Arrp
A Functional Language with Multi-dimensional Signals and Recurrence Equations

Jakob Leben, PhD student
University of Victoria, Canada

Workshop on Functional Art, Music, Modeling and Design (FARM), September 24, 2016, Nara, Japan.

Motivation

- Domains:
  - Audio synthesis and analysis
  - (Video, sensor arrays, communication, multi-media compression)
- Multi-dimensional and multi-rate signals
- Good code reusability
- High performance, real-time execution

Solutions

- Signal = Infinite array (multi-dimensional)
- Array semantics inspired by MATLAB, Octave, Numpy
- Polymorphic, higher-order functions, type inference
- Polyhedral modeling and optimization [1,2,3]

[1] Karp, Miller, Winograd. 1967.
[2] Bondhugula et al. 2008.
[3] Verdoolaege: Integer Set Library (ISL).

Signals as arrays

1: Sine wave:

```
sine_wave(f) = [t -> sin(f*t*2*pi)]
```

2: Differentiation:

```
d(x) = [n -> x[n+1] - x[n]]
```

3: Multi-rate processing:

```
downsample(k,x) = [t -> x[k*t]]
```

Arrp

Recursive arrays

1: Recursive use of name “y”

```
lp(a,x) = y = [
   0 -> 0;
   t -> a * x[t]  +  (1-a) * y[t-1]
]
```

2: Recursion using keyword “this”

```
lp(a,x) = [
  0 -> 0;
  t -> a * x[t]  +  (1-a) * this[t-1]
]
```

Multi-dimensional signals

1: 5 harmonics:

```
a = [5,~: i,t -> sin((i+1)/sr*t*2*pi)];

b = [~,5: t,i -> sin((i+1)/sr*t*2*pi)];
```

2: Differentiation across time or across channels:

```
dt_a = [5,~: i,t -> a[i,t+1] - a[i,t]];

dc_a = [4,~: i,t -> a[i+1,t] - a[i,t]];
```

How to reuse functions “sine_wave” and “d”?

Array currying and partial application

1: 5 harmonics, curried:

```
a = [5: i -> [t -> sin((i+1)/sr*t*2*pi)] ];

b = [t -> [5: i -> sin((i+1)/sr*t*2*pi)] ];
```

2: Partial application:

```
a[3] ## size [~]

b[9] ## size [5]
```

Array currying and partial application

```
sine_wave(f) = [t -> sin(f*t*2*pi)];

d(x) = [n -> x[n+1] - x[n]];
```

1: “sine_wave” uncurried into “a”

```
a = [5: i -> sine_wave((i+1)/sr)];
```

2: “d” uncurried into “dt_a”

```
dt_a = [#a: i -> d(a[i])];
```

Arrp — Pointwise operations and broadcasting

1: Pointwise

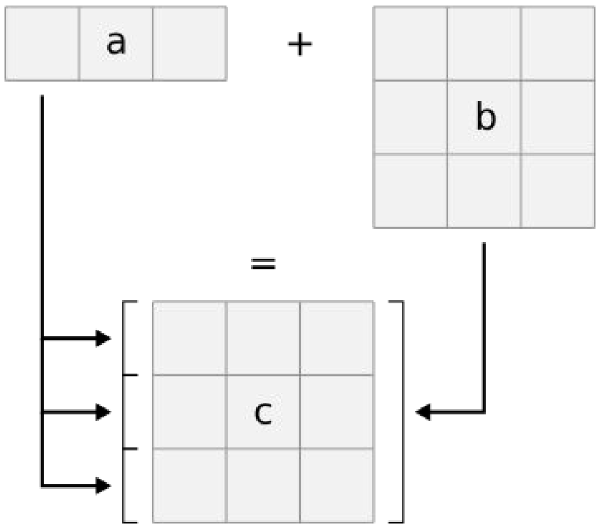
```
a = [5: i -> sine_wave((i+1)/sr)];

d(x) = [n -> x[n+1] - x[n]];

dc_a = d(a); ## size [4,~]
```

2: Pointwise and broadcasting

```
sine_wave(f) = [t -> sin(f*t*2*pi)];

b = sine_wave([5:i -> i+1]/sr);
## size [~,5]
```

Broadcasting

c[i,j] = a[i] + b[i,j]



Array bounds inference

```
a = [5: i -> sine_wave((i+1)/sr)];
```

1: $n < 5$ and $n + 1 < 5 \quad \rightarrow \quad n < 4$

```
d(x) = [n -> x[n+1] - x[n]];

dc_a = d(a);  ## size [4,~]
```

2: $i < 5$

```
dt_a = [i -> d(a[i])]; ## size [5,~]
```

Classic combinators

```
map(f,a) = [i -> f(a[i])];

scan(f,a) = [
    0 -> a[0];
    i -> f(this[i-1], a[i]);
];

fold(f,a) = s[#a-1]
    where s = scan(f,a);
```

Arrp

Abstraction in signal processing

```
delay(v,a) = [0 -> v; t -> a[t-1]];

win(size,hop,x) =
    [t -> [size: k -> x[t*hop + k]]];
```

1: Recursive LP filter:

```
lp(a,x) =
    y = a*x + (1-a)*delay(0,y);
```

2: FIR filter:

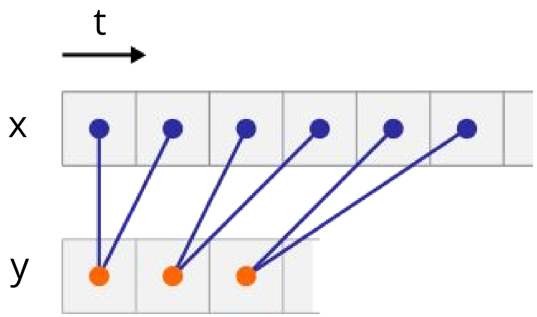
```
sum = fold(\a,b -> a + b);

fir(k) =
    map(\w -> sum(w*k)) . win(#k,1);
```

Polyhedral scheduling

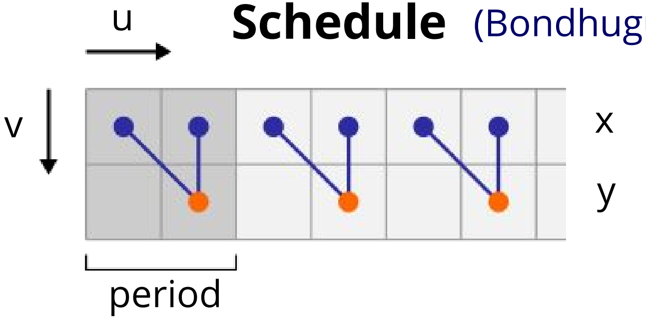
```
x = [t -> f(t)];
y = [t -> x[2*t + 1] - x[2*t]];
```
1

Model

```
for t = 0..
    x[t] = f(t)

for t = 0 ..
    y[t] = x[2*t+1] - x[2*t]
```
2



Schedule (Bondhugula et al. 2008)



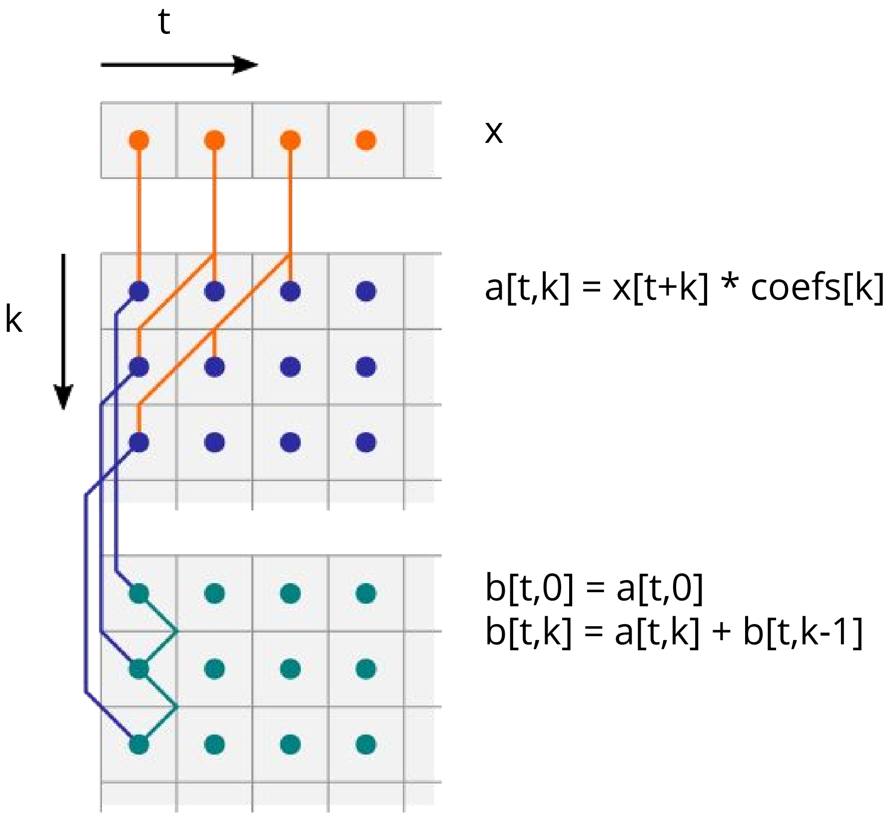

```
for u = 0..
    x[u] = f(u)
    if u % 2 == 1
        y[u/2] = x[u] - x[u-1]
```
3

```
function period() {
    for u = 0..1
        x[u] = f(u)
    y = x[1] - x[0]
    output(y)
}
```
4

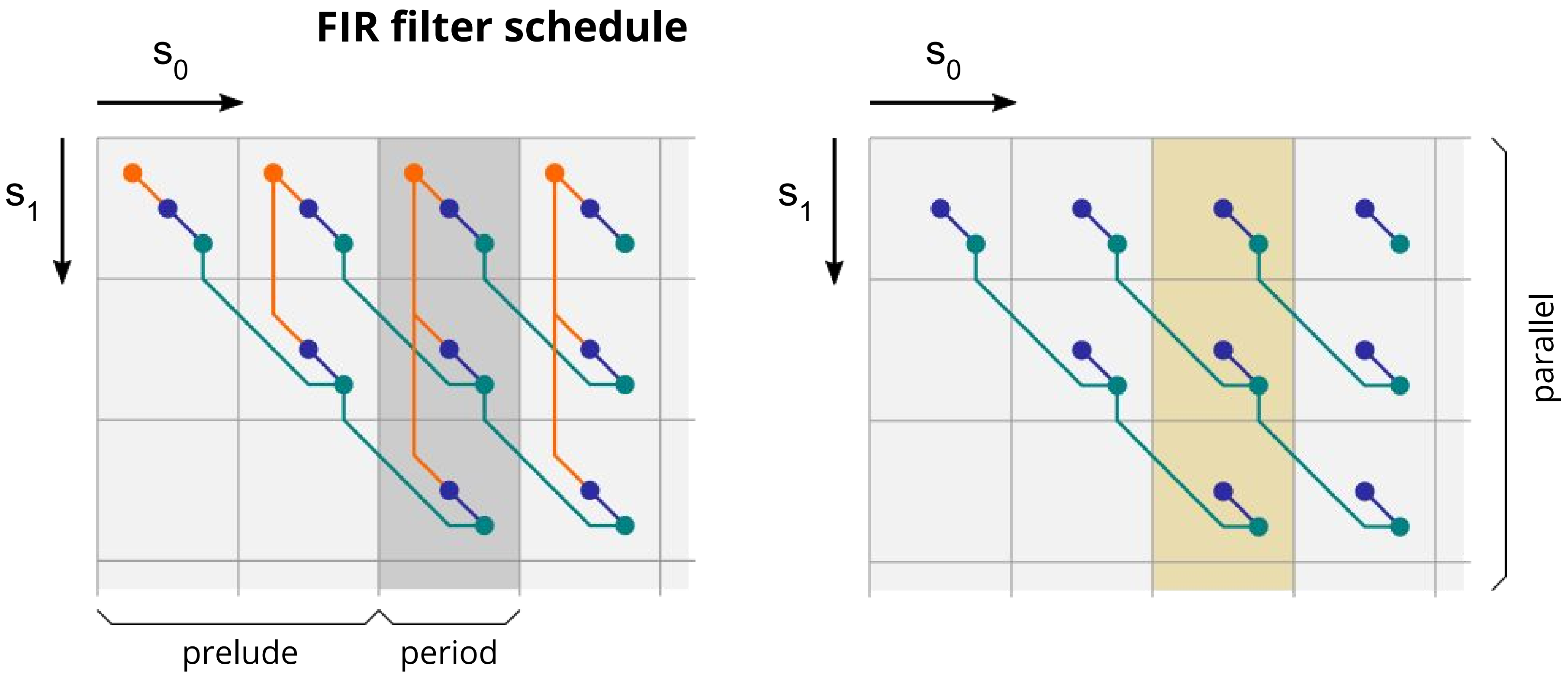
Polyhedral scheduling

FIR filter

```
input x:[~]real64,
      coefs:[10]real64;

map = ...

win = ...

sum = ...

fir(k) =
   map(\w ->sum(w*k))
   . win(#k,1);

main = fir(coefs,x);
```



Polyhedral scheduling



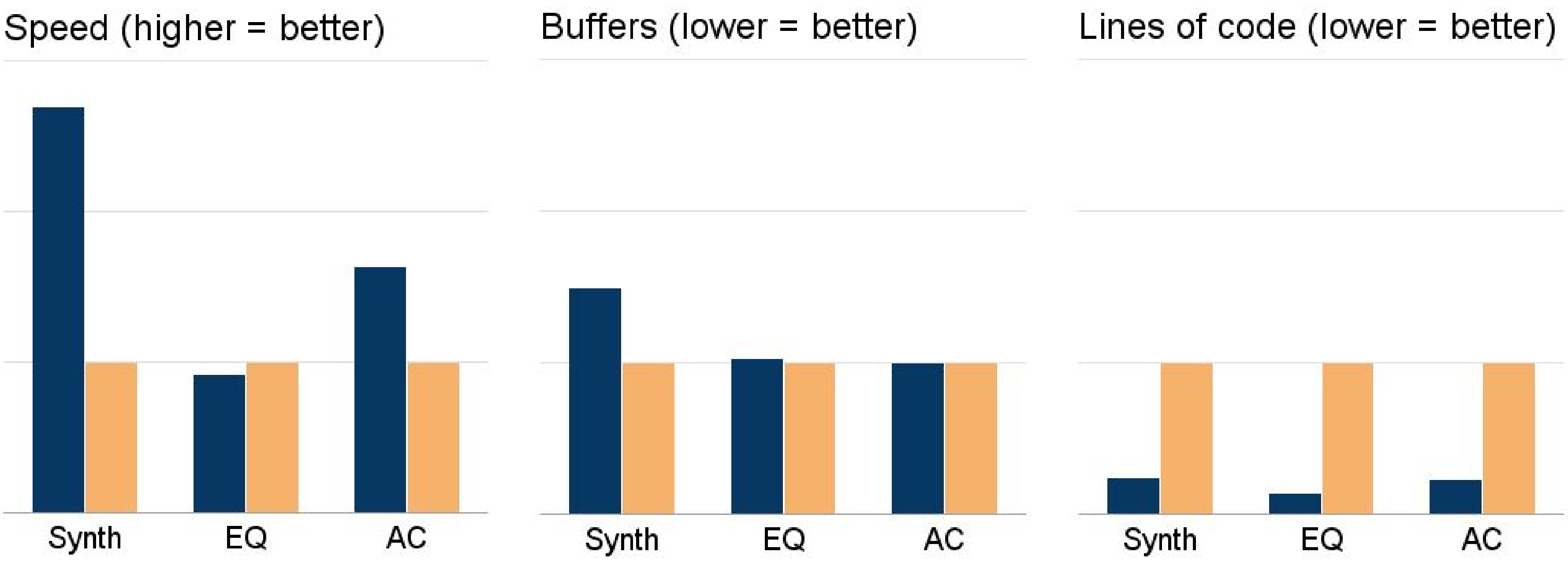
Output code generation

```
template <typename IO>
class alignas(16) program
{
public:
  IO * io;
  void prelude();
  void period();
private:
  double b[4];
  double coefs[3];
  int b_ph = 0;
};

template <typename IO>
inline void
program<IO>::prelude()
{
  ... io->input_coefs(coefs); ...
}
```

```
template <typename IO>
inline void program<IO>::period()
{
  double x;
  double main;
  double a;

  io->input_x(x);
  a = x * coefs[0];
  b[2+b_ph&3] = a;
  for (int c1=1; c1<=2; c1+=1)
  {
    a = x * coefs[c1];
    b[-c1+2+b_ph&3] = b[-c1+2+b_ph&3] + a;
  }
  main = b[0+b_ph];
  io->output(main);
  b_ph = b_ph+1&3;
}
```

Arrp Experimental evaluation

Legend: Arrp Hand-written C++



Future work

Language:

- Algebraic data types
- Recursive functions
- Foreign function calls

Performance:

- Multi-threading, GPU code (in LLVM?)
- Performance comparison: Faust, Kronos, StreamIt, etc.
- Extensive evaluation of polyhedral scheduling

- Applying the principles to other languages?

Further information

Author:

Jakob Leben - jakob.leben@gmail.com

Evaluated Arrp and C++ Code:

http://webhome.csc.uvic.ca/~jleben/farm2016

Arrp Website:

http://mess.cs.uvic.ca/arrp

Arrp Compiler:

https://github.com/jleben/arrp