SKFaust: A functional Faust implementation

Qiantan Hong

Massachusetts Institute of Technology

December 11, 2018

Table of Contents

Introduction to FAUST

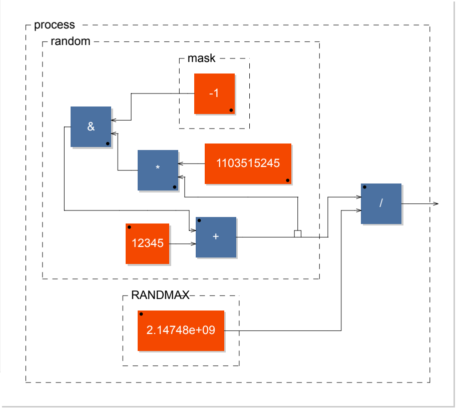
What is FAUST

- Faust (Functional Audio Stream) is a functional programming language for sound synthesis and audio processing with a strong focus on the design of synthesizers, musical instruments, audio effects, etc. Faust targets high-performance signal processing applications and audio plug-ins for a variety of platforms and standards.
- The core component of Faust is its compiler. It allows to "translate" any Faust digital signal processing (DSP) specification to a wide range of non-domain specific languages such as C++, C, JAVA, JavaScript, LLVM bit code, WebAssembly, etc. In this regard, Faust can be seen as an alternative to C++ but is much simpler and intuitive to learn.
- Thanks to a wrapping system called "architectures," codes generated by Faust can be easily compiled into a wide variety of objects ranging from audio plug-ins to standalone applications or smartphone and web apps, etc.

Introduction to FAUST
Quick overview of FAUST

A sample FAUST program:

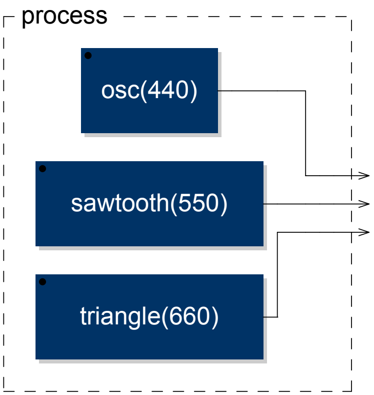
```
  noise = random
  / RANDMAX
with{
  mask =
  4294967295;
  random =
  +(12345)
  ~
  *(1103515245)
  & mask;
  RANDMAX =
  2147483647.0;
};
```



Introduction to FAUST
Quick overview of FAUST

Parallel composition:

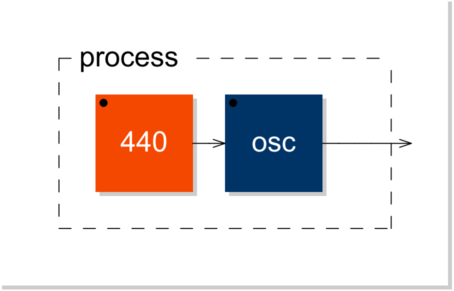
```
process =
os.osc(440),
os.sawtooth(550),
os.triangle(660);
```



Introduction to FAUST
Quick overview of FAUST

Sequential composition:

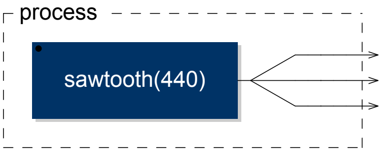
```
process =
440 : os.osc;
```



Introduction to FAUST
Quick overview of FAUST

Fan–out:

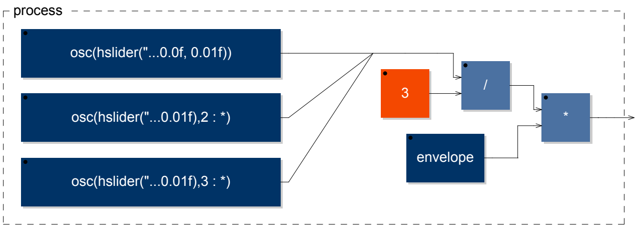
```
process =
os.sawtooth(440)
<: _,_,_;
```



Introduction to FAUST
Quick overview of FAUST

Fan-in:

```
freq = hslider("freq",440,50,3000,0.01);
gain = hslider("gain",1,0,1,0.01);
gate = button("gate");
envelope = gain*gate : si.smoo;
process = os.osc(freq),os.osc(freq*2),
os.osc(freq*3) :> /(3)*envelope;
```


process
osc(hslider("...0.0f, 0.01f))
osc(hslider("...0.01f),2 : *)
osc(hslider("...0.01f),3 : *)
3
/
*
envelope

Problems of current FAUST implementations

Problems of current FAUST implementations

- Lack of REPL (Readevalprint loop) environment.
  In a REPL, the user enters one or more expressions (rather than an entire compilation unit) and the REPL evaluates them and displays the results. A REPL gives quick feedback to the novice and is commonly used for nstantaneous prototyping.
  Current FAUST implementations have NO REPLs.
- Unsafe implementations.
  Current FAUST implementations are mostly written in C++, which tends to be vulnerable and buggy.
  (Just see GitHub issues)
- Type safety.
  Mysterious data type conversion during execution.
- Compilation speed and space consumption.
  Unnecessary C++ intermediate representation.
  Compiles very slow.

A better FAUST implementation

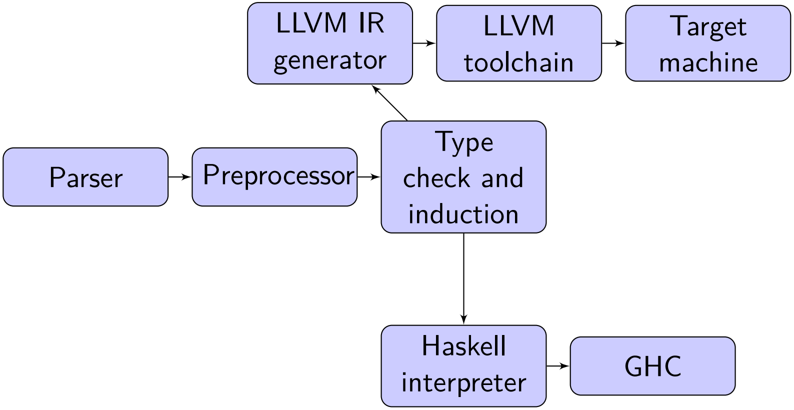
A better FAUST implementation

Key features

- An interactive developing environment.
- A safe implementation.
- Strict compile-time check and friendly error messages.

A better FAUST implementation

Architecture



Progress and challenges

Progress and challenges

Overview

- Current progress: Almost finished type checker and haskell interpreter.
- Challenge: Polymorphism;
  Flexible block algebra in FAUST; Realtime audio API.
- Plan: Finish the complete interpreter within this semester.

Progress and challenges

Polymorphism in FAUST

A simple example showing the issue:

```
simple = + 1;
problematic = +;
//Cannot no actual signal time at compile time
```

Original FAUST implementations don't have much static check, and the actual type is determined once actual signal is fed to the process. This will cause overhead and does not gruantee type correctness when feeding different types of signal to the DSP.

Progress and challenges

Polymorphism in FAUST

Current solution: decision tree search.

```
purgeCandidates :: [MultipleMonadError e a]->([MultipleMonadError e a],Int)
purgeCandidates [x] = let MultipleMonadError l a = x
                      in ([x],length l)
purgeCandidates (x:xs) = let (rs,rn) = purgeCandidates xs
                             MultipleMonadError l a = x in
                         case () of _
                                      | length l > rn -> (rs,rn)
                                      | otherwise -> ([x],length l)
typeFaustArrow i = case i of
  (ArrowFunc x) -> case x of
     (FaustIntFloatOp y) -> [return (TypeTag [FaustFloat,FaustFloat] [FaustFloat
     ]
        (ArrowoidFunc x)),
                             return (TypeTag [FaustInt,FaustInt] [FaustInt]
        (ArrowoidFunc x))]
     // ...
  (ArrowSplit a b) -> fst (purgeCandidates (liftA2 (***)
     (typeFaustArrow a) (typeFaustArrow b)))
  (ArrowCompose a b) -> fst (purgeCandidates ( liftA2 (>>>)
     (typeFaustArrow a) (typeFaustArrow b)))
```

Progress and challenges

Flexible block algebra

Such code in FAUST standard is legal:

```
incomplete = 1:+;
```

Quite unusual in other programming languages! The "+" has incomplete argument. However it can be regarded as a syntax sugar:

```
desugar = 1,_:+;
```

Progress and challenges

Flexible block algebra

Implementation: flexible diagram composition.

```
instance (Eq t, Show t) => DynamicArrow (MultipleMonadError String (TypeTag t f)
    ) where
  (>>>) i j = join ((\ x y -> let (TypeTag a b f1) = x
                                  (TypeTag c d f2) = y in
                                let o = (TypeTag a d (x .>>> y)) in
                                  case b == c of
                                    True -> return o
                                    False -> let sb = stripPrefix b c in case sb
      of
                                      Nothing -> (MultipleMonadError ["
     Incompatible type in series: "++(show b)++" and "++(show c)] (return o))
                                      Just t -> let a1 = return x
                                                    a2 = return y in
                                        (a1 *** (return (TypeTag t t unitoid)))
     >>> a2) <$> i <*> j)
  (***) = liftA2 (\x y -> let (TypeTag a b f1) = x
                              (TypeTag c d f2) = y in
                            TypeTag (a ++ c) (b ++ d) (x .*** y))
  unit = return (TypeTag [] [] unitoid)
```