Sig-adLib

A Compilable Embedded Language for Synchronous Data-Flow Programming on the Java Virtual Machine

Baltasar Trancón y Widemann[1,2] Markus Lepper[2]

[1]Nordakademie, Elmshorn, DE

[2]semantics GmbH, Berlin, DE

(TFP)
Lambda Days
2022-07-28⁄⁄29

Agenda

1. Introduction
2. Design and Implementation
3. Evaluation

Design Goalposts

Sig-adLib — Signal Processing *Ad Libitum*

- managed (JVM-hosted) language
- **dynamic, modular, extensible**
- **functional data flow**
- declarative synchronous paradigm
- conceived as compiler backend (for the textual DSL Sig)

- operate indefinitely on small constant space
- **high realtime performance**
- **procedural control flow**
- imperative processing pragmatics
- productive, fun & educational to use directly

Motivating Example (Java)

```java
float s = 0;

while (!isInterrupted()) {
  float y = input.nextFloat();
  s = s + y;
}
```

Motivating Example (Java)

```java
float s = 0;

while (!isInterrupted()) {
  float y = input.nextFloat();
  s = s + y;
}
```

- This simple algorithm is numerically **bad** . . .

Motivating Example (Java)

```java
float s = 0;

while (!isInterrupted()) {
  float y = input.nextFloat();
  s = s + y;
}
```

```java
float s = 0;
float s2 = 0;
while (!isInterrupted()) {
  float y = input.nextFloat();
  s2 = s2 + y;
  float t = s + s2;
  s2 = s2 + (s - t);
  s = t;
}
```

- This simple algorithm is numerically **bad** . . .
- . . . the accumulating errors can be compensated (Kahan 1965) . . .

Motivating Example (Java)

```java
float s = 0;

while (!isInterrupted()) {
  float y = input.nextFloat();
  s = s + y;
}
```

```java
float s = 0;
float s2 = 0;
while (!isInterrupted()) {
  float y = input.nextFloat();
  s2 = s2 + y;
  float t = s + s2;
  s2 = s2 + (s - t);
  s = t;
}
```

- This simple algorithm is numerically **bad** . . .
- . . . the accumulating errors can be compensated (Kahan 1965) . . .
- . . . but the presentation has not aged well.

What is Wrong With Ancient Fortran Style?

- No *referential transparency*
  - according to SSA analysis, s2 has 5 distinct meanings
- Variables used *before* and *after* updates

What is Wrong With Ancient Fortran Style?

- No *referential transparency*
  - according to SSA analysis, s2 has 5 distinct meanings
- Variables used *before* and *after* updates
- Reasoning about algorithmic concepts very hard
- Not stable under perturbation of assignments

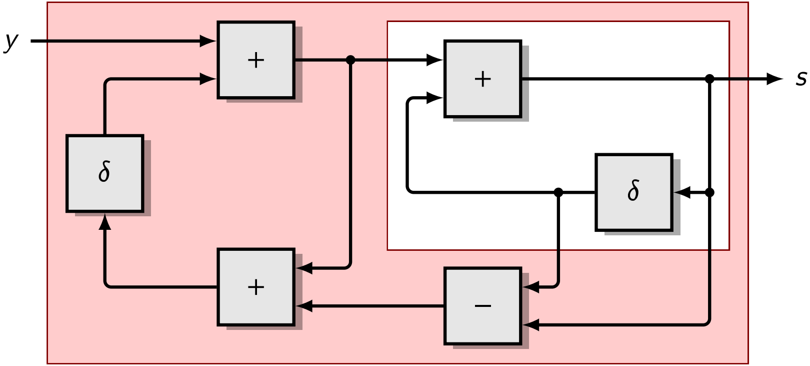
What is Wrong With Ancient Fortran Style?

- No *referential transparency*
  - according to SSA analysis, s2 has 5 distinct meanings
- Variables used *before* and *after* updates
- Reasoning about algorithmic concepts very hard
- Not stable under perturbation of assignments
- More modern escape routes:
  1. graphics (data-flow networks)
  2. algebra (Mealy machines, arrows, etc.)
  3. OOP (data abstraction)

What is Wrong With Ancient Fortran Style?

- No *referential transparency*
  - according to SSA analysis, s2 has 5 distinct meanings
- Variables used *before* and *after* updates
- Reasoning about algorithmic concepts very hard
- Not stable under perturbation of assignments
- More modern escape routes:
  1. graphics (data-flow networks)
  2. algebra (Mealy machines, arrows, etc.)
  3. OOP (data abstraction)
- (Bottom line: Sig-adLib has a bit of each)

Data-Flow Network



δ = single-step delay

Functional Reactive Programming (Rhine)

```haskell
ksum :: (Monad m, Floating a) ⇒ MSF m a a
ksum = mealy step (0, 0)
  where step y (s, s2) = (t, (t, s2b))
          where s2a = s2 + y
                t   = s + s2a
                s2b = (s − t) + s2a
```

- fairly elegant in Mealy style
- (not quite so elegant in point-free arrow style)

“French” Synchronous Languages (Lustre)

```
node ksum (y : real) returns (t : real);
let
    s2a = s2 + y
    t   = s + s2a
    s2  = 0 → pre((s − t) + s2a)
    s   = 0 → pre(t)
tel
```

- Synchronous language operational semantics distinguish time scales:
  - **macro-time** scale of observable changes (clock ticks, stream elements)
  - **micro-time** scale of update propagation
- Program analysis prevents micro-time race conditions & causal errors

Java Streams – Application

```java
int total = shoppingCart.stream()
                .filter(Item::isAvailable)
                .limit(maxOrderSize)
                .map(Item::getPrice)
                .sum();
```

- Create data pipelines from sources, processors and sinks
- The usual higher-order stream functions
- Implicit parallelization pragmatics for “biggish data”

Java Streams – Implementation

```
public interface Stream<A> {
    Spliterator<A> spliterator();
}

public interface Spliterator<A> {
    boolean tryAdvance(Consumer<A> action);
}
```

- Double-action *consumption* fuses *observation* and *transition*
- Pipelines must be linear — no `unzip`!

The Irony

```
double sum()
```

Returns the sum of elements in this stream. **Summation** is a special case of a reduction. If floating-point summation were exact, this method would be equivalent to:

```
return reduce(0, (s, y) → s + y);
```

However, since floating-point summation is not exact, the above code is not necessarily equivalent to the summation computation done by this method. [...] In particular, this method may be **implemented** using **compensated** summation or other technique [sic!] to reduce the error bound in the numerical sum compared to a **simple** summation of **double** values. [emph. added]

Java 8 SE API Documentation, Oracle (2014)

- The Java Stream EDSL uses compensated summation in escapes;
- but the algorithm is **not** expressible **in** the language

Agenda

Sig-adLib Design

- Embedded DSL: no textual syntax, no external toolchain
- Abstract syntax à la OO: self-interpreting Program Object Graphs
- Pure library solution on vanilla JVM
- Distinct APIs for (meta-)programming and execution
- Aspect-oriented: segregated control and data flow

Data-Flow API

```
@FunctionalInterface
public interface FloatSignalSource extends FloatSupplier {

    @Override public float getAsFloat();

}
```

- Encapsulates the *producer* of a signal
- Purely functional API; side effect-free observation
- Specialized for other primitive JVM datatypes + generic

Data-Flow API

```
@FunctionalInterface
public interface FloatSignalSource extends FloatSupplier {

    @Override public float getAsFloat();

}
```

- Encapsulates the *producer* of a signal
- Purely functional API; side effect-free observation
- Specialized for other primitive JVM datatypes + generic

- Constructs for constant signals & stateless lifted operations

```
x.add(y).add(z).divide(constant(3))
```

Control-Flow API

```
public interface Process {

    public void init();
    public void step(RealtimeContext context);

}
```

- Input (clock) events cause transition; no spontaneous termination
- Usage: $(\texttt{init}, (\texttt{step}, \texttt{get}^*)^*)^*$
- Cf. Arduino execution model

Control-Flow API

```
public interface Process {

    public void init();
    public void step(RealtimeContext context);

}
```

- Input (clock) events cause transition; no spontaneous termination
- Usage: $(\texttt{init}, (\texttt{step}, \texttt{get}^*)^*)^*$
- Cf. Arduino execution model

- Constructs for micro-time sequencing, rate conversion, ...

```
p.andThen(q).andThen(r.every(128))
```

- Sig-adLib program **must** specify causal firing sequence

Usage Pattern

```
XSignalSource out1 = ...; ... ZSignalSource outN = ...;
Process main = ...;

main.init();

void runAWhile(RealtimeContext rc) {
    while (needMoreData()) {
        main.step(rc);
        processData(out1.getAsX(), ..., outN.getAsZ());
    }
}
```

- Inversion-of-Control architectural pattern
  - run offline as a batch job (for max throughput, main loop style),
  - in buffer-sized chunks (for balance), or
  - single-step (for min latency, interrupt style)

Synchronization

```
public interface FloatClockedSignalSource
  extends FloatSignalSource, Process {}

abstract class FloatStoredSignalSource
  implements FloatClockedSignalSource {

    protected float value;    // to be written by init & step

    @Override
    public final float getAsFloat() { return value; }
}
```

- Synchronization objects implement **both** APIs
- Clocked operation; no asynchronous updates

Synchronization

```
public interface FloatClockedSignalSource
  extends FloatSignalSource, Process {}

abstract class FloatStoredSignalSource
  implements FloatClockedSignalSource {

    protected float value;    // to be written by init & step

    @Override
    public final float getAsFloat() { return value; }
}
```

- Synchronization objects implement **both** APIs
- Clocked operation; no asynchronous updates

- Constructs for caches, delay registers, stateful components

```
x.subtract(x.delayed(0)).stored()
```

Motivating Example, Revisited

```
FloatClockedSignalSource ksum(FloatSignalSource y) {
    FloatDelay               s    = new FloatDelay(0),
                             s2   = new FloatDelay(0);
    FloatClockedSignalSource s2a  = s2.add(y).stored();
                             t    = s.add(s2a).stored();
    FloatSignalSource        s2b  = s.subtract(t).add(s2a);
    s2.setInput(s2b);
    s.setInput(t);
    return clock(t, s2.andMeanwhile(s)
                      .andMeanwhile(s2a.andThen(t)));
}
```

- (no recursive let in the host language)

Compilation

- Pair every interpreter API method with a JVM bytecode generator
- (Fall back to interpretation if missing)

Compilation

- Pair every interpreter API method with a JVM bytecode generator
- (Fall back to interpretation if missing)

- Traverse POG and inline ruthlessly (no recursion)
- Emit bytecode as heap array (no external resources)

Compilation

- Pair every interpreter API method with a JVM bytecode generator
- (Fall back to interpretation if missing)

- Traverse POG and inline ruthlessly (no recursion)
- Emit bytecode as heap array (no external resources)

- Load and instatiate with standard JVM class loader capability
- (Wait for jit compiler to kick in)

Compilation

- Pair every interpreter API method with a JVM bytecode generator
- (Fall back to interpretation if missing)

- Traverse POG and inline ruthlessly (no recursion)
- Emit bytecode as heap array (no external resources)

- Load and instatiate with standard JVM class loader capability
- (Wait for jit compiler to kick in)

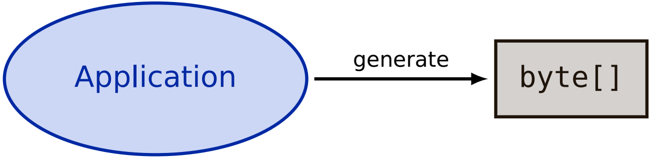
- Fully transparent: interpreted/compiled components use same APIs
- Selective compilation at any granularity

Compilation

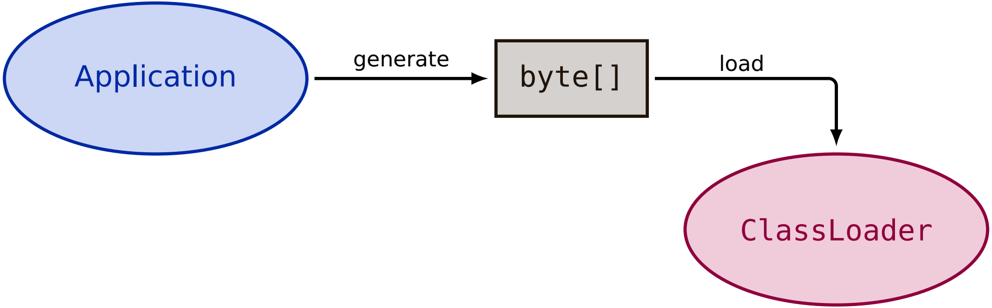
- Pair every interpreter API method with a JVM bytecode generator
- (Fall back to interpretation if missing)
- Traverse POG and inline ruthlessly (no recursion)
- Emit bytecode as heap array (no external resources)
- Load and instatiate with standard JVM class loader capability
- (Wait for jit compiler to kick in)
- Fully transparent: interpreted/compiled components use same APIs
- Selective compilation at any granularity
- Supported by the *LLJava-live* meta-programming library

Java EDSL JIT-Compiler Pipeline [MPLR'21]

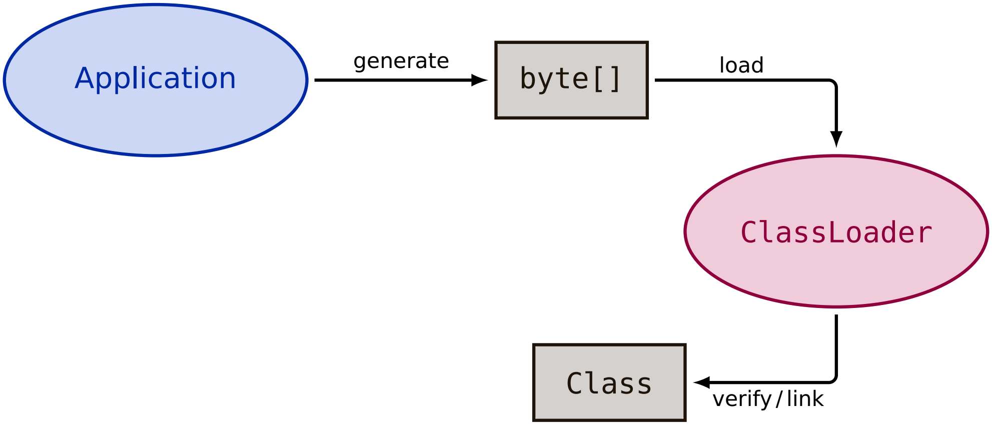
Java EDSL JIT-Compiler Pipeline [MPLR'21]



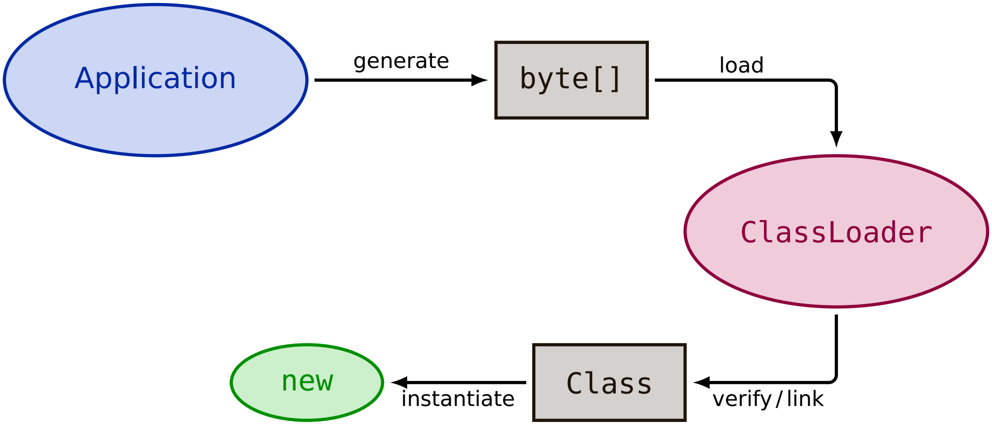
Java EDSL JIT-Compiler Pipeline [MPLR'21]



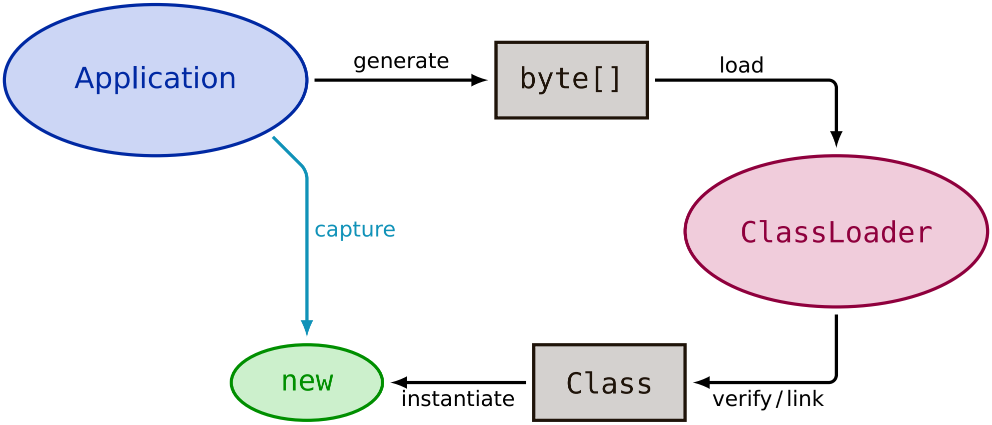
Java EDSL JIT-Compiler Pipeline [MPLR'21]



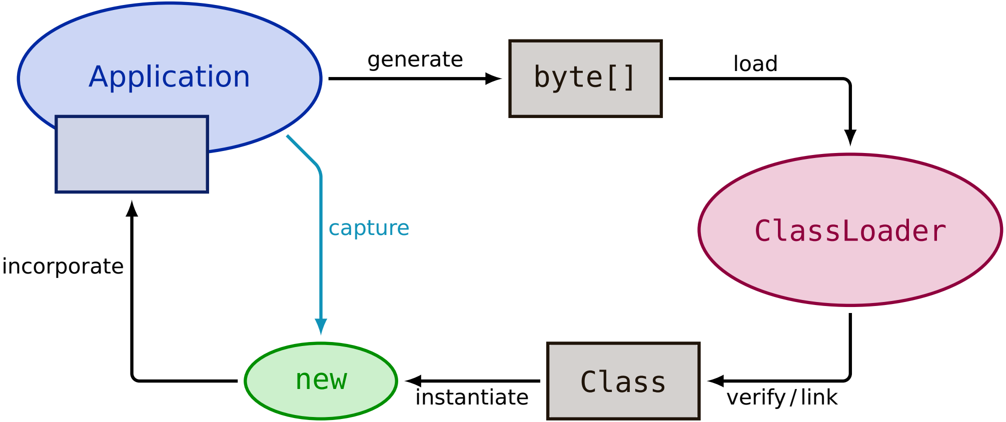
Java EDSL JIT-Compiler Pipeline [MPLR'21]



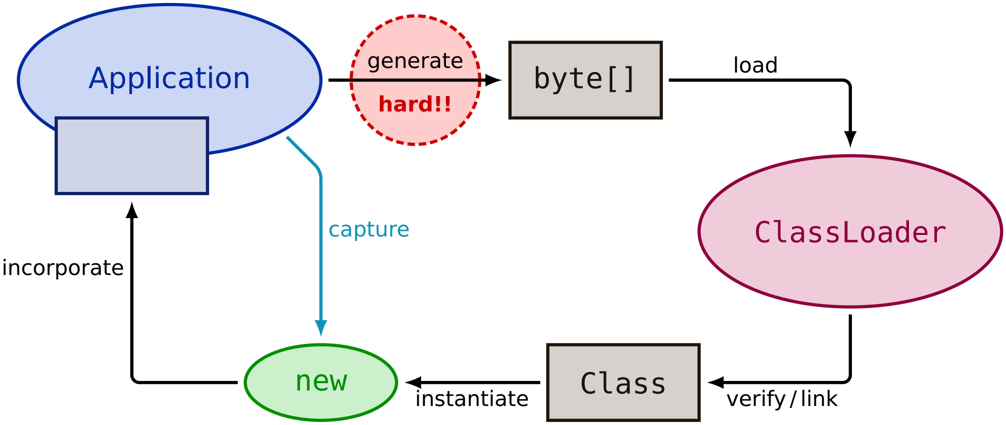
Java EDSL JIT-Compiler Pipeline [MPLR'21]



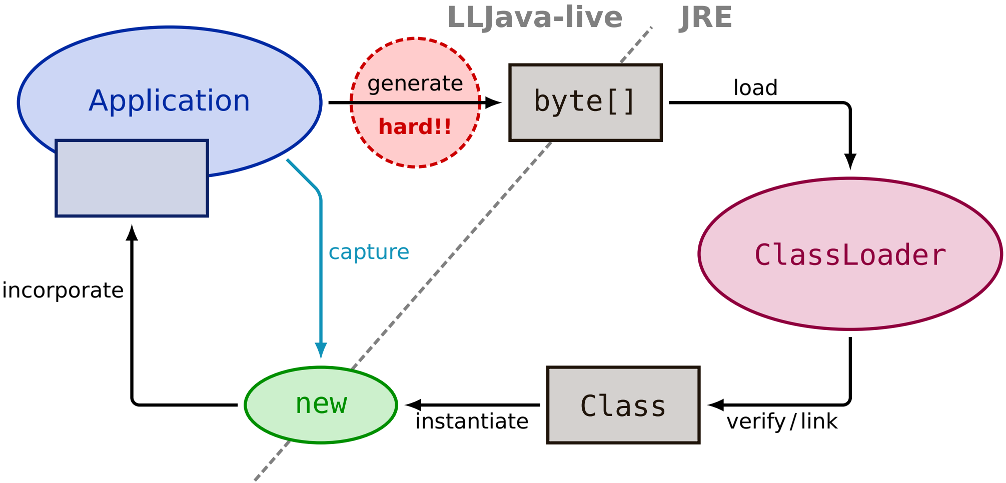
Java EDSL JIT-Compiler Pipeline [MPLR'21]



Java EDSL JIT-Compiler Pipeline [MPLR’21]

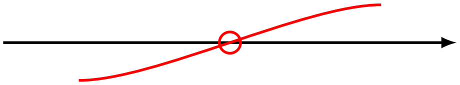


Java EDSL JIT-Compiler Pipeline [MPLR'21]



Agenda

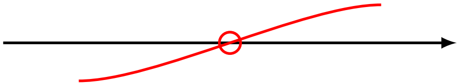
Zero-Crossing Detection



- Simple in (continuous) theory ...

$$\big(\operatorname{sgn} x(t_0 - \delta)\big)\big(\operatorname{sgn} x(t_0 + \delta)\big) = -1 \quad \text{for all } \delta \in (0; \epsilon)$$

Zero-Crossing Detection



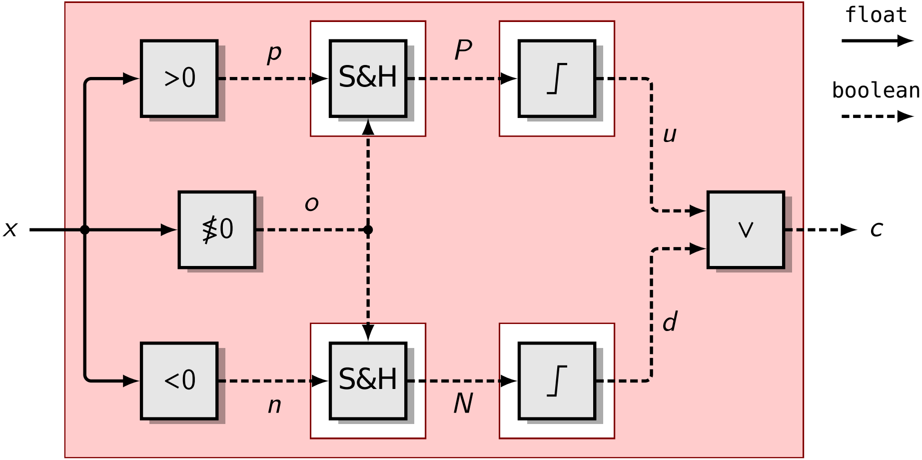
```
3.723453E3, 7.654309E-2, -0.0,
+0.0, NaN, ...
```

- Simple in (continuous) theory ...

$$\big(\operatorname{sgn} x(t_0 - \delta)\big)\big(\operatorname{sgn} x(t_0 + \delta)\big) = -1 \quad \text{for all } \delta \in (0; \epsilon)$$

- ... not so simple in practice:
  - Zeroes *at* vs. *between* discrete sampling times
  - Non-analytic signals can have sustained zero values
  - IEEE floating-point semantics include $\pm 0, \pm\infty, \mathrm{NaN}$
  - Initial conditions matter

Data-Flow Network



Implementation

```java
public BooleanClockedSignalSource zeroCrossing() {
    final FloatClockedSignalSource copy = this.stored();
    final BooleanClockedSignalSource
        neut = copy.guard(zero.or(notANumber)).stored(),
        pos  = copy.guard(positive).sampleAndHold(neut, true),
        neg  = copy.guard(negative).sampleAndHold(neut, true),
        up   = pos.rising(true),
        down = neg.rising(true);
    return up.or(down).stored().after(copy, neut, pos, neg, up, down);
}
```

Implementation

```
public BooleanClockedSignalSource zeroCrossing() {
    final FloatClockedSignalSource copy = this.stored();
    final BooleanClockedSignalSource
        neut = copy.guard(zero.or(notANumber)).stored(),
        pos  = copy.guard(positive).sampleAndHold(neut, true),
        neg  = copy.guard(negative).sampleAndHold(neut, true),
        up   = pos.rising(true),
        down = neg.rising(true);
    return up.or(down).stored().after(copy, neut, pos, neg, up, down);
}

public BooleanClockedSignalSource sampleAndHold(BooleanSignalSource hold,
                                                 boolean initialValue) {
    return delayedFeedback(initialValue, prev → hold.choose(prev, this));
}

public BooleanClockedSignalSource rising(boolean initialValue) {
    final BooleanClockedSignalSource prev = this.delayed(initialValue);
    return this.zipWith((now, before) → now & !before, prev)
               .stored().after(prev);
}
```

Benchmark Results

	Sig-adLib		C
	interpreted	**compiled**	**baseline**
time (ns/elem)	197.1	4.2	4.2
speedup	1	47	47

- Random-walk data; $K = 10^3$ repetitions over $M = 10^6$ elements
- Real averaged time after jit warmup
- Sig-adLib compilation is simply `c = c.compile();`
- OpenJDK jit vs. `gcc -O3 -fno-inline`
- Final speed difference is below measurement precision
  - (but some unexploited potential in loop unrolling)

Conclusion

- Embedded DSL for data-flow programming
  - mostly, but not quite, functional
- No global analysis ⇒ explicit micro-time control flow
  - workflow: data-flow network ▶ causal firing order
- Very tight language integration
  - reactive IoC API, minimal overhead, predictable resources
- Modular and extensible
  - clean OO abstraction
- Transparent compilation support
  - best of both worlds: rapid prototypes + high performance

The End

(Musical Demo Available)

Further Reading

- Trancón y Widemann, B. and M. Lepper (2014). “Foundations of Total Functional Data-Flow Programming”. In: *Proc. MSFP 2014*. Vol. 154. EPTCS, pp. 143–167. DOI: 10.4204/EPTCS.153.10.
- – (2015a). “Laminar Data Flow: On the Role of Slicing in Functional Data-Flow Programming”. In: *Proc. TFP 2015*. Vol. 9547. LNCS. Springer. DOI: 10.1007/978-3-319-39110-6_5.
- – (2015b). “On-Line Synchronous Total Purely Functional Data-Flow Programming on the Java Virtual Machine with Sig”. In: *Proc. PPPJ 2015*. ACM, pp. 37–50. DOI: 10.1145/2807426.2807430.
- – (2021). “LLJava live at the loop: a case for heteroiconic staged meta-programming”. In: *Proc. MPLR 2021*. ACM, pp. 113–126. DOI: 10.1145/3475738.3480942.

Agenda

Code Snippets

JVM JIT-Compiler Disassembly

```
0x7935d141:  vmovss    0x10(%r11,%r9,4),%xmm2       ; load x from array
             ; ...
0x7935d17a:  vxorps    %xmm1,%xmm1,%xmm1
0x7935d17e:  vucomiss  %xmm2,%xmm1                  ; x = 0?
0x7935d182:  jp        0x7935d18a
0x7935d184:  je        0x7935d211                   ; goto side path (0)
0x7935d18a:  vucomiss  %xmm2,%xmm2                  ; x NaN?
0x7935d18e:  jp        0x7935d249                   ; goto side path (NaN)
0x7935d194:  jne       0x7935d249                   ; goto side path (NaN)
0x7935d19a:  movzbl    0x13(%rsi),%r11d             ; load P.prev
0x7935d19f:  movzbl    0x12(%rsi),%r10d             ; load N.prev
0x7935d1a4:  xor       $0x1,%r11d                   ; !P.prev
0x7935d1a8:  xor       $0x1,%r10d                   ; !N.prev
0x7935d1ac:  xor       %r9d,%r9d
0x7935d1af:  mov       $0x1,%ecx
0x7935d1b4:  vucomiss  %xmm2,%xmm1                  ; x > 0?
0x7935d1b8:  mov       $0x1,%ebx
0x7935d1bd:  cmovbe    %r9d,%ebx                    ; p = (x > 0)
0x7935d1c1:  mov       %bl,0x11(%rsi)               ; store p
0x7935d1c4:  mov       %bl,0x13(%rsi)               ; store P
0x7935d1c7:  vucomiss  0xffffff11(%rip),%xmm2       ; x < 0?
0x7935d1cf:  cmovbe    %r9d,%ecx                    ; n = (x < 0)
0x7935d1d3:  mov       %cl,0x10(%rsi)               ; store n
0x7935d1d6:  mov       %cl,0x12(%rsi)               ; store N
0x7935d1d9:  and       %ebx,%r11d                   ; u = P & !P.prev
0x7935d1dc:  and       %ecx,%r10d                   ; d = N & !N.prev
0x7935d1df:  or        %r11d,%r10d                  ; c = u | d
0x7935d1e2:  and       $0x1,%r10d
0x7935d1e6:  mov       %r10l,0x14(%rsi)             ; store c
             ; ...
             ; (side paths)
```

C Baseline Implementation

```c
#include <stdbool.h>

#define K 1000
#define M 1000000

static float data[M];
static int i;
static bool P, N, Pprev, Nprev;
static volatile bool cross;

void zero_cross_init()
{
  P = true;
  N = true;
  Pprev = true;
  Nprev = true;
}

void zero_cross_step()
{
  float x = data[i];
  i = (i + 1) % M;
  bool p = x > 0;
  bool n = x < 0;
  bool o = (x == 0) | (x != x);
  P = o ? P : p;
  N = o ? N : n;
  bool up = P & !Pprev;
  bool down = N & !Nprev;
  Pprev = P;
  Nprev = N;
  cross = up | down;
}
```