Euterpea: From Signals to Symphonies

Paul Hudak
Yale University
Department of Computer Science
How to have fun combining your hobby with your work

or

How an application area can motivate your primary research
The goal is not to teach technology; it is to teach the mathematics, engineering, and computer science that underlie the technology.

Undergraduate:
- BS major in Computing and the Arts
- Specialized tracks in Art, Art History, Music, Theater Studies, and (coming soon) Architecture

Graduate:
- MS Degree in Computing and the Arts
- PhD Degree in CS with focus on Computing and the Arts

Laboratories for graphics and music (new)

My goal:
Figuring out how PL research can enhance all this
Euterpea

“Euterpea” is derived from “Euterpe”, who was one of the nine Greek muses, or goddesses of the arts, specifically the muse of music.
Haskell

- A non-strict ("lazy"), purely functional, general purpose, programming language.
- Designed in 1990.
- Noted for its advanced type system, monads, purity, and lazy evaluation.
- Has influenced many other language designs (e.g. Python, C#, LINQ, Java, Scala, Clean, Mercury, Curry, Cayenne, Isabelle)
- Has an excellent compiler and a thriving user support community.
Euterpea: Computer Music in Haskell

Why do this? After all, there are already many languages for computer music...

Motivation:

• A required two-course computer-music sequence
• The use of modern, state-of-the-art PL ideas in a fun area
• To combine my research with my hobby
• To use music to teach (functional) programming
• To give artists some powerful tools to enhance creativity

Last term, out of 24 students, 5 were Computing and the Arts majors – the rest were undergrads and grads with an interest in computer music, or a desire to learn about Haskell and FP.
Can we change the way artists think?

• Three ways that FP can help artists:
  – Abstraction
  – Abstraction
  – Abstraction

• Examples from the Haskell world:
  – The usual: higher-order functions, lazy evaluation, etc.
  – The unusual: higher-order types, monads, arrows, and other computational abstractions.

• “Monads for Artists”? (yeah right...)
Euterpea by Example

• Examples that demonstrate:
  – The basic ideas underlying Euterpea
  – Some of its unique design features
  – How Euterpea might help creative musicians
  – How learning computer music is a good way to learn computer programming (and vice versa)
A Simple Example

- First two bars expressed in Euterpea:
  
  \[
  (c \, 4 \, \text{qn} \,:+; \, d \, 4 \, \text{qn} \,:+; \, e \, 4 \, \text{qn} \,:+; \, f \, 4 \, \text{qn} \,:+; \\
  g \, 4 \, \text{qn} \,:+; \, f \, 4 \, \text{qn} \,:+; \, e \, 4 \, \text{qn} \,:+; \, d \, 4 \, \text{qn}) \, := \;
  (e \, 4 \, \text{qn} \,:+; \, f \, 4 \, \text{qn} \,:+; \, g \, 4 \, \text{qn} \,:+; \, a \, 4 \, \text{qn} \,:+; \\
  b \, 4 \, \text{qn} \,:+; \, a \, 4 \, \text{qn} \,:+; \, g \, 4 \, \text{qn} \,:+; \, f \, 4 \, \text{qn})
  \]

-Verbose, but accurate.
- Can we do better?
- Yes, using various forms of abstraction.
Abstraction 101

\[
\begin{align*}
(c &\ 4\ qn :+ d\ 4\ qn :+ e\ 4\ qn :+ f\ 4\ qn :+ g\ 4\ qn :+ f\ 4\ qn :+ e\ 4\ qn :+ d\ 4\ qn) := \\
&\ (e\ 4\ qn :+ f\ 4\ qn :+ g\ 4\ qn :+ a\ 4\ qn :+ b\ 4\ qn :+ a\ 4\ qn :+ g\ 4\ qn :+ f\ 4\ qn)
\end{align*}
\]

• **Using data abstraction:**
  \[
  \text{line } [c\ 4\ qn, d\ 4\ qn, e\ 4\ qn, f\ 4\ qn, g\ 4\ qn, f\ 4\ qn, e\ 4\ qn, d\ 4\ qn] := \\
  \text{line } [e\ 4\ qn, f\ 4\ qn, g\ 4\ qn, a\ 4\ qn, b\ 4\ qn, a\ 4\ qn, g\ 4\ qn, f\ 4\ qn]
  \]

• **Using functional abstraction:**
  \[
  \text{line } (\text{map fn } [c, d, e, f, g, f, e, d]) := \text{line } (\text{map fn } [e, f, g, a, b, a, g, f])
  \]
  \[
  \text{where fn } n = n\ 4\ qn
  \]

• **Using higher-order functions:**
  \[
  \text{gn } [c, d, e, f, g, f, e, d] := \text{gn } [e, f, g, a, b, a, g, f]
  \]
  \[
  \text{where gn } = \text{line . map fn; fn } n = n\ 4\ qn
  \]

Haskell Equivalence

• Q: How can we justify these simplifications?
• A: Using *equational reasoning*, also called *proof by calculation*.

• Requires only three axioms:
  – line [m1, m2, …, mn] = m1 :+: m2 :+: … :+: mn
  – map fn [x1, x2, …, xn] = [fn x1, fn x2, …, fn xn]
  – (f . g) x = f (g x)

• Complete novices can grasp this easily – no more difficult than high-school algebra.
• (We also “reveal” the definitions of line and map, along with inductive proofs of the above “axioms.”)
Simple Example Revisited

• Here’s another way to write the example
  *(polyphonic* interpretation, rather than *contrapuntal*):

  \[
  \begin{align*}
  (c\ 4\ qn\ &::\ e\ 4\ qn)\ ;\ (d\ 4\ qn\ &::\ f\ 4\ qn)\ ;\ (e\ 4\ qn\ &::\ g\ 4\ qn)\ ;\\
  (f\ 4\ qn\ &::\ a\ 4\ qn)\ ;\ (g\ 4\ qn\ &::\ b\ 4\ qn)\ ;\ (f\ 4\ qn\ &::\ a\ 4\ qn)\ ;\\
  (e\ 4\ qn\ &::\ g\ 4\ qn)\ ;\ (d\ 4\ qn\ &::\ f\ 4\ qn)\end{align*}
  \]

• It can likewise be simplified, in two steps:

  line \[c\ 4\ qn\ ::\ e\ 4\ qn,\ d\ 4\ qn\ ::\ f\ 4\ qn,\ e\ 4\ qn\ ::\ g\ 4\ qn,\]
  \[f\ 4\ qn\ ::\ a\ 4\ qn,\ g\ 4\ qn\ ::\ b\ 4\ qn,\ f\ 4\ qn\ ::\ a\ 4\ qn,\]
  \[e\ 4\ qn\ ::\ g\ 4\ qn,\ d\ 4\ qn\ ::\ f\ 4\ qn]\n
  line (map fn [(c,e), (d,f), (e,g), (f,a), (g,b), (f,a), (e,g), (d,f)])

  where \(fn\ (n1,n2) = n1\ 4\ qn\ :: n2\ 4\ qn\)
Euterpean Equivalence

- But in what sense are the contrapuntal and polyphonic versions *equivalent*?
- Note: they are *not* equivalent as Haskell values.
- But they *sound* the same when “played”.
- Thus we need a notion of *interpretation*, or *performance*.
- Define a *performance* to be a sequence of *events*.
- The event $\text{Event } t \ i \ p \ d \ v$ captures the fact that at time $t$ instrument $i$ sounds pitch $p$ for a duration $d$ with volume $v$. 
Performance

• The function \textit{perform} does the desired interpretation:

\[
\text{perform} :: \text{Context} \rightarrow \text{Music} \rightarrow \text{Performance}
\]

• \textbf{Definition:} Two music values \( m_1 \) and \( m_2 \) are \textit{equivalent}, written \( m_1 \equiv m_2 \), if and only if:

\[
( \forall c) \text{perform} \ c \ m_1 = \text{perform} \ c \ m_2
\]

• In other words:

“\text{if two things sound the same, they are the same}”

• For example:

For all \( m_1, m_2, \) and \( m_3 \):

\[
(m_1 :+: m_2) :+: m_3 \equiv m_1 :+: (m_2 :+: m_3)
\]
An Algebra of Music

• There are eight axioms that comprise an algebra of music.
• For example, (:=:) is associative and commutative.
• Another (important) example:
  Duality of (:+:) and (:=:)
  For any $m_0, m_1, m_2$, and $m_3$ such that $\text{dur } m_0 = \text{dur } m_2$:
  $(m_0 :+: m_1) :=: (m_2 :+: m_3) \equiv (m_0 :=: m_2) :+: (m_1 :=: m_3)$
• Each axiom is provably sound.
• The axiom set is also complete: If two music values are equivalent, they can be proven so using only the eight axioms.
• Furthermore, the algebra can be made polymorphic: it is valid for video, audio, animation, even dance.
• The Eight Laws of Polymorphic Temporal Media.
Puttin’ the “funk” back in functional programming.
da Funk Master
Paul Hudak at Yale

Available now at your neighborhood cafepress.com...
Self-Similar Music

• Begin with a simple melody of n notes.
• Now duplicate that melody n times, playing each in succession, but first perform these transformations:
  – transpose the ith melody proportionally to the pitch of the ith note in the original melody, and
  – scale its tempo proportionally to the duration of the ith note.
• Apply this process infinitely often, yielding an infinitely dense melody of infinitesimally shorter notes.
• To make the result playable, stop the process at some pre-determined level.
Example of Self-Similar Music
Self-Similar Music in Euterpea

• We could write an inductive mathematical definition of the tree.
• But we might as well use Euterpea instead:

```haskell
data Cluster = Node SNote [Cluster]    -- a Rose Tree
type SNote = (Dur, AbsPitch)

selfSim :: [SNote] -> Cluster
selfSim pat = Node (0,0) [mkClust p | p <- pat]
    where mkClust note =
            Node note [mkClust (addMul note p) | p <- pat]

addMul (d0,p0) (d1,p1) = (d0*d1, p0+p1)

fringe :: Int -> Cluster -> [SNote]
fringe 0 (Cluster note cls) = [note ]
fringe n (Cluster note cls) = concatMap (fringe (n-1)) cls
```
Example

• Use this 4-note motif as the seed:

• Traverse 4 levels in the tree.

• Play together with itself in reverse and transposed down one octave:

\[ m := \text{transpose} (-12) (\text{revM} \ m) \]
Other Approaches to Self-Similarity

• Fractals
• Context-free grammars
• L-Systems

• All easily expressed in Euterpea (and discussed in my book)
Musical User Interface (MUI)

• Design philosophy:
  – GUI’s are important!
  – The dataflow metaphor (“wiring together components”) is powerful!
  – Yet graphical programming is inefficient...

• Goal: MUI widgets using dataflow model at the linguistic level.

• We achieve this via two levels of abstraction:
  – The *UI Level*
    • Create widgets, attach titles, labels, etc.
    • Control layout
  – The *Signal Level*
    • A signal is conceptually a *continuous* (time-varying) value.
    • Sliders, knobs, etc. are *input* widgets.
    • Midi out, graphics, etc. are *output* widgets.
Signals

• Signals are *time-varying quantities*.
• Conceptually they can be thought of as functions of time:
  \[ \text{Signal } a = \text{Time } \rightarrow a \]
• For example, output of a slider is a time-varying *number*.
• Key idea: Lift all static functions to the signal level using a family of *lifting functions*:
  \[
  \begin{align*}
  \text{lift0} &: a \rightarrow \text{Signal } a \\
  \text{lift1} &: (a \rightarrow b) \rightarrow (\text{Signal } a \rightarrow \text{Signal } b) \\
  \text{lift2} &: (a \rightarrow b \rightarrow c) \rightarrow (\text{Signal } a \rightarrow \text{Signal } b \rightarrow \text{Signal } c)
  \end{align*}
  \]
• Haskell’s type classes make this especially easy.
• Conceptually:
  \[
  \begin{align*}
  s_1 + s_2 &= \lambda t \rightarrow s_1 t + s_2 t \\
  \sin s &= \lambda t \rightarrow \sin (s t)
  \end{align*}
  \]
• One can also *integrate* signals.
Events

• Signals are not enough... some things happen *discretely*.
• *Events* can be realized as a kind of signal:
  
  ```haskell
  data Maybe a = Nothing | Just a
  type EventS a = Signal (Maybe a)
  ```

• So events are actually *event streams*.
• Midi event streams simply have type:
  
  ```haskell
  EventS [MidiMessage]
  ```

  where *MidiMessage* encodes standard Midi messages such as Note-On, Note-Off, etc.
• Pitch translator:

\[
\text{do } ap \leftarrow \text{title "Absolute Pitch" (hiSlider 1 (0, 100) 0)} \\
\text{title "Pitch" (display (lift1 (show \circ pitch) ap))}
\]

• Output Midi note at constant rate:

\[
\text{do } ap \leftarrow \text{title "Absolute Pitch" (hiSlider 1 (0, 100) 0)} \\
f \leftarrow \text{title "Tempo" (hSlider (1, 10) 1)} \\
t \leftarrow \text{time} \\
\text{let ticks = timer t (1/f)} \\
\text{let ns = snapshot ticks ap =>> (λk → [ANote 0 k 100 0.1])} \\
midiOut 0 ns
\]
Bifurcate Me, Baby!

- Consider the recurrence equation:
  \[ x_{n+1} = r \times x_n \times (1 - x_n) \]
  Start with an initial population \( x_0 \) and iteratively apply the growth function to it, where \( r \) is the growth rate. For certain values of \( r \), the population stabilizes, but as \( r \) increases, the period doubles, quadruples, and eventually leads to chaos. It is one of the classic examples in chaos theory.
- In Euterpea:
  
  \[ \text{grow r x = r \times x \times (1-x)} \]
Then wrap it in a MUI

bifurcate = runMUI (300,500) "Bifurcate!" (do
  t <- time
  mo <- selectOutput
  f <- title "Frequency" (withDisplay (hSlider 1 10 1))
  r <- title "Growth rate" (withDisplay (hSlider 2.4 4.0 2.4))
  let ticks = timer t (1.0 / f)
      pop   = accum 0.1 (snapshot ticks (lift1 grow r))
  title "Population" (display' pop)
  midiOut mo (snapshot ticks pop =>> popToNote)

popToNote x = [ANote 0 n 64 0.05] where n = truncate (x*127)
Taking the Signal Metaphor a Step Further

• Sound synthesis and audio processing
  – “vertical” language design
  – “from signals to symphonies”
• Key idea: a signal function
• The type \( SF \ Clk \ In \ Out \) is the type of signal function that maps signals of type \( In \) to signals of type \( Out \), at abstract clock rate \( Clk \).
Key Idea

• This signal processing diagram:

\[ Y \leftarrow \text{sigfun} \rightarrow x \]

• is equivalent to this Euterpean code, using “arrow” syntax:

\[ y \leftarrow \text{sigfun} \leftarrow x \]
Physical Model of a Flute

- Physical Model of a Flute
- feedbk1
- lowpass
- Embouchure delay
- delayt
- (1/fqc/2)
- emb
- Flute bore delay
- delayt (1/fqc)
- sum1
- * feedbk1
- * feedbk2
- /x/x
- lowpass
- out
- * amp
- returnA
- lineSeg
Flute Model in Euterpea

```plaintext
flute dur freq amp vfreq =
in proc () -> do
  amp1 <- linseg ... <- ()
  amp2 <- linseg ... <- ()
  ampv <- linseg ... <- ()
  flow <- rand 1 <- amp1
  vibr <- oscils vfreq <- 0.1 * ampv
  rec
    let feedbk = body * 0.4
    body <- delay (1/freq) <- out
    x <- delay (1/freq/2) <- breath*flow
       + amp1 + vibr + feedbk
    out <- tone <- (x - x*x*x + feedbk, 2000)
returnA <- out*amp*amp2
```

Flute Demo

- f0 and f1 demonstrate the change in the breath parameter.
  
  \[
  \begin{align*}
  f0 &= \text{flute } 3 \ 0.35 \ 440 \ 0.93 \ 0.02 \\
  f1 &= \text{flute } 3 \ 0.35 \ 440 \ 0.83 \ 0.05 \\
  \end{align*}
  \]

- f2 has a weak pressure input so only plays the blowing noise.
  
  \[
  f2 = \text{flute } 3 \ 0.35 \ 440 \ 0.53 \ 0.04
  \]

- f3 takes in a gradually increasing pressure signal.
  
  \[
  f3 = \text{flute } 4 \ 0.35 \ 440 \\
  (\text{lineSeg } [0.53, \ 0.93, \ 0.93] \ [2, \ 2]) \\
  0.03
  \]

- Sequence of notes:
Implementing Signals

- As “yet another abstraction”, arrows might seem to introduce yet more computational overhead.
- But in fact arrows eliminate certain important classes of space leaks.
- Furthermore, causal commutative arrows (CCA) can improve performance dramatically:
  - Any CCA expression can be normalized into an expression with one loop and one vector of state variables.
  - This results in a factor of 50 improvement over conventional implementation methods.
  - Good enough for non-trivial real-time performance.
The Sky is the Limit

• In Music:
  – Can we create a robotic conductor?
  – What does a saxophone the size of a house sound like?
    Or a clariphone or saxonet?
  – Can we animate a new choreography?
  – Can we create new forms of artistic expression?
  – Can a computer compose music that passes the Turing Test?

• In Computer Science:
  – Can we parallelize all this?
  – What is the essence of “functional reactive programming”?
  – Is there a better syntax for Causal Commutative Arrows?
  – How can we accommodate multiple clock rates?
“Bottles”

• Composed and implemented by Donya Quick, a grad student in CS
• Done entirely in Euterpea
• Demonstrates:
  – High-level composition
    • Notes, repetitions, structure
    • Micro-tonal support
  – Low-level signal processing
    • Both “struck” bottle and “blown” bottle sounds, using physical modeling
    • Some additive synthesis as well
    • Reverb and audio processing
Thank You!!

(any questions?)