code which a C++ compiler is able to vectorize (i.e. generating SIMD assembler instructions) [Scaringella et al., 2003].

1.2 Contributions of Poing Impératif

Purely functional programming is unfamiliar for many programmers, and translating existing DSP code written in object oriented or imperative style into Faust is not straight forward because of different programming paradigms.

Poing Impératif can:

1. Make it easier to start using Faust without having to immediately start thinking in fully functional terms.
2. Make it easier to translate imperative and object oriented code to Faust. Porting programs to Faust makes them:
   (a) Easily available on many different platforms and systems.
   (b) Automatically take advantage of multiple processors.
   (c) Possibly run faster. Faust automatically optimizes code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked.

1.3 Usage

By default, Poing Impératif starts the Faust compiler automatically on the produced code. Any command line option which is unknown to Poing Impératif is sent further to the Faust compiler. Example:

```
$poing-imperatif -a jack-gtk.cpp -vec freeverb_oo.dsp >freeverb.cpp
$g++  freeverb.cpp -O2 ‘pkg-config --libs --cflags gtk+-2.0‘ -ljack -o freeverb
$./freeverb
```

[1]Note that since inheritance (subclasses) and polymorphism (method overloading) are not supported, Poing Impératif should probably not be categorized as an OO language.

2 Features

- Setting new values to variables. (I.e. providing imperative operators such as =, ++, +=, etc.)
- Conditionals. (*if*/*else*)
- Arrays of floats or ints.
- *return* operator.
- Classes, objects, methods and constructors.
- Optional explicit typing for numeric variables. Function types are implicitly typed, while object types are explicitly typed. The void type is untyped.
- All features of Faust are supported. Faust code and Poing Impératif code can be mixed.

3 Syntax (EBNF)

```
class        = "class" classname ["(" [var_list] ")"]
               "{"
                   {class_elem}
               "}" .

var_list     = var {"," var} .
var          = [number_type] varname .
number_type  = "int" | "float" .

class_elem   = array_decl | object_decl | method | statement .

array_decl   = number_type arrayname "[" expr "]" ["=" expr] ";" .
object_decl  = classname objectname ["(" [expr] ")"] ";" .
method       = [number_type] methodname "(" [var_list] ")"
               "{"
                  {statement}
               "}" .

expr         = faust_expression | inc_assign | dec_assign | class
             | method_call | object_var | array_ref .
(* Inside classes, faust expressions are extended to expr! *)

object_var   = objectname "." varname .
array_ref    = arrayname "[" expr "]" .

statement    = method_call ";" | block | single_decl
             | if | return | assignment .

method_call  = objectname "." methodname "(" [expr] ")" .
block        = "{" {statement} "}" .
single_decl  = number_type name_list ["=" expr] ";" .
if           = "if" "(" expr ")" statement ["else" statement] .
return       = "return" expr ";" .

assignment   = set_assign | inc_assign | dec_assign
             | cni_assign | obvar_set | array_set .
set_assign   = name_list "=" expr ";" .
inc_assign   = name "+" "+" ";" | "+" "+" name ";"  .
dec_assign   = name "-" "-" ";" | "-" "-" name ";"  .
cni_assign   = name assign_op "=" expr ";" .
assign_op    = "+" | "-" | "*" | "/"  .
obvar_set    = objectname "." varname "=" expr ";" .
array_set    = arrayname "[" expr "]" "=" expr ";" .

classname    = name .
varname      = name .
arrayname    = name .
objectname   = name .
methodname   = name .

name_list    = name {"," name} .
name         = alpha, {alpha | digit | "_"} .
```

4 Example of C++ code translated to Poing Impératif

The C++ implementation of the Freeverb[2] allpass filter looks like this:

```
class Allpass{
 float feedback;
 int   bufsize;
 int   bufidx;
 float *buffer;
 Allpass(float bufsize,float feedback){
   this.bufsize  = bufsize;
   this.feedback = feedback;
   buffer=calloc(sizeof(float),bufsize);
 }
}
float Allpass::process(float input){
  float bufout    = buffer[bufidx];
  float output    = -input + bufout;
  buffer[bufidx] = input + (bufout*feedback);
  if(++bufidx>=bufsize)
    bufidx = 0;
  return output;
}
```

A semi-automatic translation to Poing Impératif yields:

```
class Allpass(int bufsize,float feedback){
  float buffer[bufsize];
  int   bufidx;
  process(float input){
    float bufout   = buffer[bufidx];
    float output   = -input + bufout;
    buffer[bufidx] =  input + (bufout*feedback);
    if(++bufidx>=bufsize)
      bufidx = 0;
    return output;
  }
};
```

5 Constructor

In the Allpass example above, the Poing Impératif class had a slightly different form than the C++ version since a constructor was not needed.

For classes requiring a constructor, imperative code can be placed directly in the class block. A class describing a bank account giving 50 extra euros to all to new accounts, can be written like this:

```
class Account(int euros){

  euros += 50; // Constructor!

  debit(int amount){
    euros -= amount;
  }
  deposit(int amount){
    euros += amount;
  }
}
```

[2]Freeverb is a popular reverb algorithm made by "Jezar at Dreampoint". See Julius O. Smith's Freeverb page for more information about it: `https://ccrma.stanford.edu/~jos/pasp/Freeverb.html` (The web page is from his book "Spectral Audio Signal Processing".)

6 Accessing a Poing Impératif class from Faust

The *process* method is used to bridge Poing Impératif and Faust. If a class has a method called *process*, and that *process* method contains at least one *return* statement, Poing Impératif creates a Faust function with the same name as the class. We call this function the *class entry function*.

The arguments for the class entry function is created from the class arguments and the *process* method arguments.

⇒ For instance, the entry function for this class:

```
class Vol(float volume){
  process(float input){
    return input*volume;
  }
}
```

...looks like this:

```
Vol(volume,input) = input*volume;
```

...and it can be used like this:

```
half_volume = Vol(0.5);
process(input) = half_volume(input);
```

6.1 Recursive variables

In case the class has variables which could change state between calls to the class entry function, we use the recursive operator (~) to store the state of those variables.

⇒ For instance, this class:

```
class Accumulator{
  int sum;
  process(int inc){
    sum += inc;
    return sum;
  }
}
```

...is transformed into the following Faust code:[3]

```
Accumulator(inc) = (func0 ~ (_,_)) : retfunc0 with{
  func0(sum,not_used) = (sum+inc, inc); // sum += inc;
  retfunc0(sum,inc) = sum;              // return sum;
};
```

6.2 Constructors in the class entry function

In case a class contains constructor code or numeric values initialized to a different value than 0 or 0.0, an additional state variable is used to keep track of whether the function is called for the first time. In case this additional state variable has the value 0 (i.e. it is the first first time the class entry function is called), a special constructor function is called first to initialize those state variables.

[3]Simplified for clarity.

7 Conversion from Poing Impératif to Faust

7.1 Setting values of variables

Faust is a purely functional languages. It is not possible to give a variable a new value after the initial assignment, as illustrated by the following pseudocode:

```
Possible:
{
  int a = 5;
  return a
}

Impossible:
{
  int a = 5;
  a = 6;
  return a;
}
```

One way to circumvent this is to use a new variable each time we set new values. For instance, adding 1 to a would look like this: $a_2 = a_1 + 1$.

However, Poing Impératif uses a different approach, which is to make all operations, including assignments, into function calls.

⇒ For example, the following code:

```
float a = 1.0, b=0.0;
a = a + 1.0;
b = a + 2.3;
```

...is transformed into:

```
func0(a,b) = func1(1.0   , 0.0);   // a = 1.0, b=0.0
func2(a,b) = func3(a+1.0, b);      // a = a+1.0
func4(a,b) = func5(a     , a+2.3); // b = a+2.3
```

7.2 Conditionals

When every operation is a function call, branching is simple to implement.

⇒ For instance, the following code:

```
if(a==0)
  a=1;
else
  a=2;
```

...is transformed into:

```
func1(a) = if(a==0,func2(a),func3(a));  // if(a==0)
func2(a) = func4(1);                     // a=1
func3(a) = func4(1);                     // a=2
```

if is here a Faust macro [Gräf, 2010], and it is made to supports multiple signals. The *if* macro looks like this:

```
if(a,(k1,k2),(k3,k4)) = if(a,k1,k3),if(a,k2,k4);
if(a,k,k)             = k;
if(a,k1,k2)           = select2(a,k2,k1);
```

7.3 Methods

In Poing Impératif, an object is a list of all the variables used in a class (including method arguments). Behind the scene, every method receives the “this” object (and nothing more). Every method also returns the “this” object (and nothing more). Naturally, the “this” object may be modified during the execution of a method.

⇒ For instance, the method *add* in:

```
class Bank{
  int a;
  add(int how_much){
   a += how_much;
  }
}
```

...is transformed into:

```
Bank__add(a,how_much) = func0(a,how_much) with{
  func0(a,how_much) = (a+how_much, how_much); // a += how_much
};
```

If a method takes arguments, the corresponding variable in the “this” object is set automatically by the caller before the method function is called.

7.4 Return

A special *__return* function is created for each method which calls *return.* The reason for using the *__return* function to return values, instead of for instance using a special variable to hold a return value, is because it is possible to return more than one value (i.e. to return parallel signals). Furthermore, it is probably cleaner to use a special *__return* function than to figure out how many signals the various methods might return[4] and make corresponding logic to handle situations that might show up because of this.

The *__return* function uses an ’n’ argument (holding an integer) to denote which of the return expressions to return.

⇒ For instance, the *process* and *process__return* functions generated from this class:

```
class A{
  process(int selector){
   if(selector)
     return 2;
   else
     return 3;
  }
}
```

[4]It is also quite complicated to figure out how many output signals an expression has. See [Orlarey et al., 2004].

...look like this:

```
A__process(selector,n) = func0(selector,n) with{
  func0(selector,n) = if(selector,func1(selector,n),func2(selector,n));
  func1(selector,n) = (selector,0); // First return
  func2(selector,n) = (selector,1); // Second return
};
A__process__return(selector,n) =
  if(n==0,
     2,    // Return 2 from the first return
     3);   // Return 3 from the second return
```

7.5 Arrays

Faust has a primitive called *rwtable* which reads from and writes to an array. The syntax for *rwtable* looks like this:

```
rwtable(size, init, write_index, write_value, read_index);
```

Using *rwtable* to implement imperative arrays is not straight forward. The problem is that *rwtable* does not return a special array object. Instead, it returns the numeric value stored in the cell pointed to by ’read_index’. This means that there is no array object we can send around.

Our solution is to use *rwtable* only when reading from an array. When we write to an array, we store the new value and array position in two new variables.

⇒ For instance, the body of *process* in the following class:

```
class Array{
  float buf[1000]=1.0;
  process(int i){
    float a = buf[i];
    buf[i]  = a+1.0;
  }
}
```

...is transformed into:

```
/* float a = buf[i] */
func0(a,                                  i, buf_pos, buf_val) =
func1(rwtable(1000,1.0,buf_pos,buf_val,i), i, buf_pos, buf_val);

/* buf[i] = a+1.0 */
func1(a,                                  i, buf_pos, buf_val) =
     (a,                                  i, i,       a+1.0);
```

However, this solution has a limitation: If a buffer is written two times in a row, only the second writing will have effect.

It might be possible to use Faust’s foreign function mechanism to achieve complete array functionality, by implementing arrays directly in C. However, this could limit Faust’s and the C compilers ability to optimize. It would also complicate the compilation process, and limit Poing Impératif to only work with C and C++. (i.e. it would not work with Java, LLVM or other languages (or other bitcode/binary formats) Faust supports unless we implement array interfaces to Faust for those as well.)

A fairly relevant question is how important full array functionality is? Since full array functionality is not needed for any programs written for pure Faust, it's tempting to believe this functionality can be skipped.[5]

8 Performance compared to Faust

In Poing Impératif, Freeverb can be implemented like this:[6]

```
class Allpass(int bufsize, float feedback){
  int   bufidx;
  float buffer[bufsize];
  process(input){
   float bufout   = buffer[bufidx];
   float output   = -input + bufout;
   buffer[bufidx] = input + (bufout*feedback);
   if(++bufidx>=bufsize)
     bufidx = 0;
   return output;
  }
}

class Comb(int bufsize, float feedback, float damp){
  float filterstore;
  int   bufidx;
  float buffer[bufsize];
  process(input){
    filterstore    = (buffer[bufidx]*(1.0-damp)) + (filterstore*damp);
    float output   = input + (filterstore*feedback);
    buffer[bufidx] = output;
    if(++bufidx>=bufsize)
      bufidx = 0;
    return output;
  }
}

class MonoReverb(float fb1, float fb2, float damp, float spread){
  Allpass allpass1(allpasstuningL1+spread, fb2);
  Allpass allpass2(allpasstuningL2+spread, fb2);
  Allpass allpass3(allpasstuningL3+spread, fb2);
  Allpass allpass4(allpasstuningL4+spread, fb2);
  Comb comb1(combtuningL1+spread, fb1, damp);
  Comb comb2(combtuningL2+spread, fb1, damp);
  Comb comb3(combtuningL3+spread, fb1, damp);
  Comb comb4(combtuningL4+spread, fb1, damp);
  Comb comb5(combtuningL5+spread, fb1, damp);
  Comb comb6(combtuningL6+spread, fb1, damp);
  Comb comb7(combtuningL7+spread, fb1, damp);
  Comb comb8(combtuningL8+spread, fb1, damp);

  process(input){
    return allpass1.process(
             allpass2.process(
               allpass3.process(
                 allpass4.process(
                   comb1.process(input) +
                   comb2.process(input) +
                   comb3.process(input) +
                   comb4.process(input) +
                   comb5.process(input) +
                   comb6.process(input) +
                   comb7.process(input) +
                   comb8.process(input)
                 )
               )
             )
           );
  }
}

class StereoReverb(float fb1, float fb2, float damp, int spread){
  MonoReverb rev0(fb1,fb2,damp,0);
  MonoReverb rev1(fb1,fb2,damp,spread);
  process(float left, float right){
    return rev0.process(left+right),
           rev1.process(left+right);
  }
}

class FxCtrl(float gain, float wet, Fx){
  process(float left, float right){
    float fx_left, fx_right = Fx(left*gain, right*gain);
    return left *(1-wet) + fx_left *wet,
           right*(1-wet) + fx_right*wet;
  }
}

process = FxCtrl(fixedgain,
                 wetSlider,
                 StereoReverb(combfeed,
                              allpassfeed,
                              dampSlider,
                              stereospread
                             )
                 );
```

The version of freeverb included with the Faust distribution (performing the exact same computations) looks like this:[7]

```
allpass(bufsize, feedback) =
  (_,_ <: (*(feedback),_:+:@(bufsize)), -) ~ _ : (!,_);

comb(bufsize, feedback, damp) =
  (+:@(bufsize)) ~ (*(1-damp) : (+ ~ *(damp)) : *(feedback));

monoReverb(fb1, fb2, damp, spread)
        = _ <:  comb(combtuningL1+spread, fb1, damp),
                        comb(combtuningL2+spread, fb1, damp),
                        comb(combtuningL3+spread, fb1, damp),
                        comb(combtuningL4+spread, fb1, damp),
                        comb(combtuningL5+spread, fb1, damp),
                        comb(combtuningL6+spread, fb1, damp),
                        comb(combtuningL7+spread, fb1, damp),
                        comb(combtuningL8+spread, fb1, damp)
                +>
                        allpass (allpasstuningL1+spread, fb2)
                :       allpass (allpasstuningL2+spread, fb2)
                :       allpass (allpasstuningL3+spread, fb2)
                :       allpass (allpasstuningL4+spread, fb2)
                ;

stereoReverb(fb1, fb2, damp, spread) =
  + <: monoReverb(fb1, fb2, damp, 0),
       monoReverb(fb1, fb2, damp, spread);

fxctrl(gain,wet,Fx) =  _,_
                       <: (*(gain),*(gain) : Fx : *(wet),*(wet)),
                          *(1-wet),*(1-wet)
                       +> _,_;

process = fxctrl(fixedgain,
                 wetSlider,
                 stereoReverb(combfeed,
                              allpassfeed,
                              dampSlider,
                              stereospread
                             )
                );
```

Benchmarking these two versions against each other showed that the version written for pure Faust was approximately 30% faster than the version written for Poing Impératif.

[5]One situation where it quite undoubtedly would be useful to write or read more than once per sample iteration, is for doing resampling. But for resampling, the Faust developers are currently working on a implementing a native solution. [Jouvelot and Orlarey, 2009]

[6]The values for the constants *combtuningL1*, *combtuningL2*, *allpasstuningL1*, etc. are defined in the file "examples/freeverb.dsp" in the Faust distribution.

[7]Slightly modified for clarity.

After inspecting the generated C++ source for the Allpass class and the Comb class, it seemed like the only reason for the difference had to be the use of *rwtable* to access arrays.

By changing the Poing Impératif versions of *Comb* and *Allpass* to use Faust's delay operator @ instead of *rwtable*, we get this code:

```
class Allpass(int bufsize, float feedback){
    float bufout;
    process(float input){
        float output = -input + bufout;
        bufout       = input + (bufout*feedback) : @(bufsize);
        return output;
    }
}

class Comb(int bufsize, float feedback, float damp){
    float filterstore;
    float bufout;
    process(float input){
        filterstore  = (output*(1.0-damp)) + (filterstore*damp);
        bufout       = input + (filterstore*feedback) : @(bufsize);
        return bufout;
    }
}
```

Now the pure Faust version was only 7.5% faster than the Poing Impératif version. This result is quite good, but considering that semantically equivalent C++ code were generated both for the Comb class and the Allpass class (the Allpass class was even syntactically equivalent),[8] plus that optimal Faust code were generated for the three remaining classes (MonoReverb, StereoReverb, and FxCtrl), both versions should in theory be equally efficient. However, after further inspection of the generated C++ code, a bug in the optimization part of the Faust compiler was revealed.[9] After manually fixing the two non-optimal lines of C++ code caused by this bug in the Faust compiler, both versions of Freeverb produce similarly efficient code. The final two C++ sources also look semantically equivalent.

[8]Semantically equivalent means here that the code is equal, except that variable names might differ, independent statements could be placed in a different order, or that the number of unnecessary temporary variables differ.

[9]The decreased performance was caused by two different summing orders of the same group of signals (which is a bug, order is supposed to be equal). This again caused sub-summations not to be shared, probably because equal order is needed to identify common subexpressions. The bug only causes a slight decreased performance in certain situations, it does not change the result of the computations. The bug can also be provoked by recoding the definition of *allpass* in the pure Faust version of Freeverb to:

```
allpass(bufsize, feedback, input) = (process ~ (_,!)) : (!,_) with{
  process(bufout) = (
    (input + (bufout * feedback ): @ (bufsize )),
    (-input + bufout )
  );
};
```

...which is just another way to write the same function.

The bug was reported right before this paper was submitted, it has been acknowledged, and the problem is being looked into. Thanks to Yann Orlarey for a fast response and for confirming what could be the problem.

9 Implementation

The main part of Poing Impératif is written in the Qi language [Tarver, 2008]. Minor parts of the source are written in C++ and Common Lisp. Poing Impératif uses Faust's own lexer.

The source is released under GPL and can be downloaded from:
`http://www.notam02.no/arkiv/src/`

10 Acknowledgments

Funded by Integra, a 3-year project part-funded by the Culture programme of the European Commission.

References

Albert Gräf. 2010. Term rewriting extension for the faust programming language. *Proceedings of the Linux Audio Conference 2010*, pages 117–121.

Pierre Jouvelot and Yann Orlarey. 2009. Semantics for multirate faust. *New Computational Paradigms for Computer Music - Editions Delatour France.*

Stephane Letz, Yann Orlarey, and Dominique Fober. 2010. Work stealing scheduler for automatic parallelization in faust. *Proceedings of the Linux Audio Conference 2010*, pages 147–152.

Y. Orlarey, D. Fober, and S. Letz. 2004. Syntactical and semantical aspects of faust. *Soft Comput.*, 8:623–632, September.

Yann Orlarey, Stephane Letz, and Dominique Fober. 2009. Automatic parallelization of faust code. *Proceedings of the Linux Audio Conference 2009.*

Nicolas Scaringella, Yann Orlarey, Stephane Letz, and Dominique Fober. 2003. Automatic vectorization in faust. *Actes des Journes d'Informatique Musicale JIM2003, Montbeliard - JIM.*

Mark Tarver. 2008. *Functional Programming in Qi (second edition).*