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
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
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
• **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
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 $\Sigma$G 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 S\textsuperscript{IG} 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 \textit{simple} compositional semantics
  - Functional frontend: ADTs, pattern matching, \textbf{higher order}
  - Advanced features: physical units, \textbf{multi-rate}

- Semantics of higher layers by transformation
Basic Example: Cumulative Sum

$$\chi \rightarrow s$$
$$s := 0 \hat{\chi} (s + \chi)$$

$$\chi \rightarrow s$$
$$s := (0 \hat{\chi} s) + \chi$$
Basic Example: Cumulative Sum

\[ x \rightarrow s \]
\[ s := 0 \cdot (s + x) \]

\[ x \rightarrow s \]
\[ s := (0 \cdot s) + x \]
Basic Example: Cumulative Sum

\[
\begin{align*}
\chi &\rightarrow s \\
\text{s} &:= 0 \odot (s + \chi)
\end{align*}
\]

\[
\begin{align*}
\chi &\rightarrow s \\
\text{s} &:= (0 \odot s) + \chi
\end{align*}
\]
Basic Example: Cumulative Sum

\[ x \to s \]
\[ s := 0 \circ (s + x) \]

\[ x \to s \]
\[ s := (0 \circ s) + x \]
Basic Example: Cumulative Sum

\[
\begin{align*}
\mathbf{x} & \rightarrow \mathbf{s} \\
\mathbf{s} & := 0 \oplus (\mathbf{s} + \mathbf{x})
\end{align*}
\]

\[
\begin{align*}
\mathbf{x} & \rightarrow \mathbf{s} \\
\mathbf{s} & := (0 \oplus \mathbf{s}) + \mathbf{x}
\end{align*}
\]
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

Pitch

$\begin{align*}
\text{Time} & \\
\text{Pitch} & \\
\min & \\
\text{base} & \\
\max & \\
(2r+1 \text{ voices})
\end{align*}$

$\begin{align*}
\text{Amplitude} & \\
\min & \\
\text{base} & \\
\max & \\
\end{align*}$

$\begin{align*}
\text{t}_1 & \\
\text{r-ivl} & \\
\text{ivl} & \\
\end{align*}$

Trancón y Widemann, Lepper

Shepard @ SIG
Specification

Pitch

\[ \begin{align*}
\p_{\text{max}} & \quad \p_{\text{base}} & \quad \p_{\text{min}} \\
\end{align*} \]

Time

(2r+1 voices)

\[ \{ \text{ivl} \} \]

Amplitude

\[ \begin{align*}
1 & \quad \text{r} \cdot \text{ivl} & \quad \text{ivl} \\
\p_{\text{min}} & \quad \p_{\text{base}} & \quad \p_{\text{max}} \\
\end{align*} \]
**Specification**

- **Pitch**
  - $p_{\text{max}}$
  - $p_{\text{base}}$
  - $p_{\text{min}}$

- **Time**
  - $t_1$
  - $(2r+1 \text{ voices})$

- **Amplitude**
  - $r \cdot ivl$
  - $ivl$

---

Trancón y Widemann, Lepper

Shepard @ SIG

5/15
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
Exposition
- Main Theme: SIG in a Nutshell
- Countertheme: The Shepard Tone

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

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**

\[
\begin{align*}
(x, y \rightarrow z) \\
&\quad \text{where } z := x + y
\end{align*}
\]

\[
\begin{align*}
x \rightarrow f \\
&\quad \text{where } f := (y \rightarrow z) \\
&\quad \text{with } z := x + y
\end{align*}
\]
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**

\[
\begin{align*}
x, y \rightarrow z \\
z &:= x + y
\end{align*}
\]

\[
f_i = \{ y \mapsto z \mid z_j = x_i + y_j \}
\]
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**

\[
\begin{align*}
  &\begin{align*}
    & x, y \rightarrow z \\
    & z := x + y \\
    & x \rightarrow f \\
    & f := \begin{align*}
      & y \rightarrow z \\
      & z := x + y
    \end{align*}
  \end{align*} \\
  \end{align*}
\]

\[
  z_i = x_i + y_i
\]

\[
  f_i = \{ y \mapsto z \mid z_j = x_i + y_j \}
\]
Lambdas Got (Too Much) Rhythm

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

**Counterexample: Currying**

\[
\begin{align*}
\chi, \gamma &\rightarrow \zeta \\
\zeta &:= \chi + \gamma
\end{align*}
\]

\[
\begin{align*}
\chi &\rightarrow f \\
f &:= \begin{cases}
\gamma &\rightarrow \zeta \\
\zeta &:= \chi + \gamma
\end{cases}
\end{align*}
\]

\[
\begin{align*}
z_i &= \chi_i + \gamma_i \\
f_i &= \{ \gamma \mapsto \zeta \mid \zeta_j = \chi_i + \gamma_j \}
\end{align*}
\]
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)
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 S†G

- 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

\[
\begin{align*}
  & x, y \rightarrow z \\
  & z := x + y \\
  & x \rightarrow f \\
  & f := \# \quad y \rightarrow z \\
  & z := x + y \\
  & z_i = x_i + y_i \\
  & f_i = \{ y \mapsto z \mid z_j = x_i + y_j \}
\end{align*}
\]
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

\[
\begin{align*}
  &\begin{cases}
    x, y \rightarrow z \\
    z := x + y
  \end{cases} \\
  &z_i = x_i + y_i \\
  &\begin{cases}
    x \rightarrow f \\
    f := # \\
    y \rightarrow z \\
    z := x + y
  \end{cases} \\
  &f_i = \{y \rightarrow z \mid z_j = x_i + y_j\}
\end{align*}
\]
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

\[
\begin{align*}
\begin{bmatrix}
\chi, \gamma \rightarrow z \\
\chi := \chi + \gamma \\
\chi \rightarrow f \\
f := \# [y \rightarrow z] \\
z := \chi + \gamma
\end{bmatrix}
\end{align*}
\]

\[z_i = \chi_i + \gamma_i\]

\[
\begin{align*}
\begin{bmatrix}
y \rightarrow z \\
z := \chi + \gamma
\end{bmatrix}
\end{align*}
\]

\[f_i = \{y \mapsto z \mid z_j = \chi_i + \gamma_j\}\]
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
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 $\text{SIG}$

\[ x \rightarrow y \]
\[ y := 0 \odot y + x \ast dt \]

- $\text{SIG}$ 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

Trancón y Widemann, Lepper

Shepard @ SIG
### Multi-rate S\(\text{IG}\)

\[
\begin{bmatrix}
x \rightarrow y \\
y := 0 \cdot y + x \cdot dt
\end{bmatrix}
\]

- **S\(\text{IG}\) 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
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

\[
\begin{align*}
\text{→} & \quad \text{out} \\
\text{out} & := \text{upsample} \quad amp \ast \sin \text{phase} \\
\text{phase} & := 0 \ast \text{phase} + freq \ast dt
\end{align*}
\]

- Amplitude modulated, frequency fixed
- Static rate analysis
  \[ R(amp) \leq R(phase) = R(out) \]
  \( R_1 \leq R_2 \)
- Slicing into synchronous subcomponents
- Buffered asynchronous data flow behind the scenes
  - Well-defined scheduling rules
Implementation Strategy

\[
\begin{bmatrix}
\text{amp} \rightarrow \text{out} \\
\text{out} := \text{upsample amp} \times \sin \text{phase} \\
\text{phase} := 0 \circ \text{phase} + \text{freq} \times \text{dt}
\end{bmatrix}
\]

- Amplitude modulated, frequency fixed
- Static rate analysis
  \[
  \mathcal{R}(\text{amp}) \leq \mathcal{R}(\text{phase}) = \mathcal{R}(\text{out})
  \]
- Slicing into synchronous subcomponents
- Buffered asynchronous data flow behind the scenes
  - Well-defined scheduling rules
Implementation Strategy

\[ \text{amp} \rightarrow \text{out} \]

\[
\begin{align*}
\# & \quad \text{out} := \text{upsample amp} \times \sin \text{phase} \\
& \quad \text{phase} := 0 \circ \text{phase} + \text{freq} \times dt
\end{align*}
\]

- Amplitude modulated, frequency fixed
- Static rate analysis

\[
\begin{align*}
\mathcal{R}(\text{amp}) & \leq \mathcal{R}(\text{phase}) = \mathcal{R}(\text{out}) \\
\{ R_1 \} & \quad \{ R_2 \}
\end{align*}
\]

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

- Amplitude modulated, frequency fixed
- Static rate analysis

\[
\begin{align*}
amp & \rightarrow \\
amp^- & := amp \\
\text{out} & := amp^+ \cdot \sin \text{phase} \\
\text{phase} & := 0 \; ; \; \text{phase} + \text{freq} \cdot dt
\end{align*}
\]

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

- Amplitude modulated, frequency fixed
- Static rate analysis

\[ R(amp) \leq R(phase) = R(out) \]

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

\[ \begin{align*}
amp & \rightarrow amp^- \\
amp^- & := amp \\
amp^+ & \rightarrow out \\
out & := amp^+ \ast \sin \text{phase} \\
\text{phase} & := 0 \ast \text{phase} + \text{freq} \ast dt
\end{align*} \]
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
Exposition
- Main Theme: SIG in a Nutshell
- Countertheme: The Shepard Tone

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

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
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 =\rightarrow s_out : real
    where
      make := # [ k : int =\rightarrow 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)))
    ]
```
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
   - \textit{envelope}, waveform global function-valued parameters
   - sample at audio rate $R_3$ driven by RT audio system

```plaintext

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))
   ]
}
```

Trancón y Widemann, Lepper

Shepard @ Sig
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

```latex
osci := #[ init_phase : real ] ->
osci_2 := #[ amp, freq : real ] -> o_out : real
  where
  phase := init_phase ; phase + upsample(freq) * dt
  o_out := upsample(amp) * $\text{wave}(phase)$
```
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 advances 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
Textual frontend language compiles to Java VM

Various advances 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 advances 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 advances 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 advances 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


