Audio Processing and Sound Synthesis in Haskell

Eric Cheng and Paul Hudak

Yale University
Department of Computer Science
New Haven, CT 06520

eric.cheng@yale.edu, paul.hudak@yale.edu

January 2009

Research Report YALEU/DCS/RR-1405

1 Overview

Our goal is to develop a purely functional audio processing and sound synthesis framework using Haskell, that is more expressive than any existing framework, and that is at least as efficient.

Expressiveness and performance are of equal importance to us. Indeed, with modern-day processors, we believe that real-time performance can be achieved. A decade or two ago, the computational demand for real-time sound synthesis in software exceeded the capability of conventional processors. Software sound synthesis was usually done by using dedicated audio processing languages such as csound [Ver91] or written in low-level languages such as C. Sophisticated audio synthesis in real-time was often not feasible, and was usually done in hardware. Over the years, with the advent of ever faster processors and state-of-the-art compiler technologies for functional languages, real-time audio synthesis using a purely functional language has gradually become within reach.

A common source of inefficiency in functional programs is unexpected memory usage, otherwise known as *space leaks*. A space leak in an audio processing program could lead to stack overflow or huge memory consumption, neither of which is desirable. A program processing and generating real-time audio streams may be expected to run for a long time, if not indefinitely, and any significant space leak will soon exhaust the run-time memory. We would like to design a framework which gives the users the peace of mind that their programs will not exhibit unexpected space behaviors.

In this report, we first show an implementation of a sound synthesis framework based on arrows. Then we discuss the performance overhead caused by using arrows and possible ways of optimizing arrow-based programs. We also present our current implementation, which is based only on streams and built without arrows entirely.

Our code works best with the Glasgow Haskell Compiler (GHC) due to its support for a number of language extensions—such as arrows, multi-parameter type classes, and functional dependencies—as well as its optimization capabilities. Low-level interaction with the audio hardware is made possible by writing foreign function wrappers around the PortAudio [BB01] library.

2 Sound Synthesis Based on Arrows

We would like to provide a programming interface similar to digital circuits, where "unit generators" are modeled as signal transformers (that we prefer to call signal functions), and feedback loops route output signals of a component back to its input.

Since there is a strong correspondence between code written in terms of arrows and an actual circuit, we base our first implementation on arrows.

2.1 A Brief Introduction to Arrows

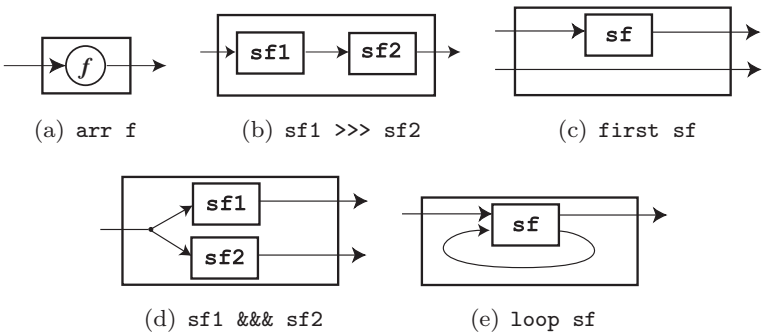
Arrows [Hug00] are a generalization to monads and provide a disciplined way to structure function compositions. Function composition must be done in a *point-free* manner in that the functions' input and output values are never directly manipulated. Values are passed around by using a set of combinators:

$$
\begin{array}{l}
\textbf{class } \mathit{Arrow}\ a \textbf{ where} \\
\quad \begin{array}{ll}
\mathit{arr} & :: (b \to c) \to a\ b\ c \\
(\ggg) & :: a\ b\ c \to a\ c\ d \to a\ b\ d \\
\mathit{first} & :: a\ b\ c \to a\ (b, d)\ (c, d) \\
\mathit{second} & :: a\ b\ c \to a\ (d, b)\ (d, c) \\
(***) & :: a\ b\ c \to a\ b'\ c' \to a\ (b, b')\ (c, c') \\
(\&\&\&) & :: a\ b\ c \to a\ b\ c' \to a\ b\ (c, c')
\end{array}
\end{array}
$$

arr lifts a pure function to an arrow computation, $(\ggg)$ composes two arrow computations by sending the output of the first arrow to the second. Branching and merging input and output values are done by combinators such as *first*, *second*, $(***)$, and $(\&\&\&)$. In fact, all the other combinators can be defined in terms of just *arr*, $(\ggg)$, and *first*, with the exception of *loop*. The *loop* combinator is useful for defining recursive structures. It is special in a sense that it feeds the output value of an arrow computation back to the arrow itself, and deserves its own class:

$$
\begin{array}{l}
\textbf{class } \mathit{ArrowLoop}\ a \textbf{ where} \\
\quad \mathit{loop} :: a\ (b, d)\ (c, d) \to a\ b\ c
\end{array}
$$

Figure 1 shows some commonly used arrow combinators.



(a) `arr f` (b) `sf1 >>> sf2` (c) `first sf`

(d) `sf1 &&& sf2` (e) `loop sf`

Figure 1: Commonly Used Arrow Combinators

Programming in a point-free style using the arrow combinators may lead to programs that are difficult to read. To mitigate this situation, recent Haskell compilers support the use of Paterson's arrow notation [Pat01]. The arrow syntax allows input and output values to be named, despite they not being first class values.

For example, consider the following program written using arrow syntax:

$$
\begin{aligned}
&\mathit{plus}\ f\ g = \mathit{proc}\ x \rightarrow \mathbf{do} \\
&\quad a \leftarrow f \multimap x \\
&\quad b \leftarrow g \multimap x \\
&\quad \mathit{returnA} \multimap a + b
\end{aligned}
$$

The program can be written in a semantically equilavent manner using the arrow combinators:

$$\mathit{plus}'\ f\ g = f \mathbin{\&\&\&} g \ggg \mathit{arr}\ (\lambda(a, b) \rightarrow a + b)$$

The keyword *proc* is similar to λ, except that it constructs an arrow instead of a function. The $\multimap$ symbol sends the value labeled by the right hand side to the arrow on the left. Output values are labeled on the left hand side of the $\leftarrow$ symbol. *returnA* specifies the final output of the arrow being defined.

The arrow syntax gives a stronger signal-processing feel to programs and is one of the reasons we use arrows as a basis for designing our sound synthesis DSL.

2.2 Basic Elements

In this section we show how audio signals and can be modeled in terms of arrows. Given a signal of type α, a *signal transformer* which turns it into a signal of type β can be viewed as a *signal function* from α to β:

$$\mathtt{SF}\,\alpha\,\beta \approx \mathtt{Signal}\,\alpha \rightarrow \mathtt{Signal}\,\beta$$

A digital audio signal can be modeled as a stream of audio samples and conveniently expressed in terms of Haskell's list type. A signal function from type a to b is therefore a function from a list of a to a list of b, where a and b are the types of input and output samples respectively.[1]

$$\mathbf{newtype}\ \mathit{SF}\ a\ b = \mathit{SF}\ ([a] \rightarrow [b])$$

The **newtype** construct hides the underlying list function so that code outside the module only sees the signal function as an abstract data type. This effectively prevents users of the API from manipulating signals at the point level, thereby making the code more abstract and less prone to space leaks.

[1] In this report we will use the terms *stream* and *list* interchangeably to denote a non-strict sequence of values.

Note that, for now, the sample rate is implicit.

We are now ready to give a minimal complete definition for our arrow instance:

```
instance Arrow SF where
  arr f = SF (map f)
  first (SF f) = SF g
    where g l = let (x, y) = unzip l in zip (f x) y
  SF f >>> SF g = SF (g . f)
```

Here *arr* lifts a function from the point level to the list level. *first* takes in a signal function and returns another, which sends its first element in the input pair through the given signal function and returns the result and the second input as a pair. (>>>) is simply function composition.

2.2.1 Unit Generators as Signal Functions

"Unit generators" are traditionally the building blocks for audio synthesis. For example, an oscillator is a unit generator that produces signals that can be subsequently processed by other unit generators, such as adders, multipliers, filters, envelopes, and so on. In our framework, we treat a unit generator as a signal function.

Some unit generators, such as oscillators, produce signals that do not necessarily depend on the input. For example, a sine wave generator might receive a control voltage as input, but its output still oscillates over time even if the input remains constant. The output of such unit generators is a function of time. Although time is explicit in our definition of signal functions, i.e. it does not appear in the type of *SF*, such unit generators can be modeled by *stateful* signal functions. Stateful signal functions keep an internal state, and their output changes over time as a result of the state change. One way to have a signal function "remember" its state in a purely functional setting is to feed the current state information back to the function itself as part of its input while it is running. The state information can be the current output, the internal state, or any combination of them. To express a feedback loop, we need to provide an instance of the *ArrowLoop* class:

```
instance ArrowLoop SF where
  loop (SF f) = SF (λx → let (y, z) = unzip (f (zip x z)) in y)
```

Due to the discrete nature of the underlying stream implementation, to prevent infinite recursion we need to introduce a unit delay. The *delay* function outputs a default initial value i followed by the input stream, thereby delaying the input signal by one time step. Its definition is simple:

```
delay :: a → SF a a
delay i = SF (i:)
```

A sine wave oscillator. We now give an example of a stateful unit generator. The following recurrence equation defines a series approximating sine, where f is the oscillating frequency, $\omega = 2\pi f$, and h is the delta time between each discrete sample:[2]

$$\begin{aligned}
&\mathrm{sf}(0) = 0 \\
&\mathrm{sf}(1) = \sin \omega h \\
&\mathrm{sf}(n) = c \cdot \mathrm{sf}(n-1) - \mathrm{sf}(n-2) \;\text{ where }\; c = 2\cos \omega h
\end{aligned}$$

In arrow parlance, this can be expressed as:

```
type Sample = Double


sine :: Double → SF () Sample
sine freq =
  let omh = 2 * pi * freq * h
      d   = sin omh
      c   = 2 * cos omh
      sf  = proc _ → do rec
              let r = c * d2 - d1
              d1 ← delay 0 ⤙ d2
              d2 ← delay d ⤙ r
      returnA ⤙ r
  in sf
```

Figure 2 shows the equivalent block diagram.

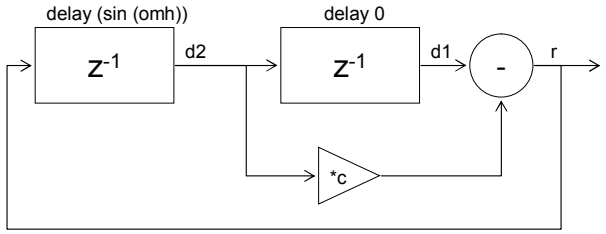


Figure 2: Block Diagram for Sine

We should note that, although signals are encoded in Haskell's *Double* type, an audible signal's value at any given sample should always range between -1 and 1, to ensure that it is within range of the low-level audio library. Out-of-range signals will be clipped, creating undesirable noises and artifacts.

[2] In Section 2.5.2 we will discuss this method in more detail.

The integral function. Here is another example of a stateful signal function. Instead of throwing away its input values, the *integral* function computes the integral of its input signal over time:

$$
\begin{aligned}
&\mathit{integral} :: \mathit{SF}\ \mathit{Sample}\ \mathit{Sample}\\
&\mathit{integral} = \mathit{loop}\ (\mathit{arr}\ (\lambda(x, i) \to i + \mathit{dt} * x) \ggg\\
&\quad \mathit{delay}\ 0 \ggg \mathit{arr}\ (\lambda x \to (x, x)))
\end{aligned}
$$

The x variable in the first line is a point of the input stream and dt is the delta time between samples. The result is duplicated into a pair, of which the second value is fed back as i. We will see uses of this function later in this report.

2.2.2 Envelopes and Operators

Envelopes modulate the amplitude of a signal. They are easy to define; we simply multiply the signal with the enveloping signal. On the other hand, adding two signals produces a new signal equivalent to playing them at the same time, which is essential in modeling additive synthesis. We define them as binary operators:

$$
\begin{aligned}
&\mathit{biOp} :: \mathit{SF}\ a\ b \to \mathit{SF}\ c\ d \to (b \to d \to e) \to \mathit{SF}\ (a, b)\ e\\
&\mathit{biOp}\ a\ b\ \mathit{op} = a \mathbin{***} b \ggg \mathit{arr}\ (\mathit{uncurry}\ \mathit{op})
\end{aligned}
$$

$$
\begin{aligned}
&(\mathbin{<+>}) :: \mathit{SF}\ a\ \mathit{Sample} \to \mathit{SF}\ b\ \mathit{Sample} \to \mathit{SF}\ (a, b)\ \mathit{Sample}\\
&a \mathbin{<+>} b = \mathit{biOp}\ a\ b\ (+)
\end{aligned}
$$

$$
\begin{aligned}
&(\mathbin{<*>}) :: \mathit{SF}\ a\ \mathit{Sample} \to \mathit{SF}\ b\ \mathit{Sample} \to \mathit{SF}\ (a, b)\ \mathit{Sample}\\
&a \mathbin{<*>} b = \mathit{biOp}\ a\ b\ (*)
\end{aligned}
$$

Other binary operators can be defined in a similar manner. These are mainly useful for writing programs without the arrow notation, since the compiler automatically lifts arithmetic operators used in arrow notation to the arrow level.[3]

2.2.3 Running a signal function

Programming in arrows prevents programmers from directly manipulating values at the point level. But eventually, to get any value out of a signal function, we need to extract and apply the function wrapped inside *SF*. We provide a function *runSF* to run a given signal function. Since it is defined in the same module as that of *SF*, it has access to the inner function.

$$
\begin{aligned}
&\mathit{runSF} :: \mathit{SF}\ a\ b \to [a] \to [b]\\
&\mathit{runSF}\ (\mathit{SF}\ f)\ \mathit{inp} = \mathit{force}\ (f\ \mathit{inp})
\end{aligned}
$$

[3]The reason we cannot provide a *Num* instance for these operators is that their type is not consistent with the type in the *Num* class, that is, $a \to a \to a$.

```
  where
    force [] = []
    force (x : xs) = x `seq` (x : force xs)
```

Essentially, *runSF* converts the signal function into a direct stream transformer. Here the use of *force* avoids accumulating too many thunks past evaluation due to laziness.

Given a unit generator $ug :: SF\ a\ Sample$, we can now write a function to create a stream of output values, which can be passed to the audio library for playing or writing to a file:

```
mkStream :: SF () Sample → [Sample]
mkStream ug = runSF ug $ repeat ()
```

2.3 Sampling Rates

2.3.1 Motivation

One of the advantages of FRP and Yampa [WH00, HCNP03] is that the programmer can treat signals as continuous entities and not worry too much about the discrete implementation details. This makes the code more succinct and more readable. In practice, however, if we simply write code as if we were building analog circuits, the resulting program might be somewhat inefficient. The reason is that the underlying implementation is approximating a continuous signal with discrete samples, and the accuracy closely depends on the sampling frequency. Nyquist's sampling theorem tells us that the sampling frequency should be at least twice the highest frequency that we wish to unambiguously represent, to avoid aliasing. Therefore, to preserve frequencies within the audible range of the human ear, audio CDs use a sampling frequency of 44,100 Hz. Our framework would work perfectly fine if every signal is running at exactly this rate, but there are some signals that don't require such high accuracy. For example, low-resolution signals such as envelopes can use a much lower sampling rate and be audibly indistinguishable from the same signal sampled at a higher rate. Generating all signals at the same high rate is computationally expensive, and thus we wish to develop ways to sample at lower rates when the higher rate is not required.

Some audio programming languages allow the coexistence of different signal rates. For example, csound [Ver91] makes the distinction between "sample rate" and "control rate" through an ad-hoc naming convention: variable names are prefixed with 'a' and 'k' for audio signals and control signals, respectively, and opcodes are "typed" in that the rate type of each argument to the opcode is enforced.

While csound's approach works, it is ad hoc and only allows a small set of fixed rates. we would like to provide a bit more flexibility. First, we want scalability:

an arbitrary number of rates can coexist in the same program. Second, signal functions should also be polymorphic: we shouldn't need to write two specialized oscillators using fundamentally the same algorithm just because we have to accommodate two different rates.

A naïve approach to this problem would be to allow the signal rate to be a run-time value which is passed to each unit generator. For example, we could imagine defining sine like this:

$$\begin{aligned} &\mathit{sine} :: \mathit{Double} \to \mathit{Int} \to \mathit{SF}\ a\ \mathit{Sample} \\ &\mathit{sine}\ \mathit{freq}\ \mathit{rate} = ... \end{aligned}$$

This has several disadvantages. It is a bit too flexible—every signal function can run at different rates, and the rates can be changed dynamically at run-time. In practice, once a rate has been decided, it is seldom changed during run-time. If every unit generator carries an extra rate argument, the program will soon be plagued with values of various rates, some common and some different. The common ones could be factored out with some global variable, but this is still inconvenient. It Is also easy to make typographical errors with all the extra arguments floating around, making it a source of programming errors. Worse, there is no way the system can check for inconsistent rates. Programs such as $\mathit{sine}\ 440\ 44100 \mathbin{<\!\ast\!>} \mathit{sine}\ 1\ 441$ do not make much sense since the operator $<\!\ast\!>$ cannot infer the signal rate of each operand. Even if it could, what should the rate of the resulting signal be?

We could fix the data type *SF* to carry an additional *rate* value. For instance:

$$\mathbf{newtype}\ \mathit{SF}\ a\ b = \mathit{SF}\ ([a] \to [b], \mathit{Int})$$

But this is rather clumsy and still does not address the inconveniences mentioned above.

2.3.2 Implicit Signal Rates

To sum up, a good solution to the sample rate problem should achieve several goals. We should allow an arbitrary number of rates to coexist in a program; the rates can be specified by the library user in the user's code; there should not be excessive parameter passing—the rates, once defined, should automatically *propagate* through the program whenever possible; and finally, signals of different rates cannot be mixed together without explicit coercion.

Haskell has a powerful type system, and a solution that leverages the type system can perform all the checks statically. Indeed, Kiselyov and Shan [KcS04] gives a solution to this kind of *configurations problem* using Haskell type classes. We now show how to adapt their technique to our problem.

We first define a type class *Clock* to embed the rate of a signal.

```
class Num r ⇒ Clock p r | p → r where
  rate :: p → r
```

It has exactly one method *rate* that returns the embedded rate value. The type parameter p is a phantom type and has no term representation. It exists so that the type system can uniquely determine the type of the rate r from p, as stipulated by the functional dependency $p \to r$.

Then we augment the type of the signal function with the phantom type p. One can think of p as the name of a particular clock rate r.

```
newtype SF p b c = SF ([b] → [c])
```

The definition of the arrow instance is unchanged except for the added type p:

```
instance Arrow (SF p) where
  arr f = SF (map f)
  ...
```

To look up the rate associated with type p, we just need to call the *rate* function. For example, we redefine *sine* to use the new facility:

```
sine :: Clock p Int ⇒ Double → SF p a Sample
sine freq =
  let omh = 2 * pi * freq / (fromIntegral sr)
      d   = sin omh
      c   = 2 * cos omh
      sr  = rate (⊥ :: p)
      sf  = proc _ → do rec
                          let r = c * d2 − d1
                          d2 ← delay d −≺ r
                          d1 ← delay 0 −≺ d2
      returnA −≺ r
  in sf
```

Note the use of the lexically-scoped type variable p in the call to *rate* to obtain the value associated with type p. Since p is never inhabited, we simply supply $\bot$ and bind it to the type p. Here the functional dependency $p \to r$ defined in the *Clock* class comes into play and uniquely determines the corresponding instance of *Clock*. Once the instance is chosen, *rate* returns the associated clock value. The *Clock* instance can exist either inside the library or be defined in the user's module:

```
data AudRate   -- audio rate
data CtrRate   -- control rate


instance Clock AudRate Int where
  rate _ = 44100
```

```
instance Clock CtrRate Int where
  rate _ = 4410


type AR = SF AudRate () Sample
type CR = SF CtrRate () Sample
```

The types *AudRate* and *CtrRate* are phantom types, so we give them empty data definitions. *AR* and *CR* are type synonyms, defined as shorthands for signal functions operating at audio rate and control rate, respectively. Now if we write

```
s = let s1 = sine 440 :: AR
        s2 = sine 1   :: CR
    in s1 <*> s2
```

We get a nice error message from GHC which tells us exactly what makes the type-checking fail:

```
Couldn't match expected type 'AudRate'
       against inferred type 'CtrRate'.
    Expected type: SF AudRate b' c'
    Inferred type: SF CtrRate () Float
```

We have shown how to embed rate values in types and leverage the Haskell type checker to do the bookkeeping for us. One benefit of encoding the sample rate into the types is that we only need to specify the rate type for each unit generator, and type inference will take care of the rest. We don't need to annotate every expression with a sample rate as long as there is no conflict or ambiguity.

2.3.3 Rate Coercion

The reason for mixing signals of different rates is efficiency—signals that require only low resolution can be generated at a lower rate than audio signals. However, operators such as <*> require that their two arguments have the same type. In other words, the two arguments, when unfolded into a stream, should produce samples of the same time interval, and only then does the semantics of <*> make sense. It then becomes necessary to coerce a signal of one rate to another:

```
coerce :: (Clock p1 Int, Clock p2 Int) =>
  SF p1 a c -> SF p2 a c
coerce (SF f) =
  let inRate  = fromIntegral $ rate (undefined :: p1)
      outRate = fromIntegral $ rate (undefined :: p2)
      ratio   = inRate / outRate
      pairs   = (0,0) : map (\(i,r) -> properFraction (r + ratio)) pairs
```

```
    skips     = fst $ unzip pairs
    g inp     = ziip skips (f inp)
  in SF g


ziip []       _              = []
ziip (i : is) elts@(e : _) = k : (ziip is ks)
  where ks@(k : _) = drop i elts
```

The function *coerce* looks up the signal rates of the input and output signals from the type variables *p1* and *p2*. It then either stretches the input stream by duplicating the same element or contracts it by skipping elements. It is also possible to define a more accurate coercion function that performs interpolation, at the expense of performance.

For simpler programs, the overhead of calling *coerce* might not be worth the time saved by generating signals with lower resolution. (Haskell's fractional number implementation is relatively slow.) Here we define a specialized up-sampling function that avoids calling *properFraction* but only works when the output rate is an integral multiple of the input rate.

```
upSample :: (Clock p1 Int, Clock p2 Int) ⇒
  SF p1 a c → SF p2 a c
upSample (SF f) =
  let inRate  = rate (⊥ :: p1)
      outRate = rate (⊥ :: p2)
      ratio   = outRate `div` inRate
  in SF (λinp → concatMap (replicate ratio) (f inp))
```

In any case, now that we have a coercion function, we can rewrite the program fragment that generated a type error, as:

```
s = let sig  = sine 440 :: AR
        amp = sine 1   :: CR
    in sig <*> coerce amp
```

Type inference automatically unifies the types on both sides of <*>, and the *amp* signal is up-sampled to the audio rate then multiplied with *sig*.

2.4 Optimizing Arrows

2.4.1 Overhead of Arrows

Arrows are useful in that they eliminate a certain class of space leaks [LH07], and as we have seen, their expressiveness fits nicely into audio synthesis. But arrows also introduce a certain amount of performance overhead. Compare

sine defined in 2.3.2 with the program shown as follows, which uses the same algorithm to approximate sine but is written without using arrows:

```
fastSine :: Frequency → Int → [Sample]
fastSine freq sr =
  let omh = 2 * pi * freq / (fromIntegral sr)
      i   = sin omh
      c   = 2 * cos omh
      sine = 0 : i : zipWith (λb a → c * a − b) sine (tail sine)
  in sine
```

If we measure the time taken to unfold and sum together $30 * 44100$ elements of both methods on the same machine, we find that *sine* executes roughly 15 times slower than *fastSine*, and 20 times slower than an array-based implementation written in C. What causes this performance degradation? Can arrows ever match the speed of C?

2.4.2 Rewrite Rules in GHC

GHC's rewrite rules mechanism is used to implement list fusion and deforestation [GLJ93, GJ94], so it serves as a good starting point if we hope to simplify arrow expressions.

In fact, in the current GHC library, there are already a few rules that capture some common arrow laws:

```
"compose/arr"     ∀ f g . arr f ⋙ arr g = arr (f ⋙ g)
"first/arr"       ∀ f .   first (arr f) = arr (first f)
"second/arr"      ∀ f .   second (arr f) = arr (second f)
"product/arr"     ∀ f g . arr f *** arr g = arr (f *** g)
"fanout/arr"      ∀ f g . arr f &&& arr g = arr (f &&& g)
"compose/first"   ∀ f g . first f ⋙ first g = first (f ⋙ g)
"compose/second"  ∀ f g . second f ⋙ second g = second (f ⋙ g)
```

With these rules in effect, GHC is able to simplify a number of contrived expressions. But the rules don't work so well in practice. One reason is that people prefer to program in arrow notation most of the time, but GHC's arrow preprocessor creates extra tupling, as independent signal functions are translated into a linear chain of dependent functions, which further limits the applicability of these rewrite rules. Neil Sculthorpe gave a good illustration on the Yampa-users mailing list [Scu08]:

> [This example] demonstrates the tupling problem and how a linear chain of dependencies gets created from (what should be) independent signal functions.
>
> The translation rules can be found in Ross Paterson's paper, "A New Notation for Arrows [Pat01]."

Here's the sugared version of a signal function:

```
robotBehaviour :: SF Input Output
robotBehaviour = proc inp → do
                   fDis ← sfDis      -≺ inp
                   lDis ← sfDisL     -≺ inp
                   rDis ← sfDisR     -≺ inp
                   let dir = turnDir fDis lDis rDis
                   col  ← sfLampCol -≺ fDis
                   sfOut             -≺ (col, dir)
```

And here's the unsugared version. (To be fair, the arrow pre-processor does then optimise this somewhat, but this demonstrates the point.)

```
robotBehaviour :: SF Input Output
robotBehaviour =
  arr id &&& sfDisF >>>
  arr id &&& (arr (λ(inp, fDis) → inp)          >>> sfDisL)       >>>
  arr id &&& (arr (λ((inp, fDis), lDis) → inp) >>> sfDisR)       >>>
  arr id &&& (arr (λ(((inp, fDis), lDis), rDis) →
                (fDis, lDis, rDis))             >>> arr turnDir) >>>
  arr id &&& (arr (λ((((inp, fDis), lDis), rDis), dir) → fDis)
                                                  >>> sfLampCol)  >>>
  arr (λ(((((inp, fDis), lDis), rDis), dir), col) →
            (col, dir))                           >>> sfOut
```

We could come up with more rewrite rules that apply to more cases, and could improve the arrow preprocessor to simplify the resulting program even more. However, rewrite rules are somewhat delicate and difficult to manipulate. For programs written strictly using these combinators, the rules work fine. In reality, users often lift their own functions into arrows, which cannot by any means be optimized away by rewrite rules. Rules become even less robust when we have to control exactly which rules need to fire before or after certain phases of inlining during compilation.

2.4.3 Double Lifting

In this section we show how GHC can be nudged into doing more optimization on pure arrows.

Consider a simple program written in arrow notation:

```
foo = proc x → do
  y ← f   -≺ x + 1
  g       -≺ 2 * y
  let z = x + y
```

$$
\begin{aligned}
&t \leftarrow h \quad \multimap x * z \\
&returnA \multimap t + z
\end{aligned}
$$

Let's also define f, g, and h as follows:

$$
\begin{aligned}
f &= arr\ (+5) \\
g &= arr\ (+6) \\
h &= arr\ (+7)
\end{aligned}
$$

The arrow preprocessor turns this into:

$$
\begin{aligned}
&(arr\ (\lambda x \rightarrow (x, x)) \ggg \\
&\quad (first\ (arr\ (\lambda x \rightarrow x + 1) \ggg f) \ggg \\
&\quad\quad arr\ (\lambda (y, x) \rightarrow (y, (x, y)))) \\
&\quad\quad \ggg \\
&\quad\quad (first\ (arr\ (\lambda y \rightarrow 2 * y) \ggg g) \ggg \\
&\quad\quad\quad arr \\
&\quad\quad\quad\quad (\lambda (\_, (x, y)) \rightarrow \\
&\quad\quad\quad\quad\quad \textbf{let} \\
&\quad\quad\quad\quad\quad\quad z = x + y \\
&\quad\quad\quad\quad\quad\quad \textbf{in}\ ((x, z), z))) \\
&\quad\quad\quad \ggg \\
&\quad\quad\quad (first\ (arr\ (\lambda (x, z) \rightarrow x * z) \ggg h) \ggg \\
&\quad\quad\quad\quad arr\ (\lambda (t, z) \rightarrow t + z)))
\end{aligned}
$$

If we simply treat this program as an instance of an SF, that is, a stream-based signal function, we will end up wasting a lot of time lifting values into lists and zipping / unzipping tuples.

But what if the underlying arrow representation of this program is just based on functions? Let's recall the instance of function arrows:

$$
\begin{aligned}
&\textbf{instance}\ Arrow\ (\rightarrow)\ \textbf{where} \\
&\quad arr\ f = f \\
&\quad f \ggg g = g\ .\ f \\
&\quad first\ f = f \ast\!\!\!\ast\ id \\
&\quad second\ f = id \ast\!\!\!\ast\ f \\
&\quad (f \ast\!\!\!\ast\ g) \sim (x, y) = (f\ x, g\ y)
\end{aligned}
$$

If we annotate the type of *foo* defined above as $Int \rightarrow Int$, GHC will compile it into the following Core representation:

$$
\begin{aligned}
&\lambda (eta\_X1YO :: GHC.Base.Int) \rightarrow \\
&\quad \textbf{case}\ eta\_X1YO\ \textbf{of}\ wild\_a209 \{ GHC.Base.I\ \#\ x\_a20b \rightarrow \\
&\quad \textbf{let}\ \{ \\
&\quad\quad y\_a20e\ [Just\ L] :: GHC.Prim.Int\ \# \\
&\quad\quad [Str : DmdType] \\
&\quad\quad y\_a20e = GHC.Prim.+\#x\_a20b
\end{aligned}
$$

```
      (GHC.Prim.+#(GHC.Prim.+#x_a20b 1) 5)
  } in
    GHC.Base.I # (GHC.Prim.+#
      (GHC.Prim.+#(GHC.Prim.*#x_a20b y_a20e) 7)
      y_a20e)
  }
```

Functions with the # suffix denote unboxed values, which means we are getting pretty efficient code. Ignoring the annotations, here is the gist of the resulting function:

$$\lambda x \rightarrow \textbf{let } z = x + ((x + 1) + 5) \textbf{ in } ((x * z) + 7) + z$$

GHC is able to produce such good code because of the inlining and beta-reduction involved during optimization. In this case, these built-in optimizations subsume the arrow laws and provide much better results. But in the case of stream-based arrows, the definitions of the arrow combinators get more complicated and GHC won't be so adamant about inlining things. Even if it did, without clever techniques such as deforestation, the generated code would not always be more efficient.

With pure arrows, the problem is easy to solve. Supposing that we have a stream-based arrow implementation, we can "optimize" the previous definition of *foo* by writing

```
foo = proc x → do
    y  ← f -≺ x + 1
    ...


foo' :: SF Int Int
foo' = arr foo
```

instead of just giving *foo* the type *SF Int Int*. We call this "double lifting," since *foo* is lifted to a function-based arrow first, and then lifted to *SF*. In the first stage, GHC performs its built-in beta-reduction and inlining, transforming *foo* into a function with a simpler structure. Then the *arr* in the definition of *foo'* lifts the optimized *foo* to the *SF* level. We let GHC do what it is good at, which is optimizing functions, then lift it to a more complicated form once the optimization is done.

This works really well, until there is *loop*. We simply can't define an instance of the *ArrowLoop* class in the pure function domain. So the trick described above no longer works, since stateful signal functions have to exist at the lifted level, not the function level, and we lose most benefits we get from doing inlining and beta-reduction.

Furthermore, if we are working at the unlifted level, we could just write things in terms of functions alone and not use the arrow notation at all:

$$
\begin{aligned}
\lambda x \rightarrow \textbf{let } & y = f\ (x + 1) \\
& g = 2 * x \\
& z = x + y \\
& t = h\ (x * z) \\
\textbf{in } & t + z
\end{aligned}
$$

As a result, we cannot really get significant performance gains if we don't optimize stateful functions. The next section explores various techniques for doing so, using sine waves as an example.

2.5 Optimizing Stateful Signal Functions

2.5.1 Sine Waves

We have seen how loops in arrow code inhibit optimization, yet loops are essential if we want to construct stateful signal functions from arrow combinators.

One topic of particular interest to us is the efficient generation of sinusoidal waves. A sine wave is a stateful, time-varying signal, and is the most fundamental element in any sound synthesis system, since nearly any waveform can be approximated in terms of sinusoidal waves by Fourier analysis. Sine waves capture many aspects of FRP in a nutshell.

The value in studying various methods for generating sine waves is two-fold. First, we need an efficient sine wave generator as a building block suitable for real-time sound synthesis. Second, by expressing sine in terms of arrows and streams, we hope to gain insight into how the underlying arrow implementation could be optimized, and be able to generalize and automate the optimization process to arbitrary arrow-based programs.

In the rest of this section, we present several approaches and compare their performance by unfolding the signal function to generate 30 seconds' worth of samples at 44100 Hz. To absolutely make sure that all samples are evaluated and not deferred due to laziness, we measure the time taken to sum up all the samples. The benchmarks are taken on a 2.16 GHz Intel Core 2 Duo machine running Mac OS X 10.5.2 and GHC 6.8.2. Figures 3 and 4 summarize the results of the benchmarks.

2.5.2 Fixed frequency algorithms

Library function. A straightforward way to compute a sine wave is by calling *sin* provided by the math library in Haskell.

```
sinF :: Clock p Int ⇒ Frequency → SF p a Sample
sinF freq = SF (λ_ → cycle $ take n $ map sin [0, d ..])
  where d = 2 * pi * freq / sr
```

$$
\begin{aligned}
n &= \mathit{truncate}\ (\mathit{sr}\ /\ \mathit{freq}) \\
\mathit{sr} &= \mathit{fromIntegral}\ \$\ \mathit{rate}\ (\bot :: p)
\end{aligned}
$$

This signal function simply ignores the input and generates a sine wave at the rate embedded in the phantom type `p`. Since sine is periodic, we compute 1 period of the function and cycle through it, essentially caching the values. Generating 30 seconds of samples takes 38.8 milliseconds. Indeed, caching speeds up computation significantly. If we simply generate the samples without caching, the resulting signal function takes 180 milliseconds to run.

Goertzel's algorithm. Goertzel's algorithm computes $\sin(a + nb)$ using the previous two values in the series, $\sin(a + (n-1)b)$ and $\sin(a + (n-2)b)$[Dat00].

Using the trigonometric identity

$$\sin(a + b) = \sin a \cos b + \cos a \sin b$$

we can perform the following calculation:

$$
\begin{aligned}
\sin(a + nb) &= x \sin(a + (n-2)b) + y \sin(a + (n-1)b) \\
&= x \sin(a + nb - 2b) + y \sin(a + nb - b) \\
&= \sin(a + nb)(x \cos 2b + y \cos b) - \cos(a + nb)(x \sin 2b + y \sin b)
\end{aligned}
$$

Therefore

$$\sin(a + nb)(x \cos 2b + y \cos b - 1) = \cos(a + nb)(x \sin 2b + y \sin b)$$

For the equation to hold for all n, we must have

$$x \cos 2b + y \cos b = 1$$

and

$$x \sin 2b + y \sin b = 0$$

Solving this yields $x = -1$ and $y = 2 \cos b$. Substituting back we get

$$\sin(a + nb) = 2 \cos b \sin(a + (n-1)b) - \sin(a + (n-2)b)$$

The Goertzel algorithm computes each element of the sine wave with only one multiplication and one subtraction and is therefore very fast. The catch is that each value depends on two previous results and the time interval between the samples must be fixed, so it is only useful for generating a sine wave with a fixed frequency.

The *fastSine* function defined in 2.4.1 implements this algorithm. Wrapping it in a signal function we get:

```
sinA :: Clock p Int ⇒ Frequency → SF p a Sample
sinA freq = SF (λ_ → fastSine freq sr)
  where sr = rate (⊥ :: p)
```

This runs in 21 milliseconds.

2.5.3 Variable frequency algorithms

While the algorithms described so far are reasonably fast, more sophisticated applications, such as frequency modulation, require an oscillator whose frequency is variable. In other words, instead of taking the frequency as a parameter, the frequency should be a time-varying input signal to the unit generator.

Giorgidze and Nilsson [GN08] present a Voltage Controlled Oscillator (VCO) with dynamically controllable frequency:

```
oscSine :: (Clock p Int) ⇒ Frequency → SF p Sample Sample
oscSine f0 = proc cv → do
  let f = f0 * (2 ** cv)
  phi ← integral -≺ 2 * pi * f
  returnA -≺ sin phi
```

This runs in 556 ms.

Here is the same code, rewritten by hand, without using arrow notation:

```
oscSine' :: (Clock p Int) ⇒ Frequency → SF p Sample Sample
oscSine' f0 = arr (2**) >>> arr (*(2 * pi * f0)) >>> integral >>> arr sin
```

This runs in 400 ms, which is about 40% faster than the code produced by the arrow preprocessor. This again shows that while the arrow syntax is more readable, the preprocessor does not generate very good code.

Even without the added overhead of arrow syntax, there is still room for improvement. We present a more efficient algorithm with variable frequency input which uses table look-ups. *oscA′* precomputes a single cycle of sine wave at a high resolution and stores the values into an array as a look-up table. The *idx* signal function takes frequencies as input and converts them into the corresponding indices in the look-up table.

```
oscA' :: Clock p Int ⇒ SF p Sample Sample
oscA' =
  let sr    = rate (⊥ :: p)
      sin1  = fastSine 1 sr
      array = listArray (0, sr - 1) sin1 :: UArray Int Sample
      idx   = loop (arr (λ(d, i) → (i + d) `mod` sr) >>>
                    delay 0 >>> arr dup)
  in arr truncate >>> idx >>> arr (unsafeAt array)
```

```
dup :: t → (t, t)
dup x = (x, x)
```

This runs in 261 ms, 50% faster than *oscSine′*.

Optimizing the *loop* combinator. In *oscA′*, since the *loop* part involves *delay* and extra tupling, the algorithm should run faster if we replace that with a tighter loop.

This hand-coded version uses the exact same algorithm, but explicit sharing is expressed in terms of *zipWith*, where the list *bs* is used in its own definition:

```
idx = SF (λas → let bs = 0 : zipWith (λd i → (i + d) `mod` sr) bs as in bs)
```

This version runs in 164 ms, about 60% faster than that defined using *loop*. This is by far the fastest way known to us to compute recursive functions like this using lists.

With arrows we must combine two outputs with a tuple, evident in the type of the arrow inside the *loop* combinator. This contributes to the performance hit. In fact, we have come across similar patterns several times. *fastSine* is also written this way, except that a term depends on the previous two instead of one. Another similar example is the *integral* function defined in Section 2.2.1, which can be rewritten in terms of *zipWith* as follows:

```
integral′ :: Clock p Int ⇒ SF p Sample Sample
integral′ = SF (λas → let bs = 0 : zipWith (λx i → i + dt ∗ x) bs as in bs)
  where dt = 1 / (fromIntegral sr)
        sr = rate (⊥ :: p)
```

As a result, we can rewrite *oscSine′*:

```
oscSine′′ :: (Clock p Int) ⇒ Frequency → SF p Sample Sample
oscSine′′ f0 = arr (2∗∗) ⋙ arr (∗(2 ∗ pi ∗ f0)) ⋙ integral′ ⋙ arr sin
```

This runs in 297 ms.

Finally, using the same *zipWith* technique, we can define a version that calls the library function. Curiously, this becomes very similar to the original *oscSine* by Giorgidze, but the input to the signal function is the frequency instead of the control voltage.

```
sinF′ :: Clock p Int ⇒ SF p Sample Sample
sinF′ = SF (λfreqs → let f = 0 : zipWith (λf0 fr → f0 + d ∗ fr) f freqs
                     in map sin f)
  where d  = 2 ∗ pi / (fromIntegral sr)
        sr = rate (⊥ :: p)
```

This runs in 186 ms.

The recurring pattern suggests that we can translate code of the form *loop* (*arr f* $\ggg$ *delay z* $\ggg$ *arr dup*) to *SF* ($\lambda as \rightarrow$ **let** *bs* = *z* : *zipWith* (*curry f*) *bs as* **in** *bs*) and get a huge performance gain. Being able to identify this kind of recurring pattern and optimize it should prove very useful. There are several things we may try:

- Generalize the above transformation into an axiom and have GHC rewrite it into the *zipWith* form. This should get rid of the zips and unzips caused by *dup* and all the wiring.
- Define a primitive construct for such recurrence relations. While the programmer can simply write *zipWith*, doing so exposes the underlying list implementation. We would like to still be able to use arrows. Although the *loop* combinator already provides such expressiveness, one can argue that it is a bit too general for many applications. The repeated use of *dup* and *delay* suggests there might be a better abstraction.

Discussion. Optimization on arrows largely depends on the nature of arrows. More aggressive optimizations can be carried out if we know that certain signal functions are pure. But in sound synthesis, a lot of signal functions are stateful. Without giving a richer semantics to arrows and strengthening the arrow laws, it is unlikely that much can be done in this respect, at least with GHC's current optimization capabilities.

Arrows are inherently tied with tupling, as we can see from the type of the combinators such as *first*, *second*, ($\ast\!\!\ast\!\!\ast$), (&&&), and *loop*. This means that GHC has to create/project from a tuple every time a value is passed through these operators, which involves non-trivial overhead. Couple that with lists, and we get extraneous zips and unzips everywhere—which are not always easy to deforest.

Are arrows really worth the extra overhead? If we didn't use arrows at all, could we still prevent space leaks? We will answer these questions in the next chapter.

Graphs The following graphs compare the performance of various sine algorithms previously discussed.

Figure 3 shows the performance of fixed frequency generators. For comparison's sake, we toss in a variable frequency unit generator (*oscA′*) with the fixed frequency ones. We also include execution times for Goertzel's algorithm implemented in C using arrays. All cached versions generate one period of sine wave and cycle through it. *constA* is a signal function whose output is always constant; it shows the overhead of using arrows.

Figure 4 shows the performance of variable frequency generators. As we can see from the results, rewriting arrow syntax by hand and tightening arrow loops with *zipWith* can go a long way in improving performance.

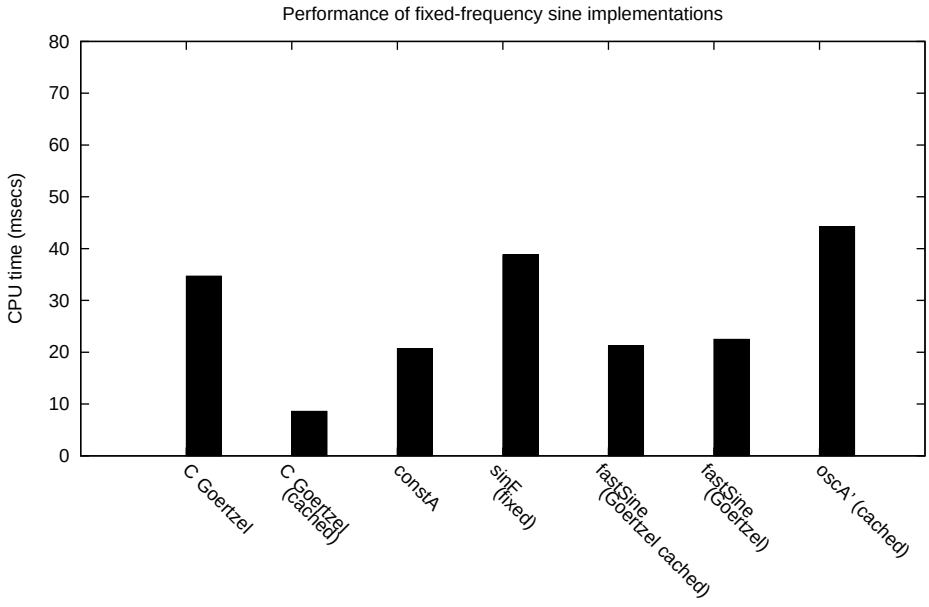


Figure 3: Fixed frequency: 30 seconds of 440Hz at 44.1kHz

2.6 Example: A Slide-Flute

In this section we show how to model a waveguide slide-flute [Mik00] based on Perry Cook's instrument. This is an example of digital waveguide modeling where delay lines are used to simulate samples of a traveling wave and its reflection. Unlike additive synthesis, waveguide instruments are modeled after physical systems, and can sound more realistic when properly tuned.

Figure 5 shows the block diagram of a physical model of a slide-flute. In the following paragraphs we will go through the implementation of each component, and show how to put everything together.

Delay line revisited. We have seen the unit delay function in Section 2.2.1. For the flute model, we need a more general delay line whose duration can be parametrized. The delay line inserts a silent signal of the given duration before the input signal. Its definition is also straightforward:

$$
\begin{aligned}
&\mathit{delayt} :: \mathit{Clock}\ p\ \mathit{Int} \Rightarrow \mathit{Time} \rightarrow \mathit{SF}\ p\ \mathit{Sample}\ \mathit{Sample} \\
&\mathit{delayt}\ \mathit{dur}\ (S\ \mathit{st}) = S\ \$\ \mathit{replicate}\ \mathit{samples}\ 0 +\!\!+ \mathit{st} \\
&\quad \textbf{where}\ \mathit{samples} = \mathit{time2samples}\ \mathit{sr}\ \mathit{dur} \\
&\qquad \mathit{sr} = \mathit{fromIntegral}\ \$\ \mathit{rate}\ (\bot :: p)
\end{aligned}
$$

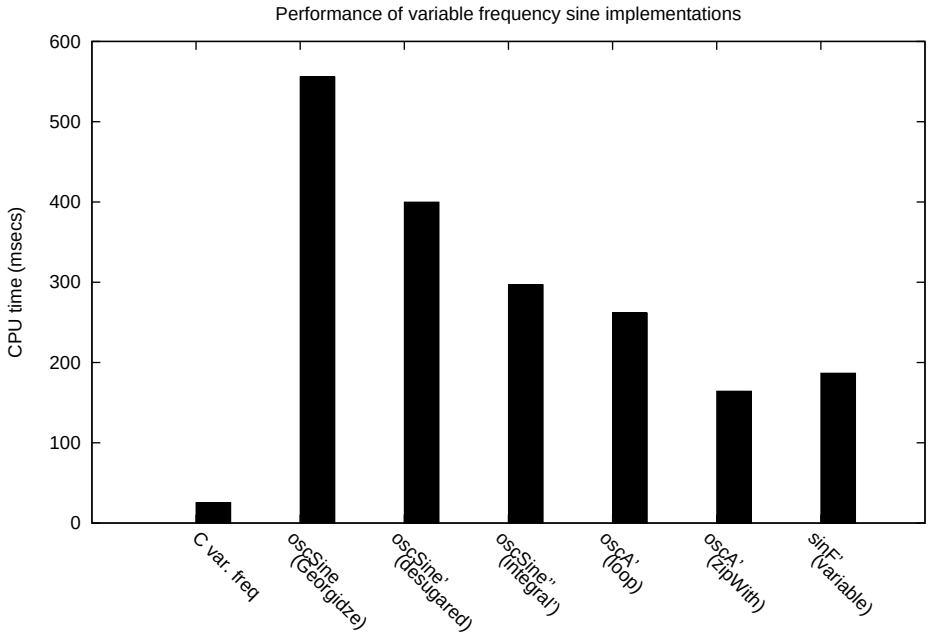


Figure 4: Variable Frequency: 30 seconds of 440Hz at 44.1kHz

To emphasize the notion of time, the delay line is parametrized by time instead of the number of samples, since the number of samples per a given period of time depends on the sample rate of the signal. Using the number of samples as an argument makes the code susceptible to changes in sample rates. Under the hood, we define a helper function *time2samples* to convert time in seconds into number of samples.

$$time2samples\ sr\ dur = truncate\ (dur * sr)$$

ADSR envelopes. ADSR envelopes, consisting of four phases—Attack, Decay, Sustain, and Release—are an important component in electronic instrument design. They are normally used to modulate the amplitude throughout the duration of a note. The duration of each phase can be specified, and a contour is formed by connecting between the starting and ending amplitudes of each phase.

The attack phase specifies how quickly the signal reaches its full amplitude once it is started. The decay phase specifies how quickly the signal drops to the sustain level after the initial attack. The sustain phase is usually flat, and the amplitude of the signal stays at the sustain level until the release phase. The release phase specifies how quickly the signal should attenuate when a note ends.

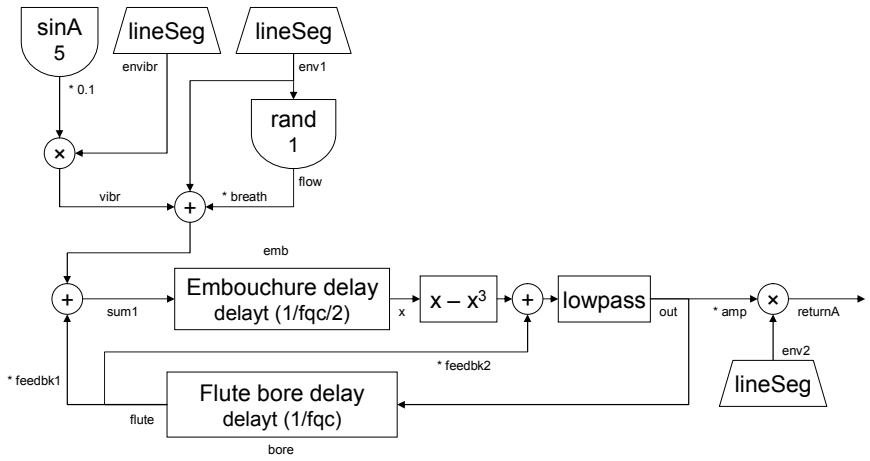


Figure 5: Block Diagram of the Slide-Flute

We provide a slightly more general envelope function than the traditional ADSR describe above. Our version is modeled after the *linseg* opcode in Csound. Instead of limiting the envelope signal to four phases, *lineSeg* generates a signal whose amplitude traces an arbitrary number of line segments between specified points. The *amps* argument represents the list of endpoints of each segment, and *durs* is a list containing the duration (in seconds) of each segment.

```
lineSeg :: Clock p Int ⇒ [Double] → [Time] → SF p a Sample
lineSeg amps durs =
  let lineSeg0 (a1 : a2 : as) (dur : ds) = sig ++ lineSeg0 (a2 : as) ds
    where sig     = take samples $ iterate (+del) a1
          del     = (a2 − a1) / (dur ∗ sr)
          samples = truncate (dur ∗ sr)
          sr      = fromIntegral $ rate (⊥ :: p)
    lineSeg0 _ _ = []
  in SF (λ_ → lineSeg0 amps durs)
```

Random number generator. To simulate a breath sound, we use the pseudo-random number generator in Haskell to generate a white noise signal.

```
rand :: Clock p Int ⇒ Double → SF p Sample Sample
rand amp (S amps) = S $ zipWith (∗) (randomRs (negate amp, amp) g) amps
  where g = mkStdGen 1234
```

Here *amp* denotes the absolute value of the signal range. *rand* takes in a signal and multiplies it by the white noise it generates; this is simply for convenience.

A low-pass filter. A low-pass filter passes low frequencies while attenuating high frequencies in a signal. There are many kinds of low-pass filters with different effects and behaviors, but for our purposes, a simple one works well enough [Wik08]. It is defined as the following equation:

$$output_t = output_{t-1} + c * (input_t - output_{t-1})$$

where $c = [0, 1]$ determines the cutoff point. When $c = 1$, all frequencies are passed; when $c = 0$, nothing is passed. For efficiency reasons discussed in 2.5.3, we define it in terms of *zipWith* instead of unit delays.

```
lowpass :: Clock p Int ⇒ Double → SF p Sample Sample
lowpass c (S as) =
  let bs = 0 : zipWith (λinp out → out + c * (inp − out)) as bs in S bs
```

Putting it all together. We are finally in a position to define the flute itself. Note the resemblance between the code and the block diagram. *dur* is the duration of a note, and the envelopes *kenv1*, *kenv2*, and *kenvibr* are defined in terms of it. *amp* denotes the output amplitude of the flute, *fqc* the frequency of the note, and *press* the breath pressure.

```
flute0 :: Time → Double → Double → Double → Double → AR
flute0 dur amp fqc press breath =
  let kenv1   = lineSeg [0, 1.1 * press, press, press, 0]
                        [0.06, 0.2, dur − 0.16, 0.02] :: CR
      kenv2   = lineSeg [0, 1, 1, 0]
                        [0.01, dur − 0.02, 0.01]      :: CR
      kenvibr = lineSeg [0, 0, 1, 1]
                        [0.5, 0.5, dur − 1]           :: CR
      bore    = delayt (1 / fqc)        -- bore delay
      emb     = delayt (1 / fqc / 2)    -- embouchure delay
      feedbk1 = 0.4
      feedbk2 = 0.4
  in proc _ → mdo
     env1   ← upSample kenv1   —≺ ()
     env2   ← upSample kenv2   —≺ ()
     envibr ← upSample kenvibr —≺ ()
     flow   ← rand 1.0         —≺ env1
     sin5   ← sinA 5           —≺ ()
     let vibr = sin5 * 0.1 * envibr    -- vibrato signal
         sum1 = breath * flow + env1 + vibr
     -- loop part
     flute  ← bore             —≺ out
     x      ← emb              —≺ sum1 + flute * feedbk1
```

$$
\begin{aligned}
&out \quad \leftarrow lowpass\ 0.27 \prec x - x * x * x + flute * feedbk2 \\
&returnA \prec out * amp * env2
\end{aligned}
$$

Although this is a moderately complex instrument, on a 2.16 GHz Intel Core 2 Duo machine and compiling using GHC 6.8.2, 30 seconds' worth of samples at 44100 Hz sampling rate can be rendered in 5.8 seconds. With minimal buffering (< 10 ms latency), the instrument can be played in real-time without skipping.

3 Programming Without Arrows and Beyond

Are arrows really the natural choice for designing a sound synthesis framework? The benefits of arrows come at a cost—the core arrow combinators are clunky to use, and their readability gets lost in tuples pretty quickly. Although the arrow notation does ameliorate the situation somewhat, programming in terms of signals rather than signal functions is still more natural and intuitive. It is also easier for the compiler to generate more efficient code, as we have already seen how arrows can introduce extra overhead where there could have been none. As an example, compare the sine generator defined in 2.3.2 and the following version defined in terms of Haskell lists:

$$
\begin{aligned}
&sineL :: Double \rightarrow [Double] \\
&sineL\ freq = \\
&\quad \textbf{let}\ omh = 2 * pi * freq * h \\
&\qquad\quad d \quad = sin\ omh \\
&\qquad\quad c \quad = 2 * cos\ omh \\
&\qquad\quad r \quad = zipWith\ (\lambda d2\ d1 \rightarrow c * d2 - d1)\ d2\ d1 \\
&\qquad\quad d1 \quad = delay\ 0\ d2 \\
&\qquad\quad d2 \quad = delay\ d\ r \\
&\quad \textbf{in}\ r \\
\\
&delay = (:)
\end{aligned}
$$

The list-based version is not much different from the arrow-based version, but *sineL* benefits from existing deforestation capabilities in GHC and can be easily understood by any Haskell programmer. Furthermore, the recursive nature of the signal r is conveniently expressed using Haskell's **let** construct. No awkward arrow loops are involved!

We have been emphasizing that arrows can help avoid space leaks. But if streams could be used in a disciplined way such that those space leaks are avoided, why use arrows at all?

3.1 Changes to Signal Functions

Addressing potential space leaks. Liu and Hudak [LH07] showed that the class of space leaks we were concerned about did not exist if the delta time in

the FRP implementation were fixed. This is also true for non-arrow-based FRP. In fact, in the case of audio synthesis, delta times are always constant,[4] and this justifies an arrow-free implementation of our framework. We still start out with lists—a signal is a stream of values:

$$\textbf{newtype}\ S\ p\ a = S\ p\ [a]$$

If we make the data constructor S visible only in the module it is defined, the signals are still kept abstract. The fact that signals are implemented using lists is not exposed to the user, as is the case with arrows. This might help prevent other kinds of space leaks, since the user cannot accidentally hold on to elements in a stream in unexpected ways.

With the new signal type in place, the users are still provided with a basic set of operations on streams, and functions can be lifted into signal functions as well:

$$\textbf{type}\ SF\ p\ a\ b = S\ p\ a \to S\ p\ b$$

$$\begin{aligned} &\mathit{lift} :: \mathit{Clock}\ p\ \mathit{Int} \Rightarrow (a \to b) \to \mathit{SF}\ p\ a\ b \\ &\mathit{lift}\ f\ (S\ as) = S\ (\mathit{map}\ f\ as) \end{aligned}$$

Note that we still keep the phantom type p as a key to look up the sample rate of the signal.

The type *SF* has now become a type synonym. A signal function is now a first-class Haskell function, thus giving us the option to program in a *pointful* way rather than in a completely point-free style. Programming in a point-free style sometimes yields simple and elegant code, but to pass arguments around in a way not consistent with the shape of the functions involved, we need products (tuples) to share values, hence the extra tupling and wiring. This is the main reason that causes excessive tupling in arrow programming. But when we program in a pointful style, most of the plumbing disappears because we have normal lexical environments. Yet, nothing prevents us from programming in a point-free style should we find situations where doing so gives more clarity.

The definitions for most of our existing signal generating functions can be left virtually unchanged—we just replace $SF\ (\lambda\_ \to x)$ with $S\ x$.

When designing the library, sometimes it is convenient if we can "peel off" the outer layer of a signal, revealing the stream inside:

$$\begin{aligned} &\mathit{unS} :: \mathit{Clock}\ p\ \mathit{Int} \Rightarrow S\ p\ a \to [a] \\ &\mathit{unS}\ (S\ as) = as \end{aligned}$$

As a convenience, we also provide a function that lifts a constant to a signal:

[4] Signals with different sample rates have different delta times, but we always assume that a signal maintains the same sample rate throughout its lifetime.

$constS :: Clock\ p\ Int \Rightarrow a \rightarrow S\ p\ a$
$constS = S\ .\ repeat$

The performance gained from removing arrows from the framework is striking. The flute instrument defined in 2.6 now generates 30*44100 samples in 4.5 seconds, which is 30% faster.[5]

4 Future Work

Infinite stream representation. We have used Haskell's list type for our stream representation throughout the framework. Since we almost always work with infinite and lazy streams, an alternative would be to define a stream type such as:

newtype *Stream a* = *a* ::: *Stream a*

This could have several advantages over lists. First, we don't have the overhead of pattern-matching the empty list anymore, and this might improve performance by some factor. Second, the lack of the empty case in *Stream* might have simpler strictness properties and less prone to space leaks. A disadvantage is that we will miss out the deforestation rules defined for lists in GHC, so we will need to redefine them for infinite streams.

Multi-core support. In our framework, unit generators are defined and combined in a declarative nature, and are highly parallelizable. It would be an interesting research topic to support multiple threads and processors in an efficient manner.

Interaction with Haskore. Some concepts such as reactivity could be applied to Haskore [HMGW96], a functional high-level music DSL. Haskore currently lacks the ability to process real-time MIDI events, and the interaction between MIDI, notes, instruments, and musical performance needs to be explored further if we introduce reactivity.

More unit generators. Although we have shown how to implement some proof-of-concept instruments in this report, our library still lacks many unit generators present in more popular sound synthesis languages. We need to implement efficient variable delay taps, reverberation models, better digital filters, and so forth. It is our hope to eventually build a practical computer music library that is flexible, elegant, and simple to use.

[5] The performance bottleneck here is actually Haskell's built-in pseudo-random number generator, which we use to simulate the breath sound. If we replace the PRNG with the identity signal function, the arrow-based instrument runs in 1.9 seconds, while the arrow-free version runs in 1.1 seconds. That's a 72% increase in performance!

5 Acknowledgements

This research was supported in part by NSF grants CNS-0720682 and CCF-0811665.

References

[BB01] Ross Bencina and Phil Burk. PortAudio – an Open Source Cross Platform Audio API. In *Proceedings of the International Computer Music Conference (ICMC)*, Havana, Cuba, September 2001.

[Dat00] Scott Dattalo. Sine waves. Available at `http://www.dattalo.com/technical/theory/sinewave.html`, March 2000.

[GJ94] Andrew Gill and Simon Peyton Jones. Cheap deforestation in practice: An optimizer for Haskell. In *IFIP Congress (1)*, pages 581–586, 1994. International refereed conference, cited 19 times.

[GLJ93] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993.

[GN08] George Giorgidze and Henrik Nilsson. Switched-on yampa. In Paul Hudak and David Scott Warren, editors, *PADL*, volume 4902 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2008.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Robots, arrows, and functional reactive programming. In *Summer School on Advanced Functional Programming, Oxford University*. Springer Verlag, LNCS 2638, 2003.

[HMGW96] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[Hug00] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.

[KcS04] Oleg Kiselyov and Chung chieh Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 33–44, New York, NY, USA, 2004. ACM.

[LH07] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193:29–45, 2007.

[Mik00] Hans Mikelson. Mathematical Modeling with Csound. In Richard Boulanger, editor, *The Csound Book*, pages 372–374. MIT Press, 2000.

[Pat01] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[Scu08] Neil Sculthorpe. Email message: subject: Fruit. Message posted on the Yampa-users mailing list, available at `http://mailman.cs.yale.edu/pipermail/yampa-users/2008-February/000263.html`, February 2008.

[Ver91] Barry Vercoe. *The Csound Reference Manual*. Cambridge, MA, 1991.

[WH00] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proceedings of Symposium on Programming Language Design and Implementation*, pages 242–252. ACM, 2000.

[Wik08] Wikipedia. Low-pass filter, 2008. Online; accessed 12-May-2008.