An LLVM-based Signal-Processing-Compiler embedded in Haskell

Henning Thielemann
Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg,
Von-Seckendorff-Platz 1,
06110 Halle,
Germany,
henning.thielemann@informatik.uni-halle.de

Abstract

We discuss a programming language for real-time audio signal processing that is embedded in the functional language Haskell and uses the Low-Level Virtual Machine as back-end. With that framework we can code with the comfort and type safety of Haskell while achieving maximum efficiency of fast inner loops and full vectorisation. This way Haskell becomes a valuable alternative to special purpose signal processing languages.

Keywords

Functional progamming, Haskell, Low-Level Virtual machine, Embedded Domain Specific Language

1 Introduction

Given a data flow diagram as in Figure 1 we want to generate an executable machine program. First we must (manually) translate the diagram to something that is more accessible by a machine. Since we can translate data flows almost literally to function expressions, we choose a functional programming language as the target language, here `Haskell` [Peyton Jones, 1998]. The result can be seen in Figure 2. The second step is to translate the function expression to a machine oriented presentation. This is the main concern of our paper.

Since we represent signals as sequences of numbers, signal processing algorithms are usually loops that process these numbers one after another. Thus our goal is to generate efficient loop bodies from a functional signal processing representation. We have chosen the Low-Level Virtual-Machine (LLVM) [Lattner and Adve, 2004] for the loop description, because LLVM provides a universal representation for machine languages of a wide range of processors. The LLVM library is responsible for the third step, namely the translation of portable virtual machine code to actual machine code of the host processor.

Our contributions are

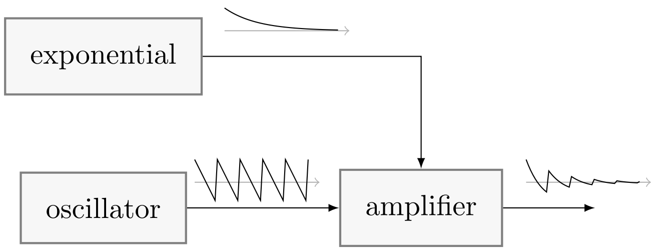

Figure 1: *Data flow for creation of a very simple percussive sound*

```
amplify
   (exponential halfLife amp)
   (osci Wave.saw phase freq)
```

Figure 2: *Functional expression for the diagram in Figure 1*

- a representation of an LLVM loop body that can be treated like a signal, described in Section 3.1,
- a way to describe causal signal processes, which is the dominant kind of signal transformations in real-time audio processing and which allows us to cope efficiently with multiple uses of outputs and with feedback of even small delays, guaranteed deadlock-free, developed in Section 3.2,
- and the handling of internal filter parameters in a way that is much more flexible than traditional control rate/sample rate schemes, presented in Section 3.3.

Due to space constraints we omitted some parts, like the use of vector arithmetic and according benchmarks, that you can find in [Thielemann, 2010b].

2 Background

We want to generate LLVM code from a signal processing algorithm written in a declarative way. We like to write code close to a data flow diagram and the functional paradigm seems to be appropriate.

We could design a new language specifically for this purpose, but we risk the introduction of design flaws. We could use an existing signal processing language, but usually they do not scale well to applications other than signal processing. Alternatively we can resort to an existing general purpose functional programming language or a subset of it, and write a compiler with optimisations adapted to signal processing needs. But writing a compiler for any modern "real-world" programming language is a task of several years, if not decades. A compiler for a subset of an existing language however would make it hard to interact with existing libraries. So we can still tune an existing compiler for an existing language, but given the complexity of modern languages and their respective compilers this is still a big effort. It might turn out that a change that is useful for signal processing kills performance for another application.

A much quicker way to adapt a language to a special purpose is the Embedded Domain Specific Language (EDSL) approach [Landin, 1966]. In this terminology "*embedded*" means, that the domain specific (or "special purpose") language is actually not an entirely new language, but a way to express domain specific issues using corresponding constructs and checks of the host language. For example, writing an SQL command as string literal in Java and sending it to a database, is not an EDSL. In contrast to that, Hibernate [Elliott, 2004] is an EDSL, because it makes database table rows look like ordinary Java objects and it makes the use of foreign keys safe and comfortable by making foreign references look like Java references.

In the same way we want to cope with signal processing in `Haskell`. In the expression

```
amplify
   (exponential halfLife amp)
   (osci Wave.saw phase freq)
```

the call to `osci` shall not produce a signal, but instead it shall generate LLVM code that becomes part of a signal generation loop later. In the same way `amplify` assembles the code parts produced by `exponential` and `osci` and defines the product of their results as its own result. In the end every such signal expression is actually a high-level LLVM macro and finally, we pass it to a driver function that compiles and runs the code. Where Hibernate converts Java expressions to SQL queries, sends them to a database and then converts the database answers back to Java objects, we convert `Haskell` expressions to LLVM bitcode, send it to the LLVM Just-In-Time (JIT) compiler and then execute the resulting code. We can freely exchange signal data between pure `Haskell` code and LLVM generated code.

The EDSL approach is very popular among `Haskell` programmers. For instance interfaces to the Csound signal processing language [Hudak et al., 1996] and the real-time software synthesiser SuperCollider [Drape, 2009] are written this way. This popularity can certainly be attributed to the concise style of writing `Haskell` expressions and to the ease of overloading number literals and arithmetic operators. We shall note that the EDSL method has its own shortcomings, most notably the *sharing problem* that we tackle in Section 3.2.

In [Thielemann, 2004] we have argued extensively, why we think that `Haskell` is a good choice for signal processing. Summarised, the key features for us are polymorphic but strong static typing and lazy evaluation. Strong typing means that we have a wide range of types that the compiler can distinguish between. This way we can represent a trigger or gate signal by a sequence of boolean values (type `Bool`) and this cannot be accidentally mixed up with a PCM signal (sample type `Int8`), although both types may be represented by bytes internally. We can also represent internal parameters of signal processes by opaque types that can be stored by the user but cannot be manipulated (cf. Section 3.3). Polymorphic typing means that we can write a generic algorithm that can be applied to single precision or double precision floating point numbers, to fixed point numbers or complex numbers, to serial or vectorised signals. Static typing means that the `Haskell` compiler can check that everything fits together when compiling a program or parts of it. Lazy evaluation means, that we can transform audio data, as it becomes available, while programming in a style, that treats those streams, as if they would be available at once.

The target of our embedded compiler is LLVM. It differs from Csound and SuperCollider in that LLVM is not a signal processing system. It is a high-level assembler and we have to write the core signal processing building blocks ourselves. However, once this is done, assembling those blocks is as simple as writing Csound orchestra files or SuperCollider SCLang programs. We could have chosen a concrete machine language as target, but LLVM does a much better job for us: It generates machine code for many different processors, thus it can be considered a portable assembler. It also supports the vector units of modern processors and target dependent instructions (*intrinsics*) and provides us with a large set of low-level to high-level optimisations, that we can even select and arrange individually. We can run LLVM code immediately from our `Haskell` programs (JIT), but we can also write LLVM bitcode files for debugging or external usage.

3 Implementation

We are now going to discuss the design of our implementation [Thielemann, 2010a].

3.1 Signal generator

In our design a signal is a sequence of sample values and a signal generator is a state transition system, that ships a single sample per request while updating the state. E.g. the state of an exponential curve is the current amplitude and on demand it returns the current amplitude as result while decreasing the amplitude state by a constant factor. In the same way an oscillator uses the phase as internal state. Per request it applies a wave function on the phase and delivers the resulting value as current sample. Additionally it increases the phase by the oscillator frequency and wraps around the result to the interval $[0, 1)$. This design is much inspired by [Coutts et al., 2007].

According to this model we define an LLVM signal generator in `Haskell` essentially as a pair of an initial state and a function, that returns a tuple containing a flag showing whether there are more samples to come, the generated sample and the updated state.

```
type Generator a = forall state.
   (state,
    state -> Code (V Bool, (a, state)))
```

Please note, that the actual type definition in the library is a bit different and much larger for technical reasons.

The lower-case identifiers are *type variables* that can be instantiated with actual types. The variable `a` is for the sample type and `state` for the internal state of the signal generator. Since `Generator` is not really a signal but a description for LLVM code, the sample type cannot be just a `Haskell` number type like `Float` or `Double`. Instead it must be the type for one of LLVM's virtual registers, namely `V Float` or `V Double`, respectively. The types `V` and `Code` are imported from a `Haskell` interface to LLVM [O'Sullivan and Augustsson, 2010]. Their real names are `Value` and `CodeGenFunction`, respectively.

The type parameter is not restricted in any way, thus we can implement a generator of type `Generator (V Float, V Float)` for a stereo signal generator or `Generator (V Bool, V Float)` for a gate signal and a continuous signal that are generated synchronously. We do not worry about a layout in memory of an according signal at this point, since it may be just an interim signal that is never written to memory. E.g. the latter of the two types just says, that the generated samples for every call to the generator can be found in two virtual registers, where one register holds a boolean and the other one a floating point number.

We like to complement this general description with the simple example of an exponential curve generator.

```
exponential ::
   Float -> Float -> Generator (V Float)
exponential halfLife amp =
   (valueOf amp,
    \y0 -> do
      y1 <- mul y0 (valueOf
                      (2**(-1/halfLife)))
      return (valueOf True, (y0, y1)))
```

For simplification we use the fixed type `Float` but in the real implementation the type is flexible. The implementation is the same, only the real type of `exponential` is considerably more complicated because of many constraints to the type parameters.

The function `valueOf` makes a `Haskell` value available as constant in LLVM code. Thus the power computation with `**` in the `mul` instruction is done by `Haskell` and then implanted into the LLVM code. This also implies that the power is computed only once. The whole transition function, that is the second element of the pair, is a *lambda expression*, also known as *anonymous function*. It starts with a backslash and its argument `y0`, which identifies the virtual register, that holds the current internal state. It returns always `True` because the curve never terminates and it returns the current amplitude `y0` as current sample and the updated amplitude computed by a multiplication to be found in the register identified by `y1`.

We have seen, how basic signal generators work, however, signal processing consists largely of transforming signals. In our framework a signal transformation is actually a generator transformation. That is, we take apart given generators and build something new from them. For example the controlled amplifier dissects the envelope generator and the input generator and assembles a generator for the amplified signal.

```
amplify ::
   Generator (V Float) ->
   Generator (V Float) ->
   Generator (V Float)
amplify (envInit, envTrans)
        (inInit,  inTrans) =
   ((envInit, inInit),
    (\(e0,i0) -> do
       (eCont,(ev,e1)) <- envTrans e0
       (iCont,(iv,i1)) <- inTrans  i0
       y    <- mul ev iv
       cont <- and eCont iCont
       return (cont, (y, (e1,i1)))))
```

So far our signals only exist as LLVM code, but computing actual data is straightforward:

```
render ::
   Generator (V Float) ->
   V Word32 -> V (Ptr Float) ->
   Code (V Word32)
render (start, next) size ptr = do
   (pos,_) <- arrayLoop size ptr start $
      \ ptri s0 -> do
         (cont,(y,s1)) <- next s0
         ifThen cont () (store y ptri)
         return (cont, s1)
   ret pos
```

The ugly branching that is typical for assembly languages including that of LLVM is hidden in our custom functions `arrayLoop` and `ifThen`. `Haskell` makes a nice job as macro assembler. Again, we only present the most simple case here. The alternative to filling a single buffer with signal data is to fill a sequence of chunks, that are created on demand. This is called *lazy evaluation* and one of the key features of `Haskell`.

At this point, we might wonder, whether the presented model of signal generators is general enough to match all kinds of signals, that can appear in real applications. The answer is “yes”, since given a signal there is a generator that emits that signal. We simply write the signal to a buffer and then use a signal generator, that manages a pointer into this buffer as internal state. This generator has a real-world use when reading a signal from a file. We see that our model of signal generators does not impose a restriction on the kind of signals, but it well restricts the access to the generated data: We can only traverse from the beginning to the end of the signal without skipping any value. This is however intended, since we want to play the signals in real-time.

3.2 Causal Processes

While the above approach of treating signal transformations as signal generator transformations is very general, it can be inefficient. For example, for a signal generator `x` the expression `mix x x` does not mean that the signal represented by `x` is computed once and then mixed with itself. Instead, the mixer runs the signal generator `x` twice and adds the results of both instances. I like to call that the *sharing problem*. It is inherent to all DSLs that are embedded into a purely functional language, since in those languages objects have no identity, i.e. you cannot obtain an object’s address in memory. The sharing problem also occurs, if we process the components of a multi-output signal process individually, for instance the channels of a stereo signal or the lowpass, bandpass, highpass components of a state variable filter. E.g. for delaying the right channel of a stereo signal we have to write `stereo (left x) (delay (right x))` and we run into the sharing problem, again.

We see two ways out: The first one is relying on LLVM’s optimiser to remove the duplicate code. However this may fail since LLVM cannot remove duplicate code if it relies on seemingly independent states, on interaction with memory or even on interaction with the outside world. Another drawback is that the temporarily generated code may grow exponentially compared to the code written by the user. E.g. in

```
let y = mix x x
    z = mix y y
in  mix z z
```

the generator `x` is run eight times.

The second way out is to store the results of a generator and share the storage amongst all users of the generator. We can do this by rendering the signal to a lazy list, or preferably to a lazily generated list of chunks for higher performance. This approach is a solution to the general case and it would also work if there are signal processes involved that shrink the time line, like in `mix x (timeShrink x)`.

While this works in the general case, there are many cases where it is not satisfying. Especially in the example `mix x x` we do not really need to store the result of `x` anywhere, since it is consumed immediately by the mixer. Storing the result is at least inefficient in case of a plain `Haskell` singly linked list and even introduces higher latency in case of a chunk list.

So what is the key difference between `mix x x` and `mix x (timeShrink x)`? It is certainly, that in the first case data is processed in a synchronous way. Thus it can be consumed (mixed) as it is produced (generated by `x`). However, the approach of signal transformation by signal generator transformation cannot model this behaviour. When considering the expression `mix x (f x)` we have no idea whether `f` maintains the "speed" of its argument generator. That is, we need a way to express that `f` emits data synchronously to its input. For instance we could define

```
type Map a b = a -> Code b
```

that represents a signal transformation of type `Generator a -> Generator b`. It could be applied to a signal generator by a function `apply` with type

```
Map a b -> Generator a -> Generator b
```

and where we would have written `f x` before, we would write `apply f x` instead.

It turns out that `Map` is too restrictive. Our signal process would stay synchronous if we allow a running state as in a recursive filter and if we allow termination of the signal process before the end of the input signal as in the `Haskell` list function `take`. Thus, what we actually use, is a definition that boils down to

```
type Causal a b = forall state.
   (state, (a, state) ->
           Code (V Bool, (b, state)))  .
```

With this type we can model all kinds of causal processes, that is, processes where every output sample depends exclusively on the current and past input samples. The `take` function may serve as an example for a causal process with termination.

```
take :: Int -> Causal a a
take n =
   (valueOf n,
    \(a,toDo) -> do
      cont <- icmp IntULT (valueOf 0) toDo
      stillToDo <- sub toDo (valueOf 1)
      return (cont, (a, stillToDo)))
```

The function `apply` for applying a causal process to a signal generator has the signature

```
apply :: Causal a b ->
             Generator a -> Generator b
```

and its implementation is straightforward. The function is necessary to do something useful with causal processes, but it loses the causality property. For sharing we want to make use of facts like that the serial composition of causal processes is causal, too, but if we have to express the serial composition of processes `f` and `g` by `apply f (apply g x)`, then we cannot make use of such laws. The solution is to combine processes with processes rather than transformations with signals. E.g. with `>>>` denoting the serial composition we can state that `g >>> f` is a causal process.

In the base `Haskell` libraries there is already the `Arrow` abstraction, that was developed for the design of integrated circuits in the Lava project, but it proved to be useful for many other applications. The `Arrow` type class provides a generalisation of plain `Haskell` functions. For making `Causal` an instance of `Arrow` we must provide the following minimal set of methods and warrant the validity of the arrow laws [Hughes, 2000].

```
arr :: (a -> b) -> Causal a b
(>>>) :: Causal a b ->
              Causal b c -> Causal a c
first :: Causal a b -> Causal(a,c)(b,c)
```

The infix operator `>>>` implements (serial) function composition, the function `first` allows for parallel composition, and the function `arr` generates stateless transformations including rearrangement of tuples as needed by `first`. It turns out, that all of these *combinators* maintain causality. They allow us to express all kinds of causal processes without feedback. If `f` and `mix` are causal processes, then we can translate the former `mix x (f x)` to

```
arr (\x -> (x,x)) >>> second f >>> mix
where second p = swap >>> p >>> swap
      swap = arr (\(a,b) -> (b,a))  .
```

For implementation of feedback we need only one other combinator, namely `loop`.

```
loop ::
  c -> Causal (a,c) (b,c) -> Causal a b
```

The function `loop` feeds the output of type `c` of a process back to its input channel of the same type. In contrast to the `loop` method of the standard `ArrowLoop` class we must delay the value by one sample and thus need an initial value of type `c` for the feedback signal. Because of the way, `loop` is designed, it cannot run into deadlocks. In general deadlocks can occur whenever a signal processor runs ahead of time, that is, it requires future input data in order to compute current output data. Our notion of a causal process excludes this danger.

In fact, feedback can be considered another instance of the sharing problem and `loop` is its solution. For instance, if we want to compute a comb filter for input signal `x` and output signal `y`, then the most elegant solution in `Haskell` is to represent `x` and `y` by lists and write the equation `let y = x + delay y in y` which can be solved lazily by the `Haskell` runtime system. In contrast to that if `x` and `y` are signal generators, this would mean to produce infinitely large code since it holds

```
y = x + delay y
  = x + delay (x + delay y)
  = x + delay (x + delay (x + delay y))
 ...
```

With `loop` however we can share the output signal `y` with its occurrences on the right hand side. Therefore, the code would be

```
y = apply (mixFanout >>> second delay) x
       where mixFanout =
          arr (\(a,b) -> (a+b,a+b))  .
```

Since the use of arrow combinators is somehow less intuitive than regular function application and `Haskell`'s recursive `let` syntax, there is a preprocessor that translates a special arrow syntax into the above combinators. Further on there is a nice abstraction of causal processes, namely commutative causal arrows [Liu et al., 2009].

We like to note that we can even express signal processes that are causal with respect to one input and non-causal with respect to another one. E.g. frequency modulation is causal with respect to the frequency control but non-causal with respect to the input signal. This can be expressed by the type

```
freqMod :: Generator (V a) ->
           Causal (V a) (V a)  .
```

In retrospect, our causal process data type looks very much like the signal generator type. It just adds a parameter to the transition function. Vice versa the signal generator data type could be replaced by a causal process with no input channel. We could express this by

```
type Generator a = Causal () a
```

where `()` is a nullary tuple. However for clarity reasons we keep `Generator` and `Causal` apart.

3.3 Internal parameters

It is a common problem in signal processing that recursive filters [Hamming, 1989] are cheap in execution, but computation of their internal parameters (mainly feedback coefficients) is expensive. A popular solution to this problem is to compute the filter parameters at a lower sampling rate [Vercoe, 2009; McCartney, 1996]. Usually, the filter implementations hide the existence of internal parameters and thus they have to cope with the different sampling rates themselves.

In this project we choose a more modular way. We make the filter parameters explicit but opaque and split the filtering process into generation of filter parameters, filter parameter resampling and actual filtering. Static typing asserts that filter parameters can only be used with the respective filters.

This approach has several advantages:

- A filter only has to treat inputs of the same sampling rate. We do not have to duplicate the code for coping with input at rates different from the sample rate.
- We can provide different ways of specifying filter parameters, e.g. the resonance of a lowpass filter can be controlled either by the slope or by the amplification of the resonant frequency.
- We can use different control rates in the same program.
- We can even adapt the speed of filter parameter generation to the speed of changes in the control signal.
- For a sinusoidal controlled filter sweep we can setup a table of filter parameters for logarithmically equally spaced cutoff frequencies and traverse this table at varying rates according to arcus sine.
- Classical handling of control rate filter parameter computation can be considered as

resampling of filter parameters with constant interpolation. If there is only a small number of internal filter parameters, then we can resample with linear interpolation of the filter parameters.

The disadvantage of our approach is that we cannot write something simple like `lowpass (sine controlRate) (input sampleRate)` anymore, but with `Haskell`'s *type class* mechanism we let the `Haskell` compiler choose the right filter for a filter parameter type and thus come close to the above concise expression.

4 Related Work

Our goal is to make use of the elegance of `Haskell` programming for signal processing. Our work is driven by the experience, that today compiled `Haskell` code cannot compete with traditional signal processing packages written in C. There has been a lot of progress in recent years, most notably the improved support for arrays without overhead, the elimination of temporary arrays (*fusion*) and the Data-Parallel Haskell project that aims at utilising multiple cores of modern processors for array oriented data processing. However there is still a considerable gap in performance between idiomatic `Haskell` code and idiomatic C code. A recent development is an LLVM-backend for the Glasgow Haskell Compiler (GHC), that adds all of the low-level optimisations of LLVM to GHC. However we still need some tuning of the high-level optimisation and a support for processor vector types in order to catch up with our EDSL method.

In Section 2 we gave some general thoughts about possible designs of signal processing languages. Actually for many combinations of features we find instances: The two well-established packages Csound [Vercoe, 2009] and SuperCollider [McCartney, 1996] are domain specific untyped languages that process data in a chunky manner. This implies that they have no problem with sharing signals between signal processors, but they support feedback with short delay only by small buffers (slow) or by custom plugins (more development effort). Both packages support three rates: note rate, control rate and sample rate in order to reduce expensive computations of internal (filter) parameters. With the `Haskell` wrappers [Hudak et al., 1996; Drape, 2009] it is already possible to control these programs as if they were part of `Haskell`, but it is not possible to exchange audio streams with them in real-time. This shortcoming is resolved with our approach.

Another special purpose language is ChucK [Wang and Cook, 2004]. Distinguishing features of ChucK are the generalisation to many different rates and the possibility of programming while the program is running, that is while the sound is playing. As explained in Section 3.3 we can already cope with control signals at different rates, however the management of sample rates at all could be better if it was integrated in our framework for physical dimensions. Since the `Haskell` systems Hugs and GHC both have a fine interactive mode, `Haskell` can in principle also be used for live coding. However it still requires better support by LLVM (shared libraries) and by our implementation.

Efficient short-delay feedback written in a declarative manner can probably only be achieved by compiling signal processes to a machine loop. This is the approach implemented by the Structured Audio Orchestra Language of MPEG-4 [Scheirer, 1999] and Faust [Orlarey et al., 2004]. Faust started as compiler to the C++ programming language, but it does now also support LLVM. Its block diagram model very much resembles `Haskell`'s arrows (Section 3.2). A difference is, that Faust's combinators contain more automatisms, which on the one hand simplifies binding of signal processors and on the other hand means, that errors in connections cannot be spotted locally.

Before our project the compiling approach embedded in a general purpose language was chosen by Common Lisp Music [Schottstaedt, 2009], Lua-AV [Smith and Wakefield, 2007], and Feldspar (`Haskell`) [programming group at Chalmers University of Technology, 2009].

Of all listed languages only ChucK and `Haskell` are strongly and statically typed, and thus provide an extra layer of safety. We like to count Faust as being weakly typed, since it provides only one integer and one floating point type.

5 Conclusions and further work

The speed of our generated code is excellent, yet the generating `Haskell` code looks idiomatic. The next step is the integration of the current low-level implementation into our existing framework for signal processing, that works with real physical quantities and statically checked physical dimensions. There is also a lot of room for automated optimisations by

GHC rules, be it for vectorisation or for reduction of redundant computations of frac.

6 Acknowledgments

I like to thank the careful proofreaders of the draft for their valuable suggestions.

References

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, apr.

Rohan Drape. 2009. hsc3: Haskell SuperCollider. `http://hackage.haskell.org/package/hsc3-0.7`, June.

James Elliott. 2004. *Hibernate: a developer's notebook*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA.

Richard W. Hamming. 1989. *Digital Filters*. Signal Processing Series. Prentice Hall, January.

Paul Hudak, T. Makucevich, S. Gadde, and B. Whong. 1996. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3), June.

John Hughes. 2000. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May.

Peter J. Landin. 1966. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March.

Hai Liu, Eric Cheng, and Paul Hudak. 2009. Causal commutative arrows and their optimization. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 35–46, New York, NY, USA. ACM.

James McCartney. 1996. Super Collider. `http://www.audiosynth.com/`, March.

Yann Orlarey, Dominique Fober, and Stephane Letz. 2004. Syntactical and semantical aspects of Faust. In *Soft Computing*.

Bryan O'Sullivan and Lennart Augustsson. 2010. llvm: Bindings to the LLVM compiler toolkit. `http://hackage.haskell.org/package/llvm-0.9.0.1`, November.

Simon Peyton Jones. 1998. Haskell 98 language and libraries, the revised report. `http://www.haskell.org/definition/`.

Functional programming group at Chalmers University of Technology. 2009. feldspar-language: A functional embedded language for DSP and parallelism. `http://hackage.haskell.org/package/feldspar-language-0.1`, November.

Eric Scheirer. 1999. Iso/iec 14496-3:1999: Information technology – coding of audio-visual objects – part 3: Audio – subpart 5: Structured audio orchestra language. Technical report, International Organization of Standardization.

Bill Schottstaedt. 2009. Common lisp music. `http://ccrma.stanford.edu/software/clm/`.

Wesley Smith and Graham Wakefield. 2007. Real-time multimedia composition using lua. In *Proceedings of the Digital Art Weeks 2007*. ETH Zurich.

Henning Thielemann. 2004. Audio processing using Haskell. In Gianpaolo Evangelista and Italo Testa, editors, *DAFx: Conference on Digital Audio Effects*, pages 201–206. Federico II University of Naples, Italy, October.

Henning Thielemann. 2010a. Audio signal processing embedded in Haskell via LLVM: Darcs repository. `http://code.haskell.org/synthesizer/llvm/`.

Henning Thielemann. 2010b. Compiling Signal Processing Code embedded in Haskell via LLVM. `http://arxiv.org/abs/1004.4796`.

Barry Vercoe. 2009. CSound. `http://www.csounds.com/`.

Ge Wang and Perry Cook. 2004. Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 812–815, New York, NY, USA. ACM.