Audio Processing using Haskell

Henning Thielemann

Center of Industrial Mathematics
University of Bremen, Bremen, Germany
thielema@math.uni-bremen.de

ABSTRACT

The software for most today's applications including signal processing applications is written in imperative languages. Imperative programs are fast because they are designed close to the architecture of the widespread computers, but they don't match the structure of signal processing very well. In contrast to that, functional programming and especially lazy evaluation perfectly models many common operations on signals.

`Haskell` is a statically typed, lazy functional programming language which allow for a very elegant and concise programming style. We want to sketch how to process signals, how to improve safety by the use of physical units, and how to compose music using this language.

1. INTRODUCTION

Imperative programming languages are the usual choice for today's software. The currently popular CPUs conform to the imperative programming paradigm and allow a fast execution of imperative programs.

Nevertheless functional programming languages like `Haskell` [1, 2] became valuable alternatives in the recent past. The term *Functional Programming* [3, 4] denotes a kind of program flow that is different from the *imperative* one. The program flow is independent from a particular type system and from whether programs can be compiled or not. In fact `Haskell` can be both interpreted [5] and compiled [6].

Today functional conceptions are integrated into almost every imperative programming language. They allow for structured and safe programming. A function is a part of the program with some declared input and output. Most imperative languages allow for bypassing the input/output interface by the use of global variables and by manipulation (update) of input objects. In contrast to that `Haskell` disallows or at least strongly discourages that. This makes things more deterministic: If you apply a function to the same argument values it will always result in the same value. Because of this strong determination `Haskell` is suited for interactive computations (see the interactive mode of Glasgow `Haskell`[6]) but you do not have to resign static type safety as in scripting languages.

The functional approach allows features that can be imagined hardly for imperative programming languages:

Lazy evaluation means that function arguments and parts of data structures are only computed if they are needed. Thus a list may contain infinitely many elements. This poses no problem since a terminating algorithm will be able to access only finitely many of them. If you represent a certain signal by a list you don't need to worry about its length – make it infinite and it will be constructed only as far as needed for the final application.

The functional approach allows for working with functions like with other kind of data. Functions can be argument and value of other functions, so called higher-order functions. Thus, loop structures need not to be hard-wired into the language but the user can create loop structures as higher-order functions that take the loop body as argument. Now you have the combinatorial power of functions that traverse through a data structure and functions that process atomic data.

This allows for a very compact notation which reduces the need for specialised library functions or library functions that do very different things depending on type and number of the passed arguments. By the way, the latter would conflict with `Haskell`'s *static typing* and the *partial application* of functions.

This article shows how the features of functional programming and especially that of `Haskell` permit elegant programming of audio processing algorithms.

The second section describes some of the basics of `Haskell`, e.g. syntax, type variables, data structures. The third section describes some routines for plain signal processing. The fourth section sketches the design of computations with physical quantities which improves the safety on using signal processing routines. The fifth section presents `Haskore`, a system for programming music, where music can be rendered into an audio or MIDI stream eventually.

2. `Haskell` BASICS

To become familiar with `Haskell` let's get some impressions of its syntax:

```
> zero :: Int
> zero = 0
```

The first line is the *signature* of the constant `zero`. It declares the *type* `Int` for that constant, where `Int` denotes machine size integers. The second line defines the value of the constant.

```
> doubleInt :: Int -> Int
> doubleInt x = x+x
```

The *type* `Int -> Int` states that `doubleInt` is a function that maps an integer to an integer. This can be more generally formulated for any numeric type:

```
> double :: Num a => a -> a
> double x = x+x
```

The identifier `Num` is a *type class* and `a` is a *type variable*. The expression `Num a` is the context of the type declaration and is separated from it by `=>`. The signature tells us that `double` can be used for all types `a` that support certain numerical operations (to be more precise: operations of algebraic rings). If you omit the type context (i.e. `double :: a -> a`) the compiler would refuse to compile because in this example it can't assert that the type `a` supports the `+` operator. This is a bit more structured than the overloading of operator symbols in C++.

How can one define functions with more than one argument? Strictly speaking it is not possible, but it can be simulated easily.

```
> add :: Num a => a -> a -> a
> add x y = x+y
```

The signature may look strange but it is pure logic: The arrow `->` is right associative. That is, the signature of `add` is equivalent to `a -> (a -> a)`. That in turn means: If `add` is applied to only one value the result is a function of one argument. E.g. the result of `(add 1)` is the increment function. Consequently, `((add 1) 2)` is a constant with value `3`. Since function application is left-associative the above example can be abbreviated to `add 1 2`. A function application that doesn't result in a constant is called *partial application*.

Partial applications are no magic. Essentially they defer computation until all function arguments are known. They are one of the reasons of `Haskell`'s concise programming style. Usually you will note that in subsequent applications of a function some arguments receive the same value all the time and others receive always different values. You should sort the arguments in the function definition according to increasing expected variability.

Data structures also follow the philosophy of deferred computation. Here the term is *lazy evaluation*. A list structure is defined roughly this way:

```
data List a = Empty | Prepend a (List a)
```

This is a recursive definition of a singly linked list meaning: A `List` over type `a` is either `Empty` or a list of type `a` with a single element of type `a` prepended. The identifiers `Empty` and `Prepend` are called *constructors*. The list of the numbers $1, 2, 3$ would be written as

```
Prepend 1 (Prepend 2 (Prepend 3 Empty))   .
```

The designers of `Haskell` decided to use `[]` instead of `Empty` and the infix constructor `:` instead of the prefix `Prepend`. Thus a standard `Haskell` list can be written as `1:(2:(3:[]))` or even shorter `1:2:3:[]`, since `:` is right-associative. The common notation `[1,2,3]` is available, too. Finally, the syntax of the list type is `[a]` instead of `List a`.

With *lazy evaluation* we can process even infinite data structures. Let's have a look at some infinite list like `repeat 0` which is a list consisting of infinitely many zeros. When the `Haskell` interpreter is asked to print the list it will actually start printing – but it will never stop. Here is another example that may convince you that *lazy evaluation* works:

```
> infiniteLoop :: Integer
> infiniteLoop = 1 + infiniteLoop
>
> bypassInfinity :: [Integer]
> bypassInfinity = tail [infiniteLoop, 1, 2]
```

Calling `infiniteLoop` leads to an infinite loop. The function `tail` removes the first element from a list. Since it ignores the value of the first element it isn't even computed and thus `bypassInfinity` results in `[1,2]` rather than an infinite loop.

A language using *strict evaluation* would compute the list `[infiniteLoop, 1, 2]` including its elements first and then it would apply `tail` to the result. *Lazy evaluation* works different: When you call `bypassInfinity` the run-time system starts thinking about how to obtain the first element of the resulting list. It will encounter that due to `tail` the first element of the original list is not required.

3. SIGNAL PROCESSING

There is already a library [7] containing many routines related to signal processing. For now let's start with some simple examples. How to superpose two signals represented by lists?

```
> superpose :: Num a => [a] -> [a] -> [a]
> superpose = zipWith (+)
```

When implementing `superpose` we omitted the arguments. This is a short notation that means roughly that a function is expressed in terms of another function. Here we use the function `zipWith` *partially applied* to the operation `+`. The function `zipWith` applies an operation to the corresponding elements of two lists. The expression `(+)` denotes the binary addition operator. The functional definition can be interpreted as: "Every occurence of `superpose x y` is expanded to `zipWith (+) x y`."

The function `zipWith` is a function from the standard `Haskell` library "Prelude" as most other functions presented here. It can be used analogously for the amplification of a signal with an arbitrary envelope: `zipWith (*)`.

The static amplification is also no problem:

```
> amplify :: Num a => a -> [a] -> [a]
> amplify v = map (v*)
```

The function `map` applies an operation to each element of the list. The notation `v*` denotes the infix operator `*` applied to only one argument, i.e. while `*` is a function with two arguments, in `v*` the first argument is fixed and thus it denotes a function with only one argument.

The function `iterate` from the standard library creates a list from interim results of an iteration:

```
> exponential :: Num a => a -> [a]
> exponential decay = iterate (decay*) 1
```

The function `foldl` accumulates values from a list using an arbitrary accumulation function. Thus the standard library defines `sum` as `foldl (+) 0` and `maximum` similarly. We can use it to determine several volume measures of a finite signal:

```
> amplitude :: (Num a, Ord a) => [a] -> a
> amplitude x = foldl max 0 (map abs x)

> euclideannorm :: Floating a => [a] -> a
> euclideannorm x = sqrt (sum (map (^2) x))
```

The function `foldl1` is similar but needs no initial value for the accumulator and requires a non-empty list. The accumulator doesn't need to be a scalar value, any other type is also fine. That allows for a compact definition of the superposition of an arbitrary number of signals:

```
> superposeMulti :: Num a => [[a]] -> [a]
> superposeMulti = foldl1 superpose
```

We defined `superposeMulti` according to its *meaning*, i.e. the first signal is superposed with the second one, then with the third one and so on. But the actual *execution* is very different: If you compute the result you will start asking for the first sample of the signal, which in turn requires `Haskell` to evaluate the first samples of the input signals. Accordingly, subsequent samples of the output are computed.

Now we know enough about `Haskell` to create a first simple instrument sound:

```
> {- an oscillator
>    with 'freq' waves per sample -}
> oscillator :: Floating a => a -> [a]
> oscillator freq =
>    map sin (iterate (2*pi*freq +) 0)

> {- a ping sound is a sine oscillator
>    enveloped by an exponential -}
> ping :: Floating a => a -> a -> [a]
> ping decay freq =
>    zipWith (*) (exponential decay)
>                (oscillator freq)
```

We have realised the power of general list functions such as `iterate`, `map`, `zipWith`, `foldl`. Note that we never cared about indices! These functions operate on lists in quite a linear manner. What about signal processing including feedback?

`Haskell`'s answer to feedback is *recursion*:

```
> echo :: Num a => Int -> a -> [a] -> [a]
> echo time gain x =
>    let y = superpose x (delay time
>                            (amplify gain y))
>    in  y
>
> delay :: Num a => Int -> [a] -> [a]
> delay time = (replicate time 0  ++)
```

For more clarity the `let` notation is used. It introduces a local *identifier* for some value and in this case it is used for recursively defining `y`.

The definition should be read as: An infinite echo is a superposition of the original signal and an attenuated and delayed version of itself. This is a kind of recursion which describes the data structure recursively. It relies on the lazy evaluation of data and is very common in `Haskell` though it is quite uncommon in other languages.

Instead of simply attenuating the signal on feedback any other processing can be applied, say a lowpass or highpass filter:

```
> echoProc :: Num a =>
>    Int -> ([a] -> [a]) -> [a] -> [a]
> echoProc time feedback x =
>    let y = superpose x (delay time
>                                (feedback y))
>    in  y
```

By defining lists recursively we can write the solution of difference equations with the common notation of differential equations. (cf. [8]) The standard function `scanl` accumulates values from a list using an arbitrary accumulation operation, but in contrast to `foldl` it returns a list of the intermediate results. Thus the definition

```
> integrate :: Num a => a -> [a] -> [a]
> integrate = scanl (+)
```

is straight-forward. The second argument of `scanl`, which is the initial value of the accumulator, turns into the first argument of `integrate` and represents the integration constant. Using it we can numerically solve the inhomogeneous oscillation equation $y'' + c_1 y' + c_0 y = u$ with the driving force u and the initial values $y(0)$ and $y'(0)$.

```
> osciODE :: Num a =>
>    (a,a) -> (a,a) -> [a] -> [a]
> osciODE (c0,y0) (c1,y'0) u =
>    let infixl 6 .+, .-
>        infixr 7 *>
>        (.+) = zipWith (+)
>        (.-) = zipWith (-)
>        (*>) = amplify
>
>        y   = integrate y0  y'
>        y'  = integrate y'0 y''
>        y'' = u .- (c0 *> y .+ c1 *> y')
>    in  y
```

Note that the apostrophe has no special meaning and is part of the identifiers. The infix operators `.+`, `.-`, `*>` are introduced just for visual convenience.

At the first glance this example looks a bit like magic. It's not obvious how the program actually solves the equation but you can verify that it computes something (i.e. there is an order of computation such that each item of the lists `y`, `y'`, `y''` depends only on values that are already computed) and that the computed signal satisfies the difference equation. This example shows how lazy evaluation allows you to concentrate on *problems* rather than *solutions*.

Recursive filters (notion taken from [9] instead of IIR) could be implemented either as solution of difference equations or, in the tradition of imperative languages, using states. That is, the sample values of the signal are not processed independently but while scanning the signal an internal state is stored and updated. The functional programming paradigm forbids update operations. They must be implemented by inputting the current state and returning the updated state. The compiler is responsible to turn this back into update operations if possible.

We like to demonstrate this technique for a first order lowpass filter:

```
> lowpass1Aupdate :: Num a =>
>    a -> a -> a -> (a,a)
> lowpass1Aupdate k u0 y1 =
>    let y0 = u0+k*(y1-u0) in (y0,y0)
>
> lowpass1A :: Num a => a -> a -> [a] -> [a]
> lowpass1A s k (u:us) =
>    let (x,news) = lowpass1Aupdate k u s
>    in  x : lowpass1A news k us
```

Here `lowpass1Aupdate` takes the filter feedback `k`, the current input signal value `u0`. The state is the previous output value `y1`. The function returns the new output value `y0` and the updated state which is `y0`, too. The function `lowpass1A` applies the filtering process to a signal. The call to `lowpass1Aupdate` can be easily replaced by a call to every state updating function of this type.

Note that constructors like `:` are sort of two-way: You can not only use the colon to prepend an element to a list but by using *pattern matching* in an argument like `(u:us)` you can also split a list into the head element `u` and the rest of the list `us`.

Because states are a common programming technique there is a data type `State`. An object of type `State s a` is essentially a function with signature `s -> (a, s)`, i.e. a function that receives the current state and outputs some data and the updated state. Since functions can be easily constructed on the fly (in fact that is the way multi-argument functions are implemented) it is also possible to feed the update function with additional data.

```
> lowpass1Bupdate :: Num a =>
>    a -> a -> State a a
> lowpass1Bupdate k u0 =
>    let update y1 =
>           let y0 = u0+k*(y1-u0) in (y0,y0)
>    in  State update
>
> lowpass1B :: Num a => a -> a -> [a] -> [a]
> lowpass1B s k u = evalState
>    (mapM (lowpass1Bupdate k) u) s
```

The expression `lowpass1Bupdate k` is of type `a -> State a a`, i.e. a function that maps an input value to a state update function. The function `mapM` applies this map to each input value and glues together the resulting update functions. Eventually `evalState` executes the actions beginning with state `s`.

4. PHYSICAL UNITS

The key tool to describe natural sound phenomena is physics. So it is an obvious question if one can use physical units rather than scalar values in `Haskell`. Physical units provide more details and allow for more consistency checks. Certainly one can argue that units are for physics what types are for informatics.

Imagine you want to simulate an echo where the sound has to cover a distance s for returning by the acoustic velocity v sampled at a rate of r. The number n of samples for the delay can be obtain from

$$\begin{aligned} s &= 100\text{ m} \\ v &= 330\text{ m/s} \\ r &= 44100\text{ samples/s} \\ n &= \frac{s\cdot r}{v} \\ &\approx 13363\text{ samples} \end{aligned}$$

and the correct unit of n verifies that our computation was not totally wrong.

Because of `Haskell`'s polymorphic type system numbers equipped with physical units can be nicely integrated into the collection of numeric types [10]. The *type classes* of `Haskell` allow the usage of infix operators like `+` and `*` for custom types. Though it should be mentioned that infix operators are pure syntactic sugar making computer formulas similar to mathematical notation. The price to be paid are lots of precedence and associativity rules, more complicated syntax checking and more difficulties in understanding syntax error messages.

As in most other languages it is not possible to generate custom compiler errors. E.g. a comparison like `'a' < 1` is rejected by the compiler due to the type mismatch. But the compiler can't be advised to reject expressions like $1\text{ m} < 2\text{ s}$. By unfolding the function calls the compiler may even realize that the expression will always result in an error but it will translate it into a permanent runtime error rather than a compilation error.

So, what's a physical quantity? A physical quantity is essentially a number equipped with a vector of the exponents of some base units. E.g. a force of 14 N can be expressed by 14 and the vector $(1, -2, 1, 0)$, where the vector contains the exponents of meter, second, kilogramme and coulomb respectively.

To stay independent from a specific unit system we define

```
> type Unit i = FiniteMap i Int
```

where the standard data type `FiniteMap` represents a dictionary with keys of type `i` and values of type `Int`. `FiniteMap` perfectly reflects the sparse structure of the exponent vector though it might seem to be somewhat overkill. For specific unit systems like the one of SI [11] one would choose the type `i` to be `Int` or some enumeration. Now one can define some operations on the exponent vectors like add, subtract.

The next step is to combine a numerical value with a unit:

```
> data PhysValue i a = PV a (Unit i)
```

It means that an object of type `(PhysValue i a)` is composed of a number of type `a` and a unit exponent vector of type `Unit i`. As an example let's look at the definition of the equality relation == for this type:

```
> instance (Eq i, Eq a) =>
>          Eq (PhysValue i a) where
>    (PV x xu) == (PV y yu) = x==y && xu==yu
```

This reads as: If both the type `i` and the type `a` are comparable then physical quantities constructed from them are comparable, as well. Two physical quantities are equal if and only if the numerical value and the unit matches.

The next step is to provide a type for a specific unit system. Here we gear towards the SI system of units. We define

```
> data SIDim =
>         Length | Time | Mass | Charge |
>         Angle | Temperature | Information
>      deriving (Eq, Ord, Enum, Show)
```

and then `Unit SIDim` is the type that represents the composed units in the SI system. Further on we require a set of constants for prefixes like `kilo`, `milli`, a set of basic units like `meter`, `second`, some physical constants like `mach` (sonic velocity).

This would be enough for plain computation but it is more convenient if physical values could be also converted to strings. Decomposing a unit into common SI units requires some heuristics but it can be done in a satisfactory manner. With such a system interactive computations with physical quantities look like

```
GHCi> 2*milli*meter
2.0*mm
GHCi> 1000*liter
1.0*m^3
GHCi> 10*watt/(220*volt)
45.45454545454545*mA
GHCi> 9*newton*meter/liter
9.0*kPa
GHCi> year/second
3.1556926080000002e7
GHCi> 100*meter * 44100/second / mach
13283.132530120482
```

Note that in contrast to other languages there is no implicit conversion of different types of numbers at runtime. The literal `2` is a polymorphic constant of a type of class `Num`. In `Haskell` syntax this would be written as

```
2 :: Num a => a   .
```

This is similar for `milli`:

```
milli :: Fractional a => a
milli = 1e-3   .
```

That is since in the expression `2*milli*meter` the generic constants `2` and `milli` are mixed with the special SI quantity `meter` the types of the constants `2` and `milli` are also specialised to SI quantities at compile time. This mechanism is called *type inference*.

Finally, here is a front-end to the first order lowpass filter using physical units:

```
> lowpass1Unit ::
>    (Ord i, Show i, Floating a) =>
>       PhysValue i a -> PhysValue i a ->
>         [a] -> [a]
> lowpass1Unit samplerate cutoff =
>    lowpass1B 0 (exp (-2*pi*
>             toScalar (cutoff/samplerate)))
```

The unit check is built into `toScalar` – In this simple implementation the program is aborted if `toScalar` is applied to a non-scalar value. Note that in general such front-ends work with every unit system independent from particular units. I.e. there must always be parameters that map physical quantities to the scalar parameters of the numeric computation. These coefficients must have the desired units. E.g. the amplitude of an oscillating voltage has unit `volt`, the sample rate of a time series has unit `1/second`, the sample rate of a function of the length has unit `1/meter`, the sample rate of a frequency spectrum has unit `second` and so on.

5. MUSIC COMPOSITION

In the past special purpose languages for composing music were developed [12]. `Haskell`'s syntax is so concise that there is hardly a need for a special markup language. The most famous approach for creating music with `Haskell` is `Haskore` [13, 14, 15]. `Haskore` turns `Haskell` into a fully programmable statically safe music description language. No extra interpreter is required.

`Haskore` is organised as follows:

1. The front-end is a data structure for abstractly describing music.
2. A *performer* function turns this data structure into a sequence of musical events.
3. Several back-ends exist that convert such a sequence into a MIDI stream [16], a CSound orchestra file [17] or into an audio stream.

This section provides some guidance on how to set up a `Music` data structure. Creating a piece of music at the level of the core data structure looks like

```
> cMajMelodic0, cMajHarmonic0 :: Music
>
> cMajMelodic0 =
>    Note (C,0) qn [] :+:
>    Note (E,0) qn [] :+:
>    Note (G,0) qn [] :+:
>    Note (C,1) qn []
>
> cMajHarmonic0 =
>    Note (C,0) qn [] :=:
>    Note (E,0) qn [] :=:
>    Note (G,0) qn [] :=:
>    Note (C,1) qn []
```

where `qn` denotes the duration of a quarter note and `[]` is an empty list that could be filled with additional note attributes.

Some assisting functions may simplify writing. The functions `c`, `d` and so on create a Note for given octave, duration, attributes.

```
> stdNote :: t -> (t -> [a] -> m) -> m
> stdNote dur n = n dur []
>
> cMajList :: [Music]
> cMajList =
>    map (stdNote qn) [c 0, e 0, g 0, c 1]
>
> cMajMelodic1, cMajHarmonic1 :: Music
> cMajMelodic1  = line  cMajList
> cMajHarmonic1 = chord cMajList
```

Now you can use all programming features for the creation of music. The simplest of them is probably an infinite melody loop:

```
> cMajLoop :: Music
> cMajLoop = repeatM cMajMelodic1
```

How about an infinite loop of notes that are randomly chosen from a given set of notes?

```
> randomChoiceLoop :: RandomGen g =>
>    [Pitch] -> Dur -> g -> Music
> randomChoiceLoop ps d g =
>    let indexToNote i = Note (ps!!i) d []
>    in  line (map indexToNote
>          (randomRs (0,(length ps)-1) g))
>
> cMajRandomLoop :: RandomGen g =>
>    g -> Music
> cMajRandomLoop = randomChoiceLoop
>    [(C,0), (E,0), (G,0), (C,1)] qn
```

The example uses the `ps!!i` operation which selects the `i`th element of the list `ps` and the function `randomRs` which generates an infinite list of random numbers of the specified range and the random number generator `g`.

Here is another example of programming music: We like to loop a pattern where the number of notes played increases over the time.

```
> partialBar ::
>    [Pitch] -> Dur -> Int -> Music
> partialBar ps d n =
>    let pitchToNote p = Note p d []
>    in  line
>         (map pitchToNote (take n ps)  ++
>          [Rest ((length ps - n)%1*d)])
>
> increasingLoop ::
>    Int -> [Pitch] -> Dur -> Music
> increasingLoop rep ps d =
>    let n = length ps
>    in  line (concatMap (replicate rep)
>          (map (partialBar ps d) [1..n]))
>
> cMajIncLoop :: Music
> cMajIncLoop = increasingLoop 4
>    [(C,0), (E,0), (G,0), (C,1)] qn
```

The last example is a bit less abstract: A function that computes a chord on a guitar for a chord given as list of tones. To each string of the guitar we assign the tone of the chord that is closest to the base tone of this string.

```
> choosePitchForString ::
>    [PitchClass] -> Pitch -> Pitch
> choosePitchForString chord str@(pc,oct) =
>    let diff x = mod
>          (pitchClass x - pitchClass pc) 12
>    in trans (minimum (map diff chord)) str
>
> guitarChord :: [PitchClass] -> [Pitch]
> guitarChord chord =
>    map (choosePitchForString chord)
>        [(E,2), (B,1), (G,1),
>         (D,1), (A,0), (E,0)]
```

The function `pitchClass` converts a pitch value like `C` into an integer and the function `trans` transposes an absolute pitch value.

6. OUTLOOK

`Haskell`'s strengths are the concise style of programming combined with static typechecking. `Haskell`'s weakness today is the low performance. On the one hand `Haskell` programs are a pleasure for optimisers because the compiler can clearly realise the flow of data. There are no hidden flows that can confuse the optimiser. On the other hand it's difficult to detect input/output values that can be turned into efficient update operations. The big flexibility makes it difficult to generate efficient code for a specific application.

Even today there are several possibilities for tuning `Haskell` programs for efficiency but the big challenge is to achieve both elegance and efficiency. Progresses in compiler technique combined with programmers assistance may make `Haskell` a very valuable tool for signal processing in future.

7. REFERENCES

[1] Simon Peyton Jones (editor), "Haskell 98 language and libraries, the revised report," `http://www.haskell.org/definition/`, 1998.

[2] Hal Daume, "Yet another haskell tutorial," `http://www.isi.edu/~hdaume/htut/`, 2004.

[3] John Hughes, "Why functional programming matters," `http://www.md.chalmers.se/~rjmh/Papers/whyfp.pdf`, 1984.

[4] Paul Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, 1989.

[5] "Hugs 98," `http://haskell.org/hugs/`, 2004.

[6] "Ghc: The glasgow haskell compiler," `http://haskell.org/ghc/`, 2004.

[7] Matthew Donadio, "Haskell dsp," `http://haskelldsp.sourceforge.net/`, 2003.

[8] Jerzy Karczmarczuk, "Lazy processing and optimization of discrete sequences," `http://users.info.unicaen.fr/~karczma/arpap/laseq.pdf`, 2004.

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

[10] Henning Thielemann, "Physical units in haskell," `http://www.math.uni-bremen.de/~thielema/Research/PhysicalUnit/`.

[11] "Si: International system of units," `http://en.wikipedia.org/wiki/SI`, March 2004.

[12] "Audio programming languages," `http://en.wikipedia.org/wiki/Category:Audio_programming_languages`, March 2004.

[13] Paul Hudak, "Haskore - music composition using haskell," `http://www.haskell.org/haskore/`, 2000.

[14] P. Hudak, T. Makucevich, S. Gadde, and B. Whong, "Haskore music notation – an algebra of music," *Journal of Functional Programming*, vol. 6, no. 3, June 1996.

[15] Paul Hudak, *The Haskell school of expression – Learning functional programming through multimedia*, Cambridge University Press, April 2000.

[16] MMA, "Midi 1.0 detailed specification: Document version 4.1.1," `http://www.midi.org/about-midi/specinfo.shtml`, February 1996.

[17] Barry Vercoe, "Csound," `http://www.bright.net/~dlphilp/linux_csound.html`.