**Institut für Informatik**

Naturwissenschaftliche Fakultät III
**Agrar-, Geowissenschaften, Mathematik und Informatik**

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

Henning Thielemann

2011-05-06

Martin-Luther-Universität Halle-Wittenberg

# Motivation

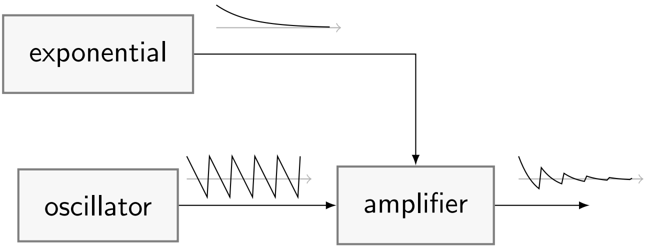
I want to program music:

- design algorithmic music patterns
  encode ideas rather than particular musical events
- break barrier between notes and audio signal
  effects like reversed music or retarded record player
- real-time and interactive
- declarative, reusable, with error-prevention
- integration with non-audio parts
- ... and often I prefer text editors, search&replace with regular expressions, text-based version management
  to clicking through multiple layers of graphical dialogues

# Haskell

- Functional non-strict programming paradigm
  → *direct translation of signal flow*
- General purpose programming language
  → *ready for other tasks than audio*
- Statically polymorphically typed (Hindley-Milner system)
  → *error prevention*
- Type classes: automatic adaption to specific types
  → *reusable code*
- Compiled language
  → *efficient*
- Interactive programming
  → *live coding*

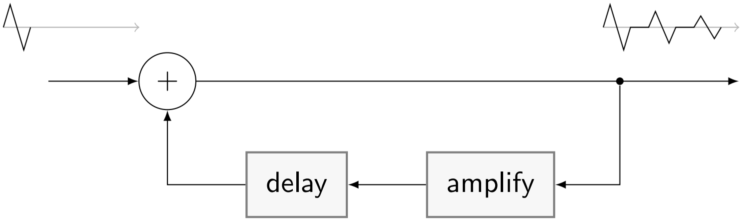
# Functional paradigm: Expression tree




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

Lazy evaluation: structure code logically, compute chronologically

# Functional paradigm: Feedback




```
let output =
      input +
      delay time (amplify gain output)
in  output
```

# Haskell problems

In principle

*compiled = efficient*

but

- lazy evaluation comes at high run-time costs,
- optimizer too often misses optimization opportunities,
- optimizer not available in interactive programming,
- not yet support for vector computing (SSE, AltiVec).

No problems in principle, but problems in current implementations.

# Embedded Domain Specific Language

- Domain Specific Language = Special Purpose Language
- Embedded = Use expressions of a host language

The expression

$$a + b$$

- does not mean "add a and b"
- but instead:
  "generate an addition command in another language"

# Music EDSLs for Haskell

EDSLs for Haskell exist for

- SuperCollider
- Csound
- ...

We use Low-Level Virtual Machine LLVM

- Compiler back-end – "portable high-level assembler"
- Just-In-Time compiler
  → *tight integration with Haskell code*
- register allocation
- wide range of optimizations
- vector computing
- processor specific instructions

# Examples for interactive programming

```
playMono (Gen.osci Wave.sine 0 (hertz 440))

playMono
   (Gen.exponential2 (second 1) 1 *
    Gen.osci Wave.triangle 0 (hertz 440))

playStereo
   (liftA2 Stereo.cons
      (Gen.osci Wave.triangle 0 (hertz 439))
      (Gen.osci Wave.triangle 0 (hertz 441)))
```

# What happens?

- `Gen.osci`, `Gen.exponential` generate LLVM loop bodies
- "`*`" combines existing LLVM loop bodies
- `playMono` closes the loop, runs the code and feeds generated signal data to the audio output

- See a disassembled LL file
- See a generated X86 assembly file

# Types of signal generators

```
exponential2 ::
   Float -> Float ->
   Generator (Value Float)
exponential2 halfLife initialValue = ...

osci ::
   (Value Float -> Code y) ->
   Float -> Float -> Generator y
osci wave phase freq = ...
```

Note:

- Higher order functions for waves
- Types prevent confusing mono with stereo signals

# Change parameters without re-compilation

Problem 1:

- Playing the same instrument at different pitches requires recompilation

Solution:

- Maintain a record of parameters
  for exchange between LLVM code and Haskell
- Turn instrument arguments into record selectors
- Constant parameters still hard-wired into LLVM code
- Parameters still expressed by number literals

# Causality

Problems 2a-c:

- Sharing
  `input + delay input` means,
  that `input` is computed twice
- Feedback
  **`let`** `comb = input + delay comb` **`in`** `comb`
  does not work
- Causal processes for real-time processing
  e.g. as needed for JACK

"Causal": every output sample depends exclusively on present and past input samples

# Causal Arrows

Turn

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

into

```
Causal a b
```

- Solve Sharing, Feedback, Causality problems
- Composition of causal arrows maintains causality
- Instead of `delay (amplify sig)`
  write `(delay . amplify) $* sig`
- Multiple input and output: `Causal (a,b) (c,d)`
- Haskell provides special arrow syntax

# Arrows

- Arrows generalize functions
- many applications including hardware design and parsers
- underlying concept of FAUST

# Coping with filter parameters

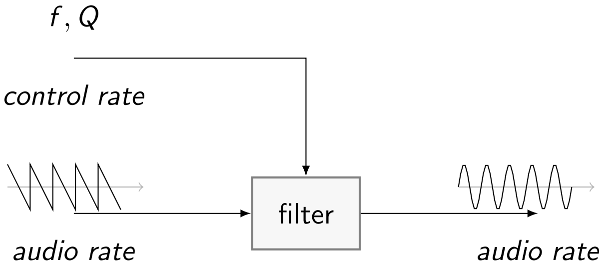
Problem 3:

- Frequency filters controlled by frequency $f$, resonance $Q$
- Computing internal filter parameters from $f, Q$ is expensive, but filter parameters may not change quickly
- Applying filters is cheap, but must be performed at audio sample rate

Solution: Separate

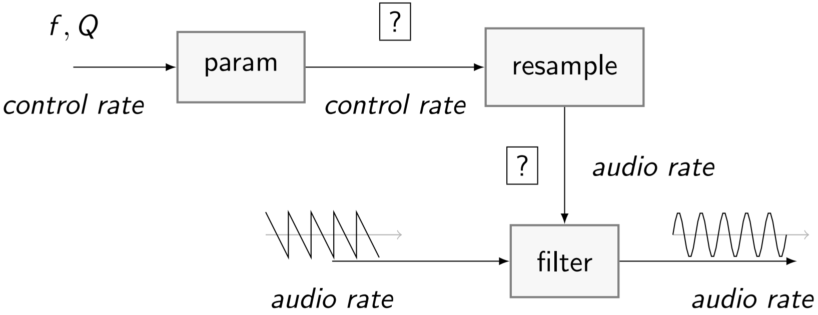
- filter parameter computation,
- rate adaption,
- filter application

# Coping with filter parameters: other programs




- Csound, SuperCollider:
  Distinguish between control rate and audio rate
- ChucK: Update parameters on demand

# Coping with filter parameters: our solution




- opaque internal filter parameters: ?

# Coping with filter parameters

- Filter parameter computation:
  select filter type, generate opaque filter parameter type
- Stretch signal of filter parameters
- Type-class selects filter corresponding to filter parameter type

Advantages:

- Filter works exclusively at audio sampling rate (simple!)
- Different ways of specifying filter parameters
- Different control rates in the same program
- Irregular control rates,
  e.g. compute filter parameters if MIDI knob is turned

# Vectorization

Problem 4:

- SSE and AltiVec allow vector operations
  like parallel multiplication of four pairs of **`Float`** numbers
- How to support these operations?
- What to share between scalar and vector implementation?

Solution:

- Divide signal into chunks of vector size
- New type of samples: `Vector`
- Re-use signal generator and arrow types
- Full automatic vectorisation impossible,
  because user has to accept compromises
- Type-classes reduce code duplication

# Expressiveness of Haskell's types

sample types

- `Value Float`, `Value Double` — samples of various precisions
- `Stereo (Value a)` — ...or quadro, surround
- `Complex (Value a)` — Fourier coefficients
- `Value Bool` — for gate signals
- `Value Int32` — for counters
- `Moog.Parameter D8 (Value a)` — filter parameters
- `Value (LLVM.Array D6 (Order2.Parameter a))`
- `Value (Vector D4 a)` — vectorized signal
- `DimensionNumber Time (Value a)` — physical quantities
- `Value a -> Code (Value b)` — each sample is a waveform
- combinations of type constructors
- custom types like `newtype Cmp = Cmp (Value Int8)`

# Expressiveness of Haskell's types

generator and process types

- `Generator (a,b)`
  Two synchronous signal generators
- `Causal a b`
  type `b` samples depend causally on type `a` samples
- `Generator a -> Generator b`
  type `b` samples depend non-causally on type `a` samples
- `stateVariableFilter :: Causal (Param,a) (a,a,a)`
  causal process with multiple inputs and outputs
- `frequencyModulation :: Generator a -> Causal t a`
  output samples (type `a`) depend causally on frequency control (type `t`) but non-causally on input samples (type `a`)

# Expressiveness of Haskell's types

type classes

- Share code between scalar and vector code
- Share rate handling between filters
- Use number literals and arithmetic operators for parameters, signals, causal processes.

# Real-time software synthesizer

- Compile code for instruments at startup
- Receive MIDI events via ALSA sequencer
- Emit signal stream via ALSA PCM
- Vector computation, filter parameter update on controller changes, react to program changes . . .

# Conclusions

- Haskell is ultimately cool
- really
- I swear
- makes you look like a wizard
- everything else is toy

# Conclusions

Embedded Domain Specific Language

- more low-level control, less declarativity
- we get: general purpose, type-safety
- designing an EDSL has its own problems

# Outlook

- make it nicer
- cleaner
- more intuitive to use
- avoid memory leaks
- tune garbage collector

Get it from

*http://code.haskell.org/synthesizer/llvm/*

# Questions

How to configure a Linux machine, such that

- it starts as few as possible things,
- starts a software synthesizer,
- automatically connects a plugged USB keyboard to that soft-synth?

wanted:

- ALSA oscilloscope
- Audacity configurable for presentations:
  thick lines, better contrast