The WaveScope Project

Functional Programming in the Wild

Ryan Newton
Greg Morrisett
Sam Madden
MIT/CSAIL

Also with:
Lewis Girod, Mike Allen, Kyle Jamieson, Yuan Mei,
Stanislav Rost, Arvind Thiagarajan,
Hari Balakrishnan

http://wavescope.csail.mit.edu/
Applications:
Stream + Signal Processing

http://wavescope.csail.mit.edu/
Applications:
Stream + Signal Processing

- Pipeline leak detection and localization

Are there anomalies in the frequency response to an introduced pulse?

http://wavescope.csail.mit.edu/
Applications: Stream + Signal Processing

- Pipeline leak detection and localization
- Seizure onset detection using EEG

Are there anomalies in the frequency response to an introduced pulse?

Is a seizure imminent given signals from various brain regions?

http://wavescope.csail.mit.edu/
Applications: Stream + Signal Processing

- Pipeline leak detection and localization
- Seizure onset detection using EEG
- \textit{In situ} animal behavior studies

Are there anomalies in the frequency response to an introduced pulse?
Is a seizure imminent given signals from various brain regions?
What time ranges contained marmot calls?

http://wavescope.csail.mit.edu/
Applications: Stream + Signal Processing

- Pipeline leak detection and localization
- Seizure onset detection using EEG
- *In situ* animal behavior studies

Are there anomalies in the frequency response to an introduced pulse?

Is a seizure imminent given signals from various brain regions?

What time ranges contained marmot calls?

http://wavescope.csail.mit.edu/
Application Features
Application Features

High data-rate

4 channels x 48 khz
= 400,000 bytes/sec (per node)

x 10-20 nodes
Application Features

High data-rate

Embedded, low-power devices

4 channels x 48 khz = 400,000 bytes/sec (per node)

x 10-20 nodes
Application Features

- Embedded, low-power devices
- Regularly sampled signals

High data-rate

4 channels x 48 khz = 400,000 bytes/sec (per node)

x 10-20 nodes
Application Features

- **Embedded, low-power devices**

- **High data-rate**
  - 4 channels x 48 khz = 400,000 bytes/sec (per node)
  - x 10-20 nodes

- **Regularly sampled signals**
  - Consistent data rates
  - Efficient time-stamping
Application Features

High data-rate

- 4 channels x 48 khz = 400,000 bytes/sec (per node)
- x 10-20 nodes

Embedded, low-power devices

- Regularly sampled signals
  - Consistent data rates
  - Efficient time-stamping

Discrete Events

```
animalcalls = detector(audio)
s2 = map(classify, marmotcalls)
```
Application Features

Embedded, low-power devices

High data-rate

• 4 channels x 48 khz = 400,000 bytes/sec (per node)
• 10-20 nodes

Regularly sampled signals

• Consistent data rates
• Efficient time-stamping

Discrete Events

• First class streams
• Higher order stream combinators (map, etc)

animalcalls = detector(audio)
s2 = map( classify, marmotcalls )
Application Features

- High data-rate
  - 4 channels \times 48 \text{ khz} = 400,000 \text{ bytes/sec (per node)}

- Regularly sampled signals
  - Consistent data rates
  - Efficient time-stamping

- Discrete Events
  - `animalcalls = detector(audio)`
  - `s2 = map( classify, marmotcalls)`

- Embedded, low-power devices
- First class streams
- Higher order stream combinators (map, etc)

- Parallelism at multiple granularities
“WaveScope” - what is it?

- WaveScript Language, Compiler
  - ML (with a few extras - generic printing, Num subkind)
  - Two-stage evaluation (metaprogramming)
  - Run-time model is a graph of stream operators
- XStream engine
  - Links to C backend, other WS backends as well
- Stream/Signal processing libraries
  - Higher order language, polymorphism, metaprogramming enables reuse, new abstractions
Execution Model

- Graph of operators (aka kernels, actors, agents)

- Many possible semantics: synchronous or asynchronous, (non)deterministic operators, shared state, atomicity/execution order, etc

- WaveScript: asynchronous streams of discrete events. Operators may fire whenever an input token is available.
  - Fairness is scheduler’s only requirement.
  - NO shared state
A bit of WaveScript code

• One primary stream operator (also merge)

```wave
iterate x in strm {
  state { cnt = 0 }
  cnt += 1;
  emit g(x,cnt);
  emit f(x,cnt);
}
```

• `iterate = Syntactic sugar`
  • second class references

• `iterate could be a pure combinator`,
  \[
  \text{iterate} :: ((a,s) \rightarrow ([b],s)) \rightarrow \text{Stream a} \rightarrow \text{Stream b}
  \]
• One primary stream operator (also merge)

iterate x in strm {
    state { cnt = 0 }
    cnt += 1;
    emit g(x,cnt);
    emit f(x,cnt);
}

• iterate = Syntactic sugar

• second class references

• iterate could be a pure combinator,

iterate :: ((a,s) -> ([b],s))
        -> Stream a -> Stream b
Compilation Workflow

WS Compiler (Scheme)
Compilation Workflow

WS Compiler

(Scheme)
Compilation Workflow

WS Compiler

(Scheme)

Scheme Backend

SML Backend

C++ Backend

XStream Runtime
Compilation Workflow

WS Compiler

- Scheme Backend
- SML Backend
- C++ Backend

XStream Runtime

quick compile, interactive queries
Compilation Workflow

- Scheme Backend
- SML Backend
- C++ Backend

WS Compiler (Scheme)

XStream Runtime

- quick compile, interactive queries
- MLton - single thread copying collector
Compilation Workflow

WS Compiler

- Scheme Backend
  - quick compile, interactive queries
- SML Backend
  - MLton - single thread copying collector
- C++ Backend
  - concurrent, reference counting collector

XStream Runtime
Compilation Workflow

- **WS Compiler**
  - **Scheme Backend**
    - Quick compile, interactive queries
  - **SML Backend**
    - MLton - single thread copying collector
  - **C++ Backend**
    - Concurrent, reference counting collector
- **XStream Runtime**

- Skip most of the compiler
- EDSL, macros
Why not a general-purpose, higher order language?
Why not a general-purpose, higher order language?

\[
\text{type } \text{Stream } t = () \rightarrow (t, \text{Stream } t)
\]
Why not a general-purpose, higher order language?

define Stream t = (t → (t, Stream t))
define Sink t = t → ()
define Stream t = Sink t → ()
Why not a general-purpose, higher order language?

type Stream t = () → (t, Stream t)

type Sink t = t → ()
type Stream t = Sink t → ()
Why not a general-purpose, higher order language?

- Probably not the most efficient code

```haskell
type Stream t = () → (t, Stream t)

type Sink t = t → ()
type Stream t = Sink t → ()
```
Why not a general-purpose, higher order language?

- Probably not the most efficient code
- You’re stuck with the execution environment

```ocaml
type Stream t = () -> (t, Stream t)
type Sink t = t -> ()
type Stream t = Sink t -> ()
```
Why not a general-purpose, higher order language?

- Probably not the most efficient code
- You’re stuck with the execution environment
  - Want to develop runtimes for tiny platforms

```plaintext
type Stream t = () → (t, Stream t)

type Sink t = t → ()

type Stream t = Sink t → ()
```
Why not a general-purpose, higher order language?

- Probably not the most efficient code
- You’re stuck with the execution environment
  - Want to develop runtimes for tiny platforms
  - Low-overhead memory management

```ocaml
type Stream t = () → (t, Stream t)

type Sink t = t → ()

type Stream t = Sink t → ()
```
Why not a general-purpose, higher order language?

- Probably not the most efficient code
- You’re stuck with the execution environment
  - Want to develop runtimes for tiny platforms
  - Low-overhead memory management
- Parallel runtime and garbage collector

```haskell
type Stream t = () -> (t, Stream t)

type Sink t = t -> ()

type Stream t = Sink t t -> ()
```
Two-Stage Evaluation

- Leverage asymmetric metaprogramming
  - Quotation / Anti-quotation free
  - Anything inside an iterate is “object code”
Two-Stage Evaluation

- Leverage asymmetric metaprogramming
  - Quotation / Anti-quotation free
  - Anything inside an iterate is “object code”
- Two meta-program evaluators:
  - Term rewriting (beta, delta reduction)
  - Interpret, marshal, inline until monomorphic
Two-Stage Evaluation

• Leverage asymmetric metaprogramming
  • Quotation / Anti-quotation free
  • Anything inside an iterate is “object code”

• Two meta-program evaluators:
  • Term rewriting (beta, delta reduction)
  • Interpret, marshal, inline until monomorphic

• Both equivalent to single-stage evaluation
  • With Scheme backend we allow single-stage
**Two-Stage Evaluation**

- Leverage asymmetric metaprogramming
  - Quotation / Anti-quotation free
  - Anything inside an iterate is “object code”
- Two meta-program evaluators:
  - Term rewriting (beta, delta reduction)
  - Interpret, marshal, inline until monomorphic
- Both equivalent to single-stage evaluation
  - With Scheme backend we *allow* single-stage
- Downside: potential code bloat
Marmot application

http://wavescope.csail.mit.edu/
Marmot application

- Goal: study calling behavior.

http://wavescope.csail.mit.edu/
Marmot application

- Goal: study calling behavior.

http://wavescope.csail.mit.edu/
Marmot application

- Goal: study calling behavior.

http://wavescope.csail.mit.edu/
Marmot application

- Goal: study calling behavior.
- Detect, record, localize, classify.

http://wavescope.csail.mit.edu/
Three phases to marmot localization

1. Detect
2. DOA
3. Fuse

http://wavescope.csail.mit.edu/
Three phases to marmot localization

1. Detect
2. DOA
3. Fuse

Snippet of 4-channel Audio

http://wavescope.csail.mit.edu/
Three phases to marmot localization

1. Detect
2. DOA
3. Fuse

Snippet of 4-channel Audio
Direction of arrival

http://wavescope.csail.mit.edu/
Three phases to marmot localization

1. Detect
2. DOA
3. Fuse

Snippet of 4-channel Audio
Direction of arrival
Probability Map
Three phases to marmot localization

1. Detect
2. DOA
3. Fuse

On Node

Snippet of 4-channel Audio
Direction of arrival
Probability Map

http://wavescope.csail.mit.edu/
Three phases to marmot localization

1. Detect
2. DOA
3. Fuse

On Node

On Server

Snippet of 4-channel Audio

Direction of arrival

Probability Map

http://wavescope.csail.mit.edu/
Three phases to marmot localization

1. Detect
2. DOA
3. Fuse

On Node → Node or Server → On Server

Snippet of 4-channel Audio
Direction of arrival
Probability Map
Schematic of Marmot-detector
(Phase 1)

http://wavescope.csail.mit.edu/
Schematic of Marmot-detector
(Phase 1)

Fast-path DSP to determine temporal ranges for marmot calls

ProfileDetector

Audio0

Audio1

Audio2

Audio3

sync

<w1,w2,w3,w4>

http://wavescope.csail.mit.edu/
Schematic of Marmot-detector

(Phase 1)

Stream’s Tuple Schema

ProfileDetector

Audio0
Audio1
Audio2
Audio3

sync

http://wavescope.csail.mit.edu/
Schematic of Marmot-detector

(Phase 1)

Stream’s Tuple Schema

Data Model:

- **Streams** are first-class values.
- **Streams** contain any type but Stream
  - Algebraic data-types (but not recursive)
  - Size of every type is statically known
  - Dynamic allocation allowed
- **SigSegs**: efficiently managed windows of samples
  - cheap to append, copy, forward, rewindow
  - fewer timestamps

http://wavescope.csail.mit.edu/
Schematic of Marmot-detector (Phase1)

Stream's Tuple Schema

Audio0
Audio1
Audio2
Audio3

ProfileDetector

<sync>

<w1,w2,w3,w4>

http://wavescope.csail.mit.edu/
WaveScript detector code

http://wavescope.csail.mit.edu/
WaveScript detector code

// Main query, phase 1

Ch0 = AudioSource(0, 48000, 1024);
Ch1 = AudioSource(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore, (64,192));
datawindows = sync4(control, Ch0, Ch1, Ch2, Ch4);
WaveScript detector code

// Main query, phase 1
Ch0 = AudioSource(0, 48000, 1024);
Ch1 = AudioSource(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);
control = profileDetect(Ch0, marmotScore, (64,192));
datawindows = sync4(control, Ch0, Ch1, Ch2, Ch4);

http://wavescope.csail.mit.edu/
fun profileDetect(S : Stream (SigSeg Int16),
    scorefun,
    (winsize, step))
{
    wins = rewindow(S, winsize, step);
    scores : Stream Float
    scores = map(scorefun o FFT, wins);
    withscores : Stream (Float, SigSeg Int16)
    withscores = zip2(scores, wins);
    return threshFilter(withscores);
}
WaveScript Code for Detector

fun profileDetect(S : Stream (SigSeg Int16), scorefun, (winsize, step))
{
    wins = rewindow(S, winsize, step);
    scores : Stream Float
    scores = map(scorefun o FFT, wins);
    withscores : Stream (Float, SigSeg Int16)
    withscores = zip2(scores, wins);
    return threshFilter(withscores);
}

Ch0 = AudioSource(0, 48000, 1024);
Ch1 = AudioSource(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);

control = profileDetect(Ch0, marmotScore, (64, 192));
datawindows = sync4(control, Ch0, Ch1, Ch2, Ch4);
fun profileDetect(S : Stream (SigSeg Int16),
    scorefun,
    (winsize,step))
{
    wins = rewindow(S, winsize, step);
    scores : Stream Float
    scores = map(scorefun o FFT, wins);
    withscores : Stream (Float, SigSeg Int16)
    withscores = zip2(scores, wins);
    return threshFilter(withscores);
}
fun profileDetect(S : Stream (SigSeg Int16),
            scorefun,
            (winsize,step))
{
    wins = rewindow(S, winsize, step);
    scores : Stream Float
    scores = map(scorefun o FFT, wins);
    withscores : Stream (Float, SigSeg Int16)
    withscores = zip2(scores, wins);
    return threshFilter(withscores);
}
fun profileDetect(S : Stream (SigSeg Int16), scorefun, (winsize, step))
{
    wins = rewindow(S, winsize, step);
    scores : Stream Float
    scores = map(scorefun o FFT, wins);
    withscores : Stream (Float, SigSeg Int16)
    withscores = zip2(scores, wins);
    return threshFilter(withscores);
}
WaveScript Code for Detector

fun profileDetect(Stream (SigSeg Int16), scorefun, (winsize, step)) {
    wins = rewindow(S, winsize, step);
    scores : Stream Float = map(scorefun o FFT, wins);
    withscores : Stream (Float, SigSeg Int16) = zip2(scores, wins);
    return threshFilter(withscores);
}

Ch0 = AudioSource(0, 48000, 1024);
Ch1 = AudioSource(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);
control = profileDetect(Ch0, marmotScore, (64, 192));
datawindows = sync4(control, Ch0, Ch1, Ch2, Ch4);

“Wide-band” Language

http://wavescope.csail.mit.edu/
fun profileDetect(S: Stream (SigSeg Int16), scorefun, (winsize, step))
{
    wins = rewindow(S, winsize, step);
    scores : Stream Float =
        map(scorefun o FFT, wins);
    withscores : Stream (Float, SigSeg Int16) =
        zip2(scores, wins);
    return threshFilter(withscores);
}

Ch0 = AudioSource(0, 48000, 1024);
Ch1 = AudioSource(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);
control = profileDetect(Ch0, marmotScore, (64, 192));
datawindows = sync4(control, Ch0, Ch1, Ch2, Ch4);

“Wide-band” Language

• High-level, query-like declarative programs
• map, project, filter streams
• apply library signal-processing ops
fun profileDetect(S: Stream (SigSeg Int16), scorefun, (winsize,step)) {
    wins = rewindow(S, winsize, step);
    scores : Stream Float
    scores = map(scorefun o FFT, wins);
    withscores : Stream (Float, SigSeg Int16)
    withscores = zip2(scores, wins);
    return threshFilter(withscores);
}

Ch0 = AudioSource(0, 48000, 1024);
Ch1 = AudioSource(1, 48000, 1024);
Ch2 = AudioSource(2, 48000, 1024);
Ch3 = AudioSource(3, 48000, 1024);
control = profileDetect(Ch0, marmotScore, (64,192));
datawindows = sync4(control, Ch0, Ch1, Ch2, Ch4);

“Wide-band” Language

- High-level, query-like declarative programs
  - map, project, filter streams
  - apply library signal-processing ops
- Low-level, imperative code within custom-operators
  - use iterate to introduce

http://wavescope.csail.mit.edu/
Comparing to handwritten C

Emstar vs. Wavescope – Active Event Detector CPU and memory usage %

- Data Acquisition
- Threading Overhead
- Event Detector

Resource Usage % (CPU or memory)

- Emstar CPU
- WS CPU
- Emstar mem
- WS mem
Comparing to handwritten C

Step 2. Direction of arrival search exec times in seconds

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Mean</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>2.85</td>
<td>3.05</td>
<td>3.4</td>
</tr>
<tr>
<td>WS</td>
<td>2.2</td>
<td>2.4</td>
<td>2.8</td>
</tr>
</tbody>
</table>

(Using MLton backend)
Implementation: 

Leveraging the DS in DSL
No shared state between operators.
Implementation: Leveraging the DS in DSL

- No shared state between operators.
- We can manage separate heaps efficiently
  - This suggests an efficient form of delayed reference counting (next slides)
Implementation: *Leveraging the DS in DSL*

- No shared state between operators.
- We can manage separate heaps efficiently
  - This suggests an efficient form of delayed reference counting (next slides)
- Distributed execution
  - Stream graph executes across multiple machines
Implementation: Leveraging the DS in DSL

• No shared state between operators.
• We can manage separate heaps efficiently
  • This suggests an efficient form of delayed reference counting (next slides)
• Distributed execution
  • Stream graph executes across multiple machines
• Intra-machine parallelism (multicore/processor)
  • For example, processing terabytes of offline data
Backends, Cont.

WS Compiler

(Scheme)

Scheme Backend

SML Backend

C++ Backend

XStream Runtime
Backends, Cont.

Currently: naive reference counting
Backends, Cont.

 Currently: naive reference counting
Current work: efficient GC

• Delayed RC: don’t track stack references
  • But then you need to trace occasionally
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
• Delayed RC: don’t track stack references
  • But then you need to trace occasionally

Current work: efficient GC
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
Current work: efficient GC

• Delayed RC: don’t track stack references
  • But then you need to trace occasionally
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
Current work: efficient GC

- Delayed RC: don’t track stack references
  - But then you need to trace occasionally
- We can manage separate heaps efficiently
  - Therefore we can collect at the end of an operator’s execution
Task, Data, and Pipeline Parallelism
(Intra-machine parallelism)
Task, Data, and Pipeline Parallelism
(Intra-machine parallelism)
Task, Data, and Pipeline Parallelism
(Intra-machine parallelism)

Task

- op1
- op2

Data

- split
- F
- F
- join
- map(F, strm)
Task, Data, and Pipeline Parallelism (Intra-machine parallelism)

map(F, strm)
Task, Data, and Pipeline Parallelism  
(Intra-machine parallelism)

- See StreamIT for good work on optimizing parallelism
  - You need information on data rates
- We use PROFILING based on sample data
  - Helps with other optimizations: e.g. data representation transforms

```
Task
   op1
   op2

Data
   split
   F
   F
   F
   join

Pipeline
   op1
   op2

map(F, strm)
```
Further abstract the representation of a stream, enable changing the “glue”.
Building Stream Abstractions

• Further abstract the representation of a stream, enable changing the “glue”.

• Pull based streams.
  • type PullStrm t = Stream () → Stream t;

• Stream transforms with a “pass through” channel (a la StreamIt’s “teleporting messages”)
  • type PassThru (a,b,t) = Strm (a,t) → Strm (b,t)

• Self-marshaling stream operators
• Further abstract the representation of a stream, enable changing the “glue”.

• Pull based streams.
  • type PullStrm t = Stream () → Stream t;

• Stream transforms with a “pass through” channel (a la StreamIt’s “teleporting messages”)
  • type PassThru (a,b,t) = Strm (a,t) → Strm (b,t)

• Self-marshaling stream operators
Building Stream Abstractions

- Further abstract the representation of a stream, enable changing the “glue”.

- Pull based streams.
  - type PullStrm t = Stream () → Stream t;

- Stream transforms with a “pass through” channel (a la StreamIt’s “teleporting messages”)
  - type PassThru (a,b,t) = Strm (a,t) → Strm (b,t)

- Self-marshaling stream operators
End.

wavescope.csail.mit.edu