Efficient signal processing using Haskell and LLVM

Henning Thielemann

2016-09-15

Introduction

Warning:

- `synthesizer-llvm` – already fast and feature-rich, but still pretty low-level.
- Nonetheless no need for much LLVM hacking.

Signal

`module Synthesizer.LLVM.Simple.Signal`

- signal simulated by signal generator
- compute and emit samples step by step (iterator)
- `Value Float` ∼ **`Float`** value in LLVM
- `Signal.T a` ∼ `[a]`

producer:

```
osciSaw :: Float -> Signal.T (Value Float)
```

modifier:

```
amplify ::
   Float ->
   Signal.T (Value Float) ->
   Signal.T (Value Float)
```

Oscillator

supported waveforms:

- Csound: waves made from tables
- SuperCollider: specialized oscillator per waveform
- `synthesizer-llvm`: any function as waveform
  `Synthesizer.LLVM.Wave`

band-limited oscillators:

- SuperCollider: available
- `synthesizer-llvm`: not available

Causal processes

Problems:

```
let x = Signal.osciSaw freq
in  x + x
```

Signal `x` is computed twice.

```
amplify ::
   Float ->
   Signal.T (Value Float) ->
   Signal.T (Value Float)
```

No warranty for usability of `amplify` in real-time processing.

Causal processes

Solution: `module Synthesizer.LLVM.Causal.Process`

- `Process.T a b` ∼ `Signal.T a -> Signal.T b`
- `instance Arrow Process.T`
- warrants causality: never accesses future input values
- e.g. `reverse` cannot be a `Process.T`
- tailored to real-time processing
- allows for sharing
- allows for feedback

Example:

```
amplify ::
   Float ->
   Process.T (Value Float) (Value Float)
```

Parameters

Problem:

```
Test> play (osciSaw (hertz 440))
...
Test> play (osciSaw (hertz 550))
...
```

Code for `osciSaw` is compiled twice.

Goal:

- compile `osciSaw` once
- add parameters to compiled code

Parameters

Solution:

```
module Synthesizer.LLVM.Parameter
module Synthesizer.LLVM.Parameterized.*
module Synthesizer.LLVM.CausalParameterized.*
```

Example:

```
amplify ::
   Param.T p Float ->
   CausalP.T p (Value Float) (Value Float)
```

- p: record containing all parameters
- `Param.T p Float`: selector from record p
- `arr fst :: Param.T (Float, Bool) Float`
- `440 :: Param.T p Float`:
  - constant 440 folded into code
  - parameter omitted in low-level parameter structure

Parameters

Example:

```
Causal.applyStorable
   (Causal.amplify (arr id))
      :: Float -> SV.Vector Float ->
                  SV.Vector Float
```

Rich sample value types

- Csound, SuperCollider: Signals of Floats
- `synthesizer-llvm`:
  - various precisions: **`Float`**, **`Double`**
  - integers (counts, linear congruence noise)
  - **`Bool`** (trigger and gate signals)
  - enumerations (comparison result)
  - stereo pairs
  - tuples (combined low-pass, band-pass, high-pass filter)
  - complex numbers (Fourier coefficients)
  - arrays (ring buffers, parallel processes)
  - serial chunks (vectorization)
  - opaque records (internal filter parameters)
  - functions (waveform)

Stereo

```
module Synthesizer.LLVM.Frame.Stereo
module Synthesizer.LLVM.Frame.StereoInterleaved
```

```
amplifyStereo ::
   a ->
   Causal.T
      (Stereo.T (Value a))
      (Stereo.T (Value a))
```

No need to resort to pairs of signals.

Ugly:

```
CausalP.takeWhile
   (LLVM.cmp LLVM.CmpGT) threshold
```

Nice:

```
CausalPV.takeWhile (%>) threshold
```

```
module Synthesizer.LLVM.Simple.Value
module Synthesizer.LLVM.Causal.ProcessValue
module Synthesizer.LLVM.CausalParameterized.ProcessValue
```

1 Features

- Signal producers and modifiers
- Causal processes: sharing and feedback
- Parameterized code
- Sample value types: Stereo sounds, binary logic signals
- **Frequency filter parameters and different signal rates**
- Vectorization
- Treat arrows like plain functions
- Compose music from tones
- MIDI control
- Integration with ALSA and JACK

2 Discussion

Frequency filter

Problem:

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

Solutions elsewhere:

- Csound, SuperCollider distinguish between:
  - high audio rate: e.g. 44100 Hz
  - low control rate: e.g. 4410 Hz
  - audio rate must be multiple of control rate
- ChucK: Update parameters on demand

Coping with internal filter parameters

module `Synthesizer.LLVM.Filter.*`

Separate

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

subsumes features of other frameworks

- filter parameters: explicit but opaque data type
- automatically select filter depending on filter parameter type:
  module `Synthesizer.LLVM.Causal.Controlled`
- dependency this way:
  - multiple ways to define filter
  - one way to perform filter

Vector computation

- modern CPUs can perform multiple operations at once,
  AVX: 8 **`Float`** multiplications in one instruction
- certainly the main way to accelerate code in future processors

utilize vector operations:

- LLVM optimizer:
  turn scalar into vector operations automatically
- custom `synthesizer-llvm` implementations
- LLVM: both generic and processor specific vector instructions
- supports non-native vector sizes

LLVM optimizer:

- Pro: transparent to `synthesizer-llvm` API
- Con: not allowed to perform certain optimizations

Custom vector implementations

possible schemes:

- serial: chop signal in chunks of vector size
- parallel: compute several equal processes in lock-step
- mixed: e.g. serial chunks of stereo signals
- pipeline: chain of equal processes

- switch between vectorization schemes: expensive
  $\rightarrow$ stick to one scheme
- serial vectorization most flexible
  automatically scales to (future) longer vectors

Custom vector implementations

- serial chunks:

```
module Synthesizer.LLVM.Frame.SerialVector
module Synthesizer.LLVM.Simple.SignalPacked
module Synthesizer.LLVM.Parameterized.SignalPacked
module Synthesizer.LLVM.Causal.ProcessPacked
module Synthesizer.LLVM.Causal.ControlledPacked
module Synthesizer.LLVM.CausalParameterized.ProcessPack
module Synthesizer.LLVM.CausalParameterized.ControlledP
```

- parallel: replace `Value a` by `Value (Vector n a)`
- mixed serial/parallel:

```
module Synthesizer.LLVM.Frame.StereoInterleaved
```

- pipeline: `Synthesizer.LLVM.Causal.Process.pipeline`

1 Features
- Signal producers and modifiers
- Causal processes: sharing and feedback
- Parameterized code
- Sample value types: Stereo sounds, binary logic signals
- Frequency filter parameters and different signal rates
- Vectorization
- **Treat arrows like plain functions**
- Compose music from tones
- MIDI control
- Integration with ALSA and JACK

2 Discussion

Arrows are cumbersome

Functional:

```
\x -> let y = lowpass x
      in  mix y (delay y)
```

Temporary variables for shared results – Wanted!

Arrow combinators:

```
mix <<< id &&& delay <<< lowpass
```

Too few temporary variables (no x)

Arrow notation:

```
proc x -> do
   y <- lowpass -< x
   z <- delay -< y
   mix -< (y,z)
```

Too many temporary variables (unnecessary z).

Turn Arrows to functions

module Synthesizer.LLVM.CausalParameterized.Functional

```
let x = Func.lift $ arr id
    y = lowpass $& x
in  mix $& y &|& (delay $& y)
```

```
Func.withArgs $ \x ->
   let y = lowpass $& x
   in  mix $& y &|& (delay $& y)
```

- Input selector instead of function parameter:
  `x :: Func.T input (Value Float)`
- Observed sharing
  `y` runs only once

1 Features

- Signal producers and modifiers
- Causal processes: sharing and feedback
- Parameterized code
- Sample value types: Stereo sounds, binary logic signals
- Frequency filter parameters and different signal rates
- Vectorization
- Treat arrows like plain functions
- Compose music from tones
- MIDI control
- Integration with ALSA and JACK

2 Discussion

Compose tones

- parameterizable signals
- render to `StorableVector`
- overlapping mix of scheduled signals
  `Synthesizer.LLVM.Storable.Signal.arrange`
- result: `StorableVector`
  accessible to further processing

- separate MIDI channels
- separate command types (note, controller, program change)
- separate controllers
- convert controller events to audio data
  or opaque filter parameters

```
module Synthesizer.LLVM.MIDI
module Synthesizer.LLVM.MIDI.BendModulation
```

Integration with ALSA and JACK

ALSA:

- separate sub-systems for
  - Audio: ALSA PCM
  - MIDI: ALSA sequencer

JACK:

- call-back design
  compatible with `Causal.Process`
- processes chunks of audio and MIDI data
- reactive audio programming
- Event+Behavior: MIDI events
- integration of Event+Behavior with audio

Integration with ALSA and JACK

- process data in chunks
- `CausalParameterized.Process.processIO`

```
module Synthesizer.LLVM.Server.ALSA
module Synthesizer.LLVM.Server.JACK
```

Signal processing EDSL in Haskell

An EDSL in Haskell as cumbersome and unsafe as C –
any advantage over C?
Advantages:

- automatic adaption to instruction set extensions (e.g. SSE, AVX, AVX2)
- put much processing in one loop
  - does not increase speed,
  - but allows for short-time feedback
- generation of efficient signal processing including short-time feedback at runtime, e.g. also at user-request. User may
  - enter custom process graphs,
  - load example graphs from disk,
  - send it via MIDI-SYSEX to your software synthesizer.

Comparison with Csound, SuperCollider etc.

Advantages of established software synthesizers:

- lots of predefined effects and examples

Disadvantage: Also need sophisticated Haskell interfaces.

Advantages of Haskell EDSL:

- exchange audio data between LLVM synthesizer and other Haskell code in memory
- smaller, more basic building blocks,
  due to richer type system and short-time feedback
- thus, easier to extend

Short-time feedback

Short-time feedback is a pretty invasive feature.
The fine print is:

- short-time feedback makes processing unstable,
  hard to predict,
  may not be reproducible at different sampling rates
- conflicts with vectorization,
  machine vectors are the new processing chunks

LLVM

Pros:

- JIT
- multiple processor back-ends
- vectorization
- optimizations

Cons:

- global variables
  (e.g. no connection between Module and Builder)
- destructive updates (e.g. in optimization)
- Phi-Nodes instead of Basic-Block-Arguments
- low responsibility:
  - frequent changes, hardly documented
  - little reactions to questions
  - bugs are quickly introduced but require years to be fixed
    (e.g. inttopointer, LLVMRunFunction)

To Do

- vectorization without vector in API types
    - vectorized Signal and Process type
    - custom LLVM vectorization pass
- storable-vector with typed chunk size
- signal with sample rate as type
- dimensional discrete time
- test mode for LLVM monad
  for virtual downgrade of the machine
- better integration with
  a reactive Haskell programming framework

Remaining technical difficulties

- Optimizations interfere badly with call-backs
  Call-backs are needed for
  - allocation and deallocation,
  - reading from lazy data structures.
- Crashes are hard to debug