

The Shepard Tone and Higher-Order Multi-rate Synchronous Data-Flow Programming in SIG

► Baltasar Trancón y Widemann Markus Lepper

Ilmenau University of Technology

semantics GmbH

FARM

2015-09-05

Agenda

- 1 Exposition**
 - Main Theme: SIG in a Nutshell
 - Countertheme: The Shepard Tone
- 2 Development**
 - Higher-Order Stream Programming
 - Multi-rate Stream Programming
- 3 Recapitulation**
 - Putting Things Together
 - Conclusion

Spoiler

- Synchronous Functional Data-Flow Language
 - Similar but not quite FRP
 - New features under development
- Classical sound construction
 - Small but nontrivial synthesis problem
 - Ideal application of new features

Spoiler

- Synchronous Functional Data-Flow Language
 - Similar but not quite FRP
 - New features under development
- Classical sound construction
 - Small but nontrivial synthesis problem
 - Ideal application of new features

Agenda

- 1 Exposition**
 - Main Theme: SIG in a Nutshell
 - Countertheme: The Shepard Tone
- 2 Development**
 - Higher-Order Stream Programming
 - Multi-rate Stream Programming
- 3 Recapitulation**
 - Putting Things Together
 - Conclusion

Context: The SIG Language

Total Functional no effects, events, recursion

Clocked Synchronous variables are mutated regularly

Data Flow control flow by data dependency

- Real-time execution model
 - pull-based, ultra-low latency
- Applications: embedded control, simulation, **audio**, ...
- Backends: JVM (C, DSP, FPGA, ...)
- Layered design
 - Core language with *simple* compositional semantics
 - Functional frontend: ADTs, pattern matching, **higher order**
 - Advanced features: physical units, **multi-rate**
- Semantics of higher layers by transformation

Context: The SIG Language

Total Functional no effects, events, recursion

Clocked Synchronous variables are mutated regularly

Data Flow control flow by data dependency

- Real-time execution model
 - pull-based, ultra-low latency
- Applications: embedded control, simulation, **audio**, ...
- Backends: JVM (C, DSP, FPGA, ...)
- Layered design
 - Core language with *simple* compositional semantics
 - Functional frontend: ADTs, pattern matching, higher order
 - Advanced features: physical units, multi-rate
- Semantics of higher layers by transformation

Context: The SIG Language

Total Functional no effects, events, recursion

Clocked Synchronous variables are mutated regularly

Data Flow control flow by data dependency

- Real-time execution model
 - pull-based, ultra-low latency
- Applications: embedded control, simulation, **audio**, ...
- Backends: JVM (C, DSP, FPGA, ...)
- Layered design
 - Core language with **simple** compositional semantics
 - Functional frontend: ADTs, pattern matching, **higher order**
 - Advanced features: physical units, **multi-rate**
- Semantics of higher layers by transformation

Context: The SIG Language

Total Functional no effects, events, recursion

Clocked Synchronous variables are mutated regularly

Data Flow control flow by data dependency

- Real-time execution model
 - pull-based, ultra-low latency
- Applications: embedded control, simulation, **audio**, ...
- Backends: JVM (C, DSP, FPGA, ...)
- Layered design
 - Core language with **simple** compositional semantics
 - Functional frontend: ADTs, pattern matching, **higher order**
 - Advanced features: physical units, **multi-rate**
- Semantics of higher layers by transformation

Basic Example: Cumulative Sum

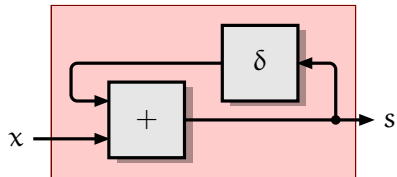
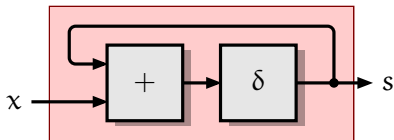
$$\left[\begin{array}{l} x \rightarrow s \\ s := 0 \ ; \ (s + x) \end{array} \right]$$

$$\left[\begin{array}{l} x \rightarrow s \\ s := (0 \ ; \ s) + x \end{array} \right]$$

Basic Example: Cumulative Sum

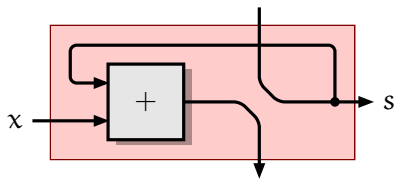
$$\left[\begin{array}{l} x \rightarrow s \\ s := 0 \ ; \ (s + x) \end{array} \right]$$

$$\left[\begin{array}{l} x \rightarrow s \\ s := (0 \ ; \ s) + x \end{array} \right]$$

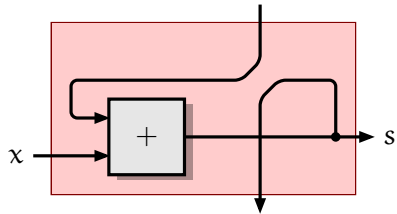


Basic Example: Cumulative Sum

$$\left[\begin{array}{l} x \rightarrow s \\ s := 0 \ ; \ (s + x) \end{array} \right]$$



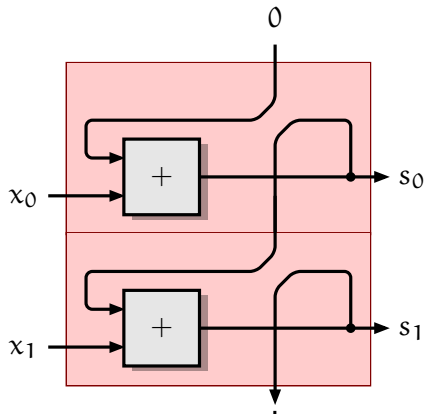
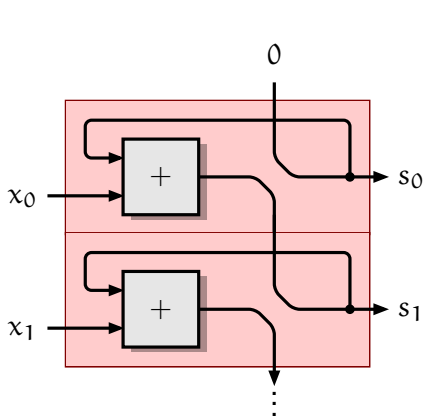
$$\left[\begin{array}{l} x \rightarrow s \\ s := (0 \ ; \ s) + x \end{array} \right]$$



Basic Example: Cumulative Sum

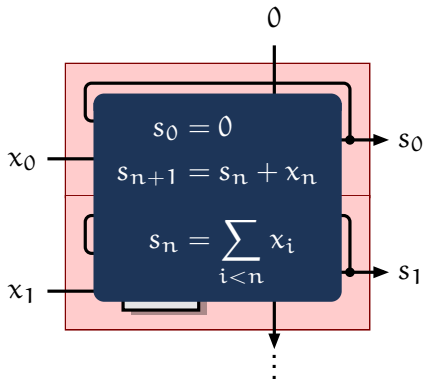
$$\left[\begin{array}{l} x \rightarrow s \\ s := 0 \ ; \ (s + x) \end{array} \right]$$

$$\left[\begin{array}{l} x \rightarrow s \\ s := (0 \ ; \ s) + x \end{array} \right]$$

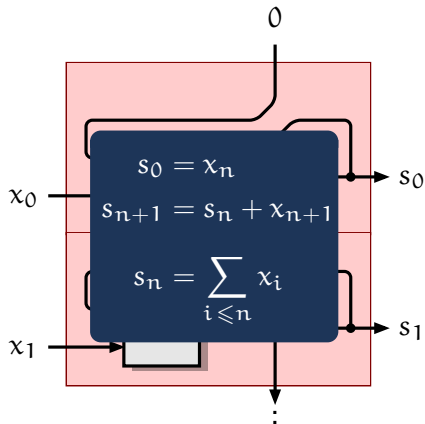


Basic Example: Cumulative Sum

$$\left[\begin{array}{l} x \rightarrow s \\ s := 0 \ ; \ (s + x) \end{array} \right]$$



$$\left[\begin{array}{l} x \rightarrow s \\ s := (0 \ ; \ s) + x \end{array} \right]$$



Agenda

1 Exposition

- Main Theme: SIG in a Nutshell
- Countertheme: The Shepard Tone

2 Development

- Higher-Order Stream Programming
- Multi-rate Stream Programming

3 Recapitulation

- Putting Things Together
- Conclusion

What is it?

- Psychoacoustic illusion (Shepard 1964; Risset 1986)
 - Contradictory short/long-term pitch perception
- Auditory strange loop, analog of this...
 - or that...
- Small but nontrivial sound synthesis problem
 - Full SIG code in the paper!

What is it?

- Psychoacoustic illusion (Shepard 1964; Risset 1986)
 - Contradictory short/long-term pitch perception
- Auditory strange loop, analog of this...
 - or that...
- Small but nontrivial sound synthesis problem
 - Full SIG code in the paper!



What is it?

- Psychoacoustic illusion (Shepard 1964; Risset 1986)
 - Contradictory short/long-term pitch perception
- Auditory strange loop, analog of this...
 - or that...
- Small but nontrivial sound synthesis problem
 - Full SIG code in the paper!

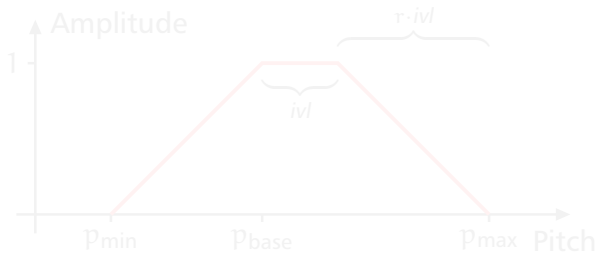
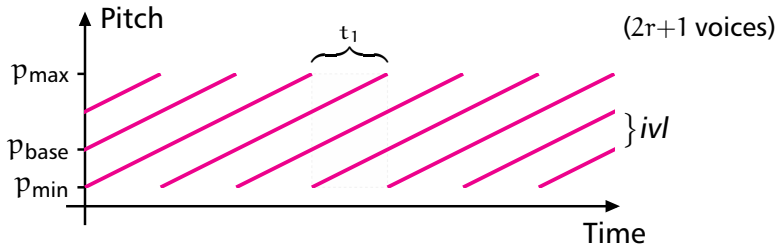


What is it?

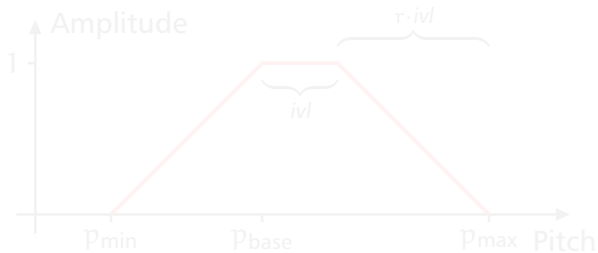
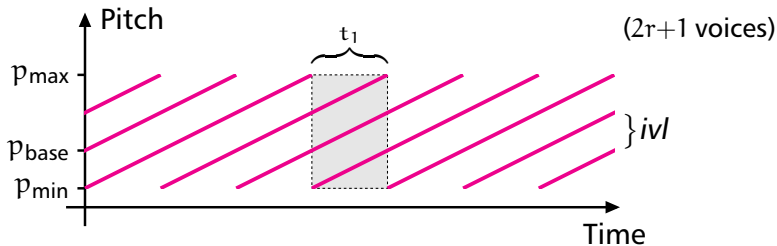
- Psychoacoustic illusion (Shepard 1964; Risset 1986)
 - Contradictory short/long-term pitch perception
- Auditory strange loop, analog of this. . .
 - or that. . .
- Small but nontrivial sound synthesis problem
 - Full SIG code in the paper!



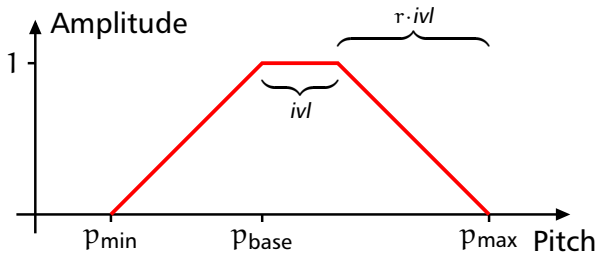
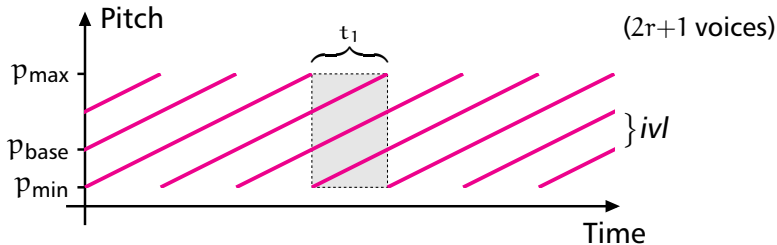
Specification



Specification



Specification



Agenda

- 1 Exposition**
 - Main Theme: SIG in a Nutshell
 - Countertheme: The Shepard Tone
- 2 Development**
 - Higher-Order Stream Programming
 - Multi-rate Stream Programming
- 3 Recapitulation**
 - Putting Things Together
 - Conclusion

Agenda

- 1 **Exposition**
 - Main Theme: SIG in a Nutshell
 - Countertheme: The Shepard Tone
- 2 **Development**
 - Higher-Order Stream Programming
 - Multi-rate Stream Programming
- 3 **Recapitulation**
 - Putting Things Together
 - Conclusion

Lambdas Got (Too Much) Rhythm

- Higher-order functions increase expressivity
- Combination with time-varying values nontrivial (Uustalu and Vene 2005)
- At odds with SIG paradigms
 - Every variable is a stream
 - Streams are synchronized

Counterexample: Currying

Lambdas Got (Too Much) Rhythm

- Higher-order functions increase expressivity
- Combination with time-varying values nontrivial (Uustalu and Vene 2005)
- At odds with SIG paradigms
 - Every variable is a stream
 - Streams are synchronized

Counterexample: Currying

Lambdas Got (Too Much) Rhythm

- Higher-order functions increase expressivity
- Combination with time-varying values nontrivial (Uustalu and Vene 2005)
- At odds with SIG paradigms
 - Every variable is a stream
 - Streams are synchronized

Counterexample: Currying

$$\left[\begin{array}{l} x, y \rightarrow z \\ z := x + y \end{array} \right]$$

$$z_i = x_i + y_i$$

$$\left[\begin{array}{l} x \rightarrow f \\ f := \left[\begin{array}{l} y \rightarrow z \\ z := x + y \end{array} \right] \end{array} \right]$$

$$f_i = \{y \mapsto z \mid z_j = x_i + y_j\}$$

Lambdas Got (Too Much) Rhythm

- Higher-order functions increase expressivity
- Combination with time-varying values nontrivial (Uustalu and Vene 2005)
- At odds with SIG paradigms
 - Every variable is a stream
 - Streams are synchronized

Counterexample: Currying

$$\left[\begin{array}{l} x, y \rightarrow z \\ z := x + y \end{array} \right]$$

$$z_i = x_i + y_i$$

$$\left[\begin{array}{l} x \rightarrow f \\ f := \left[\begin{array}{l} y \rightarrow z \\ z := x + y \end{array} \right] \end{array} \right]$$

$$f_i = \{y \mapsto z \mid z_j = x_i + y_j\}$$

Lambdas Got (Too Much) Rhythm

- Higher-order functions increase expressivity
- Combination with time-varying values nontrivial (Uustalu and Vene 2005)
- At odds with SIG paradigms
 - Every variable is a stream
 - Streams are synchronized

Counterexample: Currying

$$\left[\begin{array}{l} x, y \rightarrow z \\ z := x + y \end{array} \right]$$

$$z_i = x_i + y_i$$

$$\left[\begin{array}{l} x \rightarrow f \\ f := \left[\begin{array}{l} y \rightarrow z \\ z := x + y \end{array} \right] \end{array} \right]$$

$$f_i = \{y \mapsto z \mid z_j = x_i + y_j\}$$

Lambdas Got (Too Much) Rhythm

- Higher-order functions increase expressivity
- Combination with time-varying values nontrivial (Uustalu and Vene 2005)
- At odds with SIG paradigms
 - Every variable is a stream
 - Streams are synchronized

Counterexample: Currying

$$\left[\begin{array}{l} x, y \rightarrow z \\ z := x + y \end{array} \right]$$

$$z_i = x_i + y_i$$

$$\left[\begin{array}{l} x \rightarrow f \\ f := \left[\begin{array}{l} y \rightarrow z \\ z := x + y \end{array} \right] \end{array} \right]$$

$$f_i = \{y \mapsto z \mid z_j = x_i + y_j\}$$

To the Rescue: Staged Meta-Programming

- Operators for explicit control over evaluation order
 - Suspend** defers computation until next stage
 - Splice** escapes computation from next stage
 - Run** proceeds to next stage
- Similar to LISP quasiquotation
 - Confer `' / , / eval`
 - But strongly hygienic & typed
- Similar to off-line partial evaluation
 - Binding-time analysis creates two stages
 - Specialization runs first stage
 - But arbitrary number of stages
- Considered for adaptive high-performance computing (Kiselyov, Shan, and Kameyama 2012)

To the Rescue: Staged Meta-Programming

- Operators for explicit control over evaluation order
 - Suspend** defers computation until next stage
 - Splice** escapes computation from next stage
 - Run** proceeds to next stage
- Similar to LISP quasiquotation
 - Confer `' / , / eval`
 - But strongly hygienic & typed
- Similar to off-line partial evaluation
 - Binding-time analysis creates two stages
 - Specialization runs first stage
 - But arbitrary number of stages
- Considered for adaptive high-performance computing (Kiselyov, Shan, and Kameyama 2012)

To the Rescue: Staged Meta-Programming

- Operators for explicit control over evaluation order
 - Suspend** defers computation until next stage
 - Splice** escapes computation from next stage
 - Run** proceeds to next stage
- Similar to LISP quasiquotation
 - Confer `' / , / eval`
 - But strongly hygienic & typed
- Similar to off-line partial evaluation
 - Binding-time analysis creates two stages
 - Specialization runs first stage
 - But arbitrary number of stages
- Considered for adaptive high-performance computing (Kiselyov, Shan, and Kameyama 2012)

To the Rescue: Staged Meta-Programming

- Operators for explicit control over evaluation order
 - Suspend** defers computation until next stage
 - Splice** escapes computation from next stage
 - Run** proceeds to next stage
- Similar to LISP quasiquotation
 - Confer `' / , / eval`
 - But strongly hygienic & typed
- Similar to off-line partial evaluation
 - Binding-time analysis creates two stages
 - Specialization runs first stage
 - But arbitrary number of stages
- Considered for adaptive high-performance computing (Kiselyov, Shan, and Kameyama 2012)

No Stage Fright in SIG

- Operators # / ~ / %
- All anonymous functions must be suspended
- Suspension freezes time
 - Free variable refers to *element* at suspension time
 - No implicit cross-stage synchronization

Counterexample Defused

$$\left[\begin{array}{l} x, y \rightarrow z \\ z := x + y \end{array} \right]$$

$$z_i = x_i + y_i$$

$$\left[\begin{array}{l} x \rightarrow f \\ f := \# \left[\begin{array}{l} y \rightarrow z \\ z := x + y \end{array} \right] \end{array} \right]$$

$$f_i = \{y \mapsto z \mid z_j = x_i + y_j\}$$

No Stage Fright in SIG

- Operators # / ~ / %
- All anonymous functions must be suspended
- Suspension freezes time
 - Free variable refers to *element* at suspension time
 - No implicit cross-stage synchronization

Counterexample Defused

$$\left[\begin{array}{l} x, y \rightarrow z \\ z := x + y \end{array} \right]$$

$$z_i = x_i + y_i$$

$$\left[\begin{array}{l} x \rightarrow f \\ f := \# \left[\begin{array}{l} y \rightarrow z \\ z := x + y \end{array} \right] \end{array} \right]$$

$$f_i = \{y \mapsto z \mid z_j = x_i + y_j\}$$

No Stage Fright in SIG

- Operators # / ~ / %
- All anonymous functions must be suspended
- Suspension freezes time
 - Free variable refers to *element* at suspension time
 - No implicit cross-stage synchronization

Counterexample Defused

$$\left[\begin{array}{l} x, y \rightarrow z \\ z := x + y \end{array} \right]$$

$$z_i = x_i + y_i$$

$$\left[\begin{array}{l} x \rightarrow f \\ f := \# \left[\begin{array}{l} y \rightarrow z \\ z := x + y \end{array} \right] \end{array} \right]$$

$$f_i = \{y \mapsto z \mid z_j = x_i + y_j\}$$

Agenda

1 Exposition

- Main Theme: SIG in a Nutshell
- Countertheme: The Shepard Tone

2 Development

- Higher-Order Stream Programming
- Multi-rate Stream Programming

3 Recapitulation

- Putting Things Together
- Conclusion

Multi-rate Audio Signal Processing

- One system, many rates:
 - Audio** waveform @ 44 kHz (CD), 96 kHz (studio)
 - Control** parameters @ 1/64 audio, 1 kHz
 - Event** 24/quarter (MIDI), 120/minute (techno)
 - Zero** Initialization
- Asynchronous data flows both ways
 - Modulation** parameters from slow to fast
 - Aggregation** statistics from fast to slow
 - Configuration** components from slow to fast

Multi-rate Audio Signal Processing

- One system, many rates:
 - Audio** waveform @ 44 kHz (CD), 96 kHz (studio)
 - Control** parameters @ 1/64 audio, 1 kHz
 - Event** 24/quarter (MIDI), 120/minute (techno)
 - Zero** Initialization
- Asynchronous data flows both ways
 - Modulation** parameters from slow to fast
 - Aggregation** statistics from fast to slow
 - Configuration** components from slow to fast

Multi-rate Audio Signal Processing

- One system, many rates:
 - Audio** waveform @ 44 kHz (CD), 96 kHz (studio)
 - Control** parameters @ 1/64 audio, 1 kHz
 - Event** 24/quarter (MIDI), 120/minute (techno)
 - Zero** Initialization
- Asynchronous data flows both ways
 - Modulation** parameters from slow to fast
 - Aggregation** statistics from fast to slow
 - Configuration** components from slow to fast

Multi-rate SIG

$$\left[\begin{array}{l} x \rightarrow y \\ y := 0 \text{ ; } y + x * \mathbf{dt} \end{array} \right]$$

- SIG programs have implicit rate
 - Components reusable at different rates (*polydromic*)
 - But possibly reflected as local constant
 - Passive execution model, external driver
- Data flow is synchronous by default
 - Rate equations
 - Independent subsystems possibly at different rates
- Exceptions by explicit *resampling*
 - Rate inequations
 - Fixed conversion factors

Multi-rate SIG

$$\left[\begin{array}{l} x \rightarrow y \\ y := 0 \text{ ; } y + x * dt \end{array} \right]$$

- SIG programs have implicit rate
 - Components reusable at different rates (*polydromic*)
 - But possibly reflected as local constant
 - Passive execution model, external driver
- Data flow is synchronous by default
 - Rate equations
 - Independent subsystems possibly at different rates
- Exceptions by explicit *resampling*
 - Rate inequations
 - Fixed conversion factors

Multi-rate SIG

$$\left[\begin{array}{l} x \rightarrow y \\ y := 0 \text{ ; } y + x * \mathbf{dt} \end{array} \right]$$

- SIG programs have implicit rate
 - Components reusable at different rates (*polydromic*)
 - But possibly reflected as local constant
 - Passive execution model, external driver
- Data flow is synchronous by default
 - Rate equations
 - Independent subsystems possibly at different rates
- Exceptions by explicit *resampling*
 - Rate inequations
 - Fixed conversion factors

Implementation Strategy

```
# [ → out
  out := upsample amp * sin phase
  phase := 0 ; phase + freq * dt ]
```

- Amplitude modulated, frequency fixed
- Static rate analysis

$$\underbrace{\mathcal{R}(amp)}_{R_1} \leq \underbrace{\mathcal{R}(phase) = \mathcal{R}(out)}_{R_2}$$

- Slicing into synchronous subcomponents
- Buffered asynchronous data flow behind the scenes
 - Well-defined scheduling rules

Implementation Strategy

```
# [ amp → out
  out := upsample amp * sin phase
  phase := 0 ; phase + freq * dt ]
```

- Amplitude modulated, frequency fixed
- Static rate analysis

$$\underbrace{\mathcal{R}(amp)}_{R_1} \leq \underbrace{\mathcal{R}(phase) = \mathcal{R}(out)}_{R_2}$$

- Slicing into synchronous subcomponents
- Buffered asynchronous data flow behind the scenes
 - Well-defined scheduling rules

Implementation Strategy

```
# [ amp → out
  out := upsample amp * sin phase
  phase := 0 ; phase + freq * dt ]
```

- Amplitude modulated, frequency fixed
- Static rate analysis

$$\underbrace{\mathcal{R}(amp)}_{R_1} \leq \underbrace{\mathcal{R}(phase) = \mathcal{R}(out)}_{R_2}$$

- Slicing into synchronous subcomponents
- Buffered asynchronous data flow behind the scenes
 - Well-defined scheduling rules

Implementation Strategy

$$\# \left[\begin{array}{l} \text{amp} \rightarrow \\ \text{amp}^- := \text{amp} \end{array} \right] \# \left[\begin{array}{l} \rightarrow \text{out} \\ \text{out} := \text{amp}^+ * \sin \text{phase} \\ \text{phase} := 0 \ ; \ \text{phase} + \text{freq} * \text{dt} \end{array} \right]$$

- Amplitude modulated, frequency fixed
- Static rate analysis

$$\underbrace{\mathcal{R}(\text{amp})}_{R_1} \leq \underbrace{\mathcal{R}(\text{phase}) = \mathcal{R}(\text{out})}_{R_2}$$

- Slicing into synchronous subcomponents
 - Buffered asynchronous data flow behind the scenes
 - Well-defined scheduling rules

Implementation Strategy

$$\# \left[\begin{array}{l} amp \rightarrow amp^- \\ amp^- := amp \end{array} \right] \quad \# \left[\begin{array}{l} amp^+ \rightarrow out \\ out := amp^+ * \sin phase \\ phase := 0 \ ; \ phase + freq * dt \end{array} \right]$$

- Amplitude modulated, frequency fixed
- Static rate analysis

$$\underbrace{\mathcal{R}(amp)}_{R_1} \leq \underbrace{\mathcal{R}(phase) = \mathcal{R}(out)}_{R_2}$$

- Slicing into synchronous subcomponents
- Buffered asynchronous data flow behind the scenes
 - Well-defined scheduling rules

Agenda

- 1 Exposition**
 - Main Theme: SIG in a Nutshell
 - Countertheme: The Shepard Tone
- 2 Development**
 - Higher-Order Stream Programming
 - Multi-rate Stream Programming
- 3 Recapitulation**
 - Putting Things Together
 - Conclusion

Agenda

1 Exposition

- Main Theme: SIG in a Nutshell
- Countertheme: The Shepard Tone

2 Development

- Higher-Order Stream Programming
- Multi-rate Stream Programming

3 Recapitulation

- Putting Things Together
- Conclusion

The Shepard Tone in SIG

- Straightforward architecture
 - Three tiers, loosely corresponding to three rates
 - Two-stage (curried) config/runtime separation each
- Not quite trivial code
 - 27 LoC, as detailed in the paper
 - References to primitives not discussed here

Top-Down Walkthrough

- 1 Ensemble maintains an array of live voices (functions)
 - shift at rate R_1 driven by external clock
 - outputs mixed together
- 2 Voice modulates oscillator (amp+freq)
 - linear pitch increase
 - quantized at rate R_2 driven by subdivided clock
- 3 Oscillator maintains phase continuity
 - *envelope*, waveform global function-valued parameters
 - sample at audio rate R_3 driven by RT audio system

```
shepard_2 := #[ clk_1 : void -> s_out : real
  where
    make                := #[ k : int -> m_out := $voice(k * ivl, ivl / dt(clk_1)) ]
    ensemble @ clk_1 := %make(seq(-r, +r)) ; shiftr(%make(-r), ensemble)
    s_out              := sum(ensemble(upsample(clk_1, res)))
]
```

Top-Down Walkthrough

- 1 Ensemble maintains an array of live voices (functions)
 - shift at rate R_1 driven by external clock
 - outputs mixed together
- 2 Voice modulates oscillator (amp+freq)
 - linear pitch increase
 - quantized at rate R_2 driven by subdivided clock
- 3 Oscillator maintains phase continuity
 - *envelope*, waveform global function-valued parameters
 - sample at audio rate R_3 driven by RT audio system

```

voice := #[ init_pitch, ascent : real ->
  voice_2 := #[ clk_2 : void -> v_out : real
    where
      pitch @ clk_2 := init_pitch ; pitch + ascent * dt
      v_out          := $($osci(0))($envelope(pitch), base_freq * exp(pitch))
    ]
  }

```

Top-Down Walkthrough

- 1 Ensemble maintains an array of live voices (functions)
 - shift at rate R_1 driven by external clock
 - outputs mixed together
- 2 Voice modulates oscillator (amp+freq)
 - linear pitch increase
 - quantized at rate R_2 driven by subdivided clock
- 3 Oscillator maintains phase continuity
 - *envelope*, waveform global function-valued parameters
 - sample at audio rate R_3 driven by RT audio system

```
osci := #[ init_phase : real ->
  osci_2 := #[ amp, freq : real -> o_out : real
    where
      phase := init_phase ; phase + upsample(freq) * dt
      o_out := upsample(amp) * $wave(phase)
    ]
  ]
```

Agenda

- 1 Exposition**
 - Main Theme: SIG in a Nutshell
 - Countertheme: The Shepard Tone
- 2 Development**
 - Higher-Order Stream Programming
 - Multi-rate Stream Programming
- 3 Recapitulation**
 - Putting Things Together
 - Conclusion

Implementation Status

- Textual frontend language compiles to Java VM
- Various advanced features **implemented**
 - ADTs & pattern matching
 - Staged higher-order components
- Runtime environment **complete**
 - Including multi-rate components
- Rate analysis support **incomplete**
 - Multi-rate code generator out of order
- Demo with **manual** Java coding
 - Simulated code generation
 - Against actual API; binary compatible

Implementation Status

- Textual frontend language compiles to Java VM
- Various advanced features **implemented**
 - ADTs & pattern matching
 - Staged higher-order components
- Runtime environment **complete**
 - Including multi-rate components
- Rate analysis support **incomplete**
 - Multi-rate code generator out of order
- Demo with **manual** Java coding
 - Simulated code generation
 - Against actual API; binary compatible

Implementation Status

- Textual frontend language compiles to Java VM
- Various advanced features **implemented**
 - ADTs & pattern matching
 - Staged higher-order components
- Runtime environment **complete**
 - Including multi-rate components
- Rate analysis support **incomplete**
 - Multi-rate code generator out of order
- Demo with **manual** Java coding
 - Simulated code generation
 - Against actual API; binary compatible

Implementation Status

- Textual frontend language compiles to Java VM
- Various advanced features **implemented**
 - ADTs & pattern matching
 - Staged higher-order components
- Runtime environment **complete**
 - Including multi-rate components
- Rate analysis support **incomplete**
 - Multi-rate code generator out of order
- Demo with **manual** Java coding
 - Simulated code generation
 - Against actual API; binary compatible

Implementation Status

- Textual frontend language compiles to Java VM
- Various advanced features **implemented**
 - ADTs & pattern matching
 - Staged higher-order components
- Runtime environment **complete**
 - Including multi-rate components
- Rate analysis support **incomplete**
 - Multi-rate code generator out of order
- Demo with **manual** Java coding
 - Simulated code generation
 - Against actual API; binary compatible

Concluding Stuff

- Many things to do
 - Make rates work
 - Whole-program analysis & optimization
 - Hard real-time runtime environments
- Take-home message:

Multi-rate higher-order programming rocks

- Dynamic reconfiguration of signal processing networks
- Applications ranging from trivial to hugely complex
- Clean compositional semantics & reliability essential

Concluding Stuff

- Many things to do
 - Make rates work
 - Whole-program analysis & optimization
 - Hard real-time runtime environments
- Take-home message:

Multi-rate higher-order programming rocks

- Dynamic reconfiguration of signal processing networks
- Applications ranging from trivial to hugely complex
- Clean compositional semantics & reliability essential

SIG Bibliography I



Trancón y Widemann, B. and M. Lepper (2011). “tSig — Towards Semantics for a Functional Synchronous Signal Language”. In: *16. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2011)*. Ed. by H. Kuchen and M. Müller-Olm. Arbeitsberichte des Instituts für Wirtschaftsinformatik 132. Universität Münster, pp. 163–168.






— (2014a). “Foundations of Total Functional Data-Flow Programming”. In: *Mathematically Structured Functional Programming (MSFP 2014)*. Vol. 153. Electronic Proceedings in Theoretical Computer Science, pp. 143–167. DOI: 10.4204/EPTCS.153.10.



— (2014b). “Sound and Soundness – Practical Total Functional Data-Flow Programming”. In: *Functional Art, Music, Modeling and Design (FARM 2014)*. Demo abstract. ACM Digital Library, pp. 35–36. DOI: 10.1145/2633638.2633644.

SIG Bibliography II

-  Trancón y Widemann, B. and M. Lepper (2014c). “Towards Execution of the Synchronous Functional Data-Flow Language Sig”. In: *Implementation and Application of Functional Languages (IFI 2014)*. Draft proceedings.
-  — (2015a). “Laminar Data Flow: On the Role of Slicing in Functional Data-Flow Programming”. In: *Trends in Functional Programming (TFP 2015)*. Draft proceedings.
-  — (2015b). “On-Line Synchronous Total Purely Functional Data-Flow Programming on the Java Virtual Machine with Sig”. In: *Proc. International Conference Principles and Practice of Programming in Java (PPPJ 2015)*. To appear. ACM.

References I



Kiselyov, O., C.-C. Shan, and Y. Kameyama (2012). *Bridging the theory of staged programming languages and the practice of high-performance computing*. Tech. rep. 2012–4. National Institute of Informatics, Japan. URL: <http://www.nii.ac.jp/shonan/wp-content/uploads/2011/09/No.2012--4.pdf>.



Risset, J.-C. (1986). “Pitch and rhythm paradoxes: Comments on “Auditory paradox based on fractal waveform” [J. Acoust. Soc. Am. 79, 186–189 (1986)]”. In: *The Journal of the Acoustical Society of America* 80.3, pp. 961–962. DOI: 10.1121/1.393919.



Shepard, R. N. (1964). “Circularity in Judgements of Relative Pitch”. In: *Journal of the Acoustical Society of America* 36.12, pp. 2346–2353. DOI: 10.1121/1.1919362.



Uustalu, T. and V. Vene (2005). “The Essence of Dataflow Programming”. In: *Programming Languages and Systems*. Ed. by K. Yi. Vol. 3780. Lecture Notes in Computer Science. Springer, pp. 2–18. DOI: 10.1007/11575467_2.