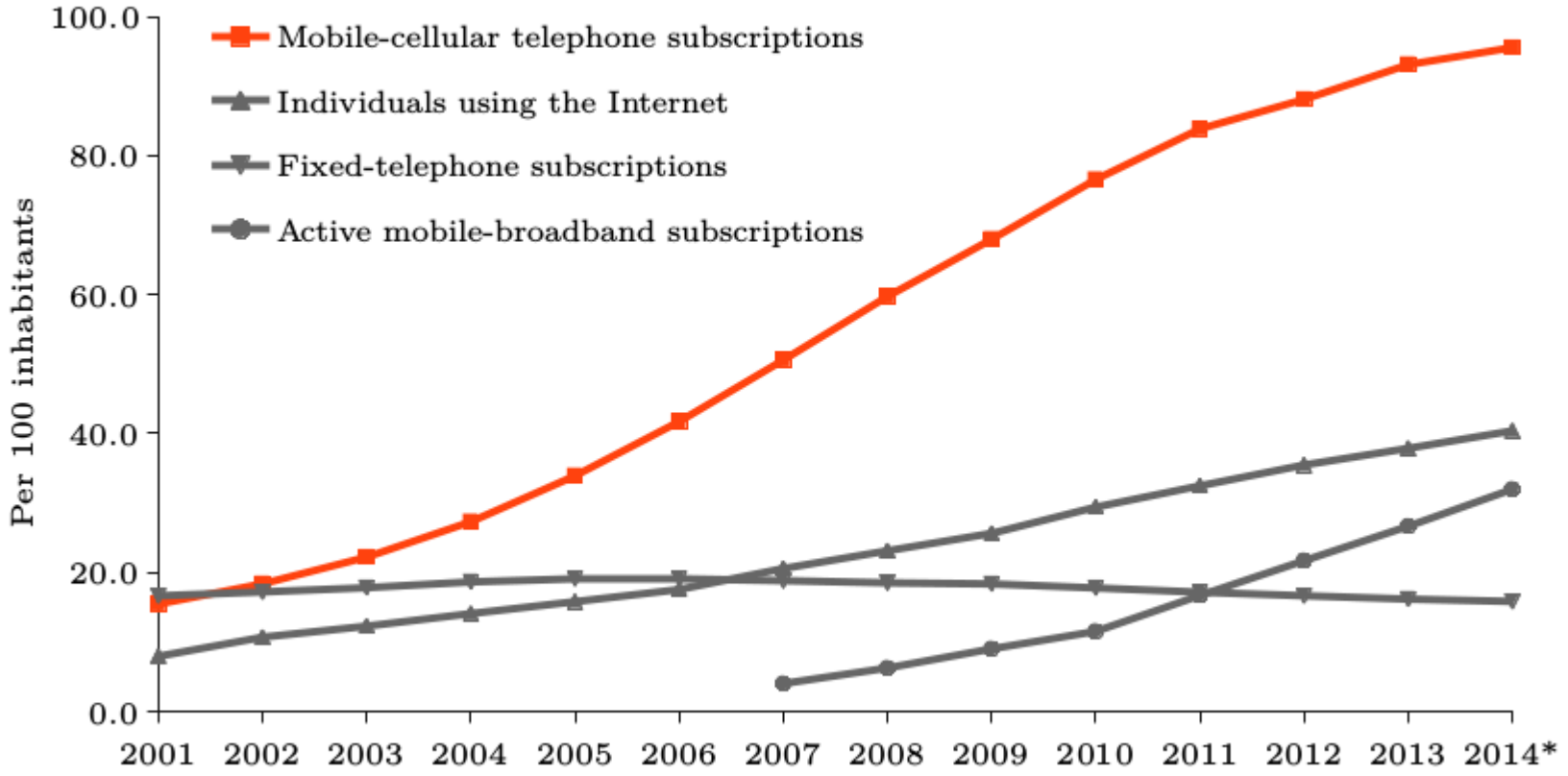
Stream Processing for EDSLs

Markus Aronsson Emil Axelsson Mary Sheeran

Chalmers University of Technology
mararon@student.chalmers.se, {emax, ms}@chalmers.se

Rise of the Telecommunications



Scalar product from AMR codec

ANSI-C specification:

```
for ( j = 0; j < L_frame
    ; j++, p++, p1++)
{
  t0 = L_mac (t0, *p, *p1);
}
corr[-i] = t0;
```

Equivalent loop optimized for specific processor:

Scalar product from AMR codec

ANSI-C specification:

```
for ( j = 0; j < L_frame
    ; j++, p++, p1++)
{
  t0 = L_mac (t0, *p, *p1);
}
corr[-i] = t0;
```

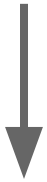
Equivalent loop optimized for specific processor:

```
#pragma MUST_ITERATE(80,160,80);
for (j = 0; j < L_frame; j++)
{
 pj_pj = _pack2 (p[j], p[j]);
 p0_p1 = _mem4_const(&p0[j+0]);
 prod0_prod1 = _smpy2 (pj_pj, p0_p1);
 t0 = _sadd (t0, _hi (prod0_prod1));
 t1 = _sadd (t1, _lo (prod0_prod1));
 p2_p3 = _mem4_const(&p0[j+2]);
 prod0_prod1 = _smpy2 (pj_pj, p2_p3);
 t2 = _sadd (t2, _hi (prod0_prod1));
 t3 = _sadd (t3, _lo (prod0_prod1));
 p4_p5 = _mem4_const(&p0[j+4]);
 prod0_prod1 = _smpy2 (pj_pj, p4_p5);
 t4 = _sadd (t4, _hi (prod0_prod1));
 t5 = _sadd (t5, _lo (prod0_prod1));
 p6_p7 = _mem4_const(&p0[j+6]);
 prod0_prod1 = _smpy2 (pj_pj, p6_p7);
 t6 = _sadd (t6, _hi (prod0_prod1));
 t7 = _sadd (t7, _lo (prod0_prod1));
}
corr[-i] = t0; corr[-i+1] = t1;
 corr[-i+2] = t2; corr[-i+3] = t3;
 corr[-i+4] = t4; corr[-i+5] = t5;
 corr[-i+6] = t6; corr[-i+7] = t7;
```

Feldspar

Feldspar is a pure functional language embedded in Haskell. It offers a high-level dataflow style of programming, as well as a more mathematical style based on vector indices, from which optimized C code is generated.

```
scalarProd
    ::  Vector (Data Float)
     →     Vector (Data Float)
     →     Data Float
scalarProd as bs = sum $ zipWith (*) as bs
```



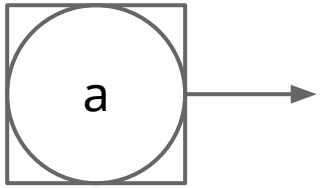
“Optimized C Code”

Our Library*

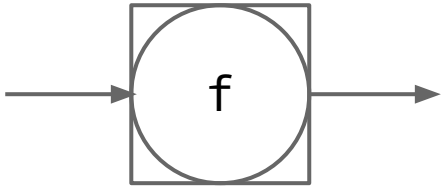
* Name is open to suggestions

Signals - The combinatorial part

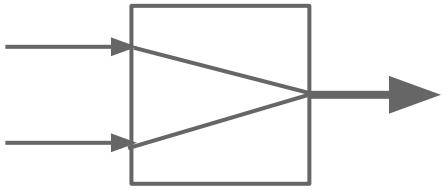
```
repeat :: a → Sig a
```



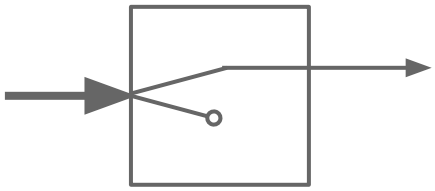

```
map :: (a → b) → Sig a → Sig b
```




```
zip :: Sig a → Sig b → Sig (a,b)
```



```
fst :: Sig (a,b) → Sig a
```



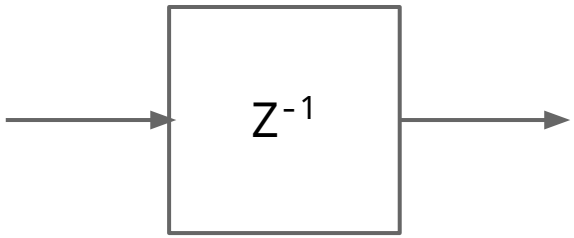
Signals - The combinatorial part

```
instance Num a ⇒ Num (Sig a)
  where
    fromInteger = repeat . fromInteger
    (+)         = zipWith (+)
    (-)         = zipWith (-)
    ...
```

s	1	2	3	4	5	...
u	10	10	9	9	8	...
s + u	11	12	12	13	13	...

Signals - The sequential side

```
delay :: a → Sig a → Sig
a
```

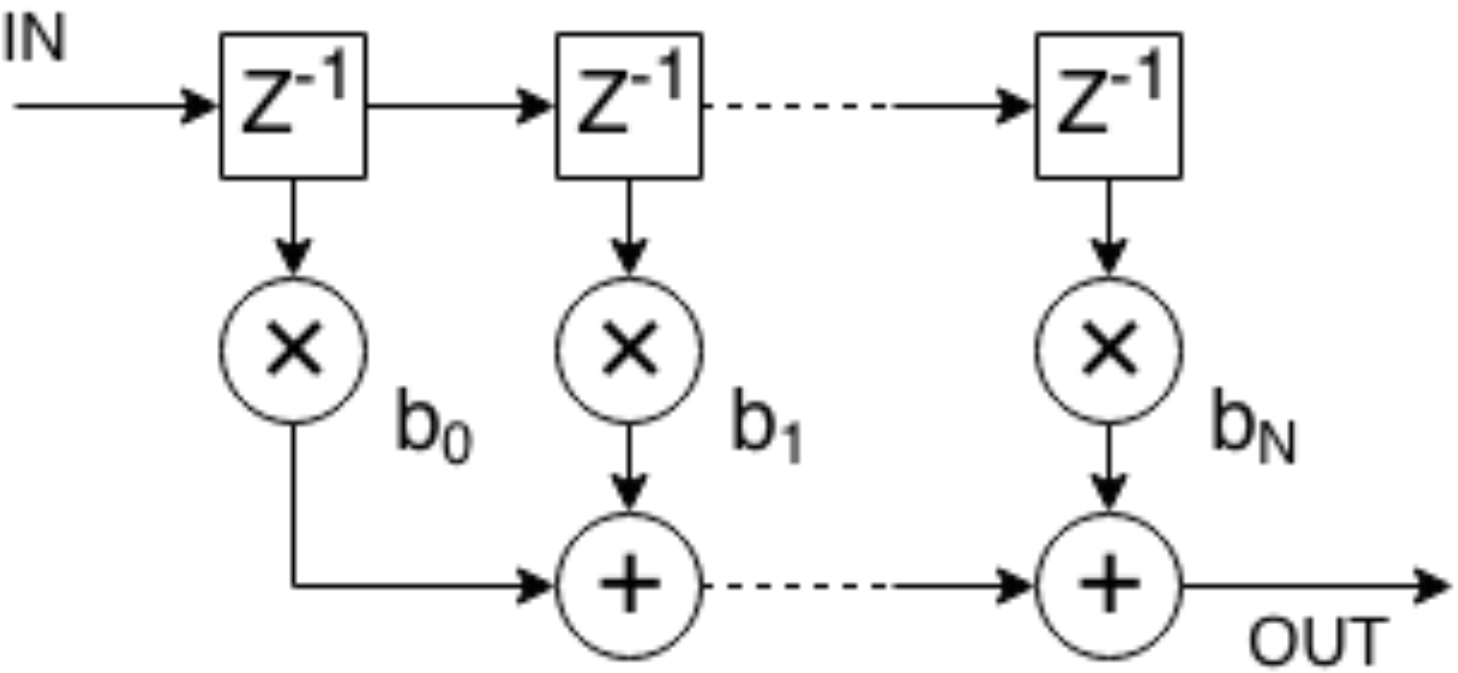



```
edge :: Sig Bool → Sig Bool
edge s = zipWith (/=) s $ delay false s
```

Finite Impulse Response (FIR) Filter

$$y_n = \sum_{i=0}^{N} b_i * x_{n-i}$$

FIR Filter

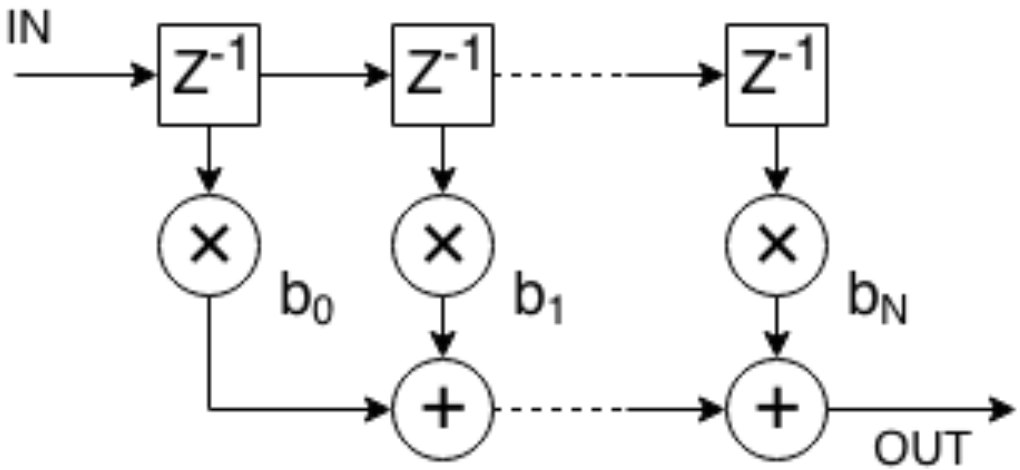


Helpers

```
import qualified Prelude as P


sums :: Num a ⇒ [Sig a] → Sig a
sums = P.foldr1 (+)

muls :: Num a ⇒ [a] → [Sig a] → [Sig a]
muls as = P.zipWith (*) (P.map repeat as)

delays :: [a] → Sig a → [Sig a]
delays as s = P.tail (P.scanl (P.flip delay) s as)
```

FIR Filter

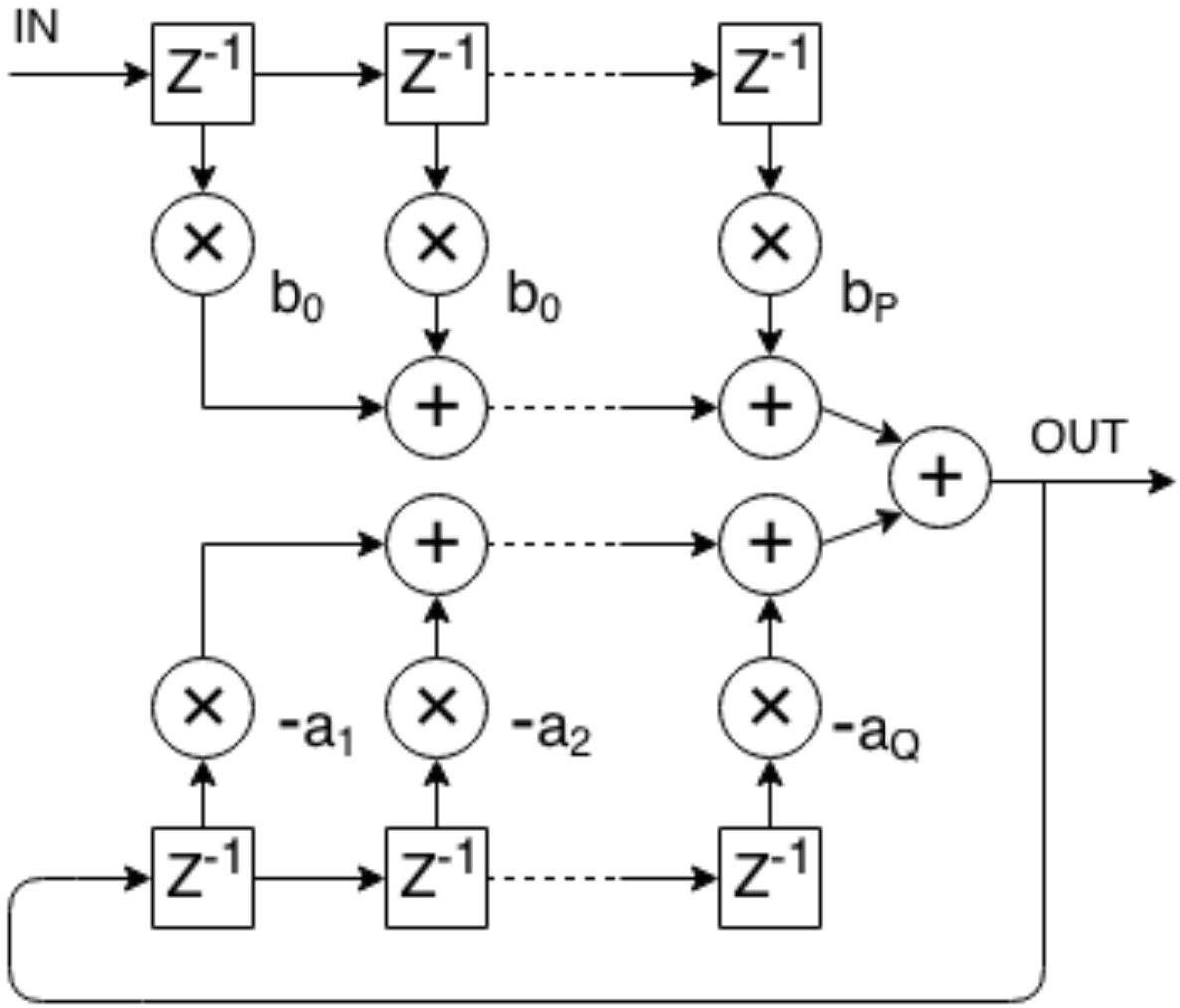
```
fir :: Num a ⇒ [a] → Sig a → Sig a
fir as = sums . muls as . delays ds
  where
    ds = P.replicate (P.length as) 0
```



Infinite Impulse Response (IIR) Filter

$$y_n = \sum_{i=0}^{P} b_i * x_{n-j} - \sum_{j=1}^{Q} a_j * y_{n-j}$$

IIR Filter

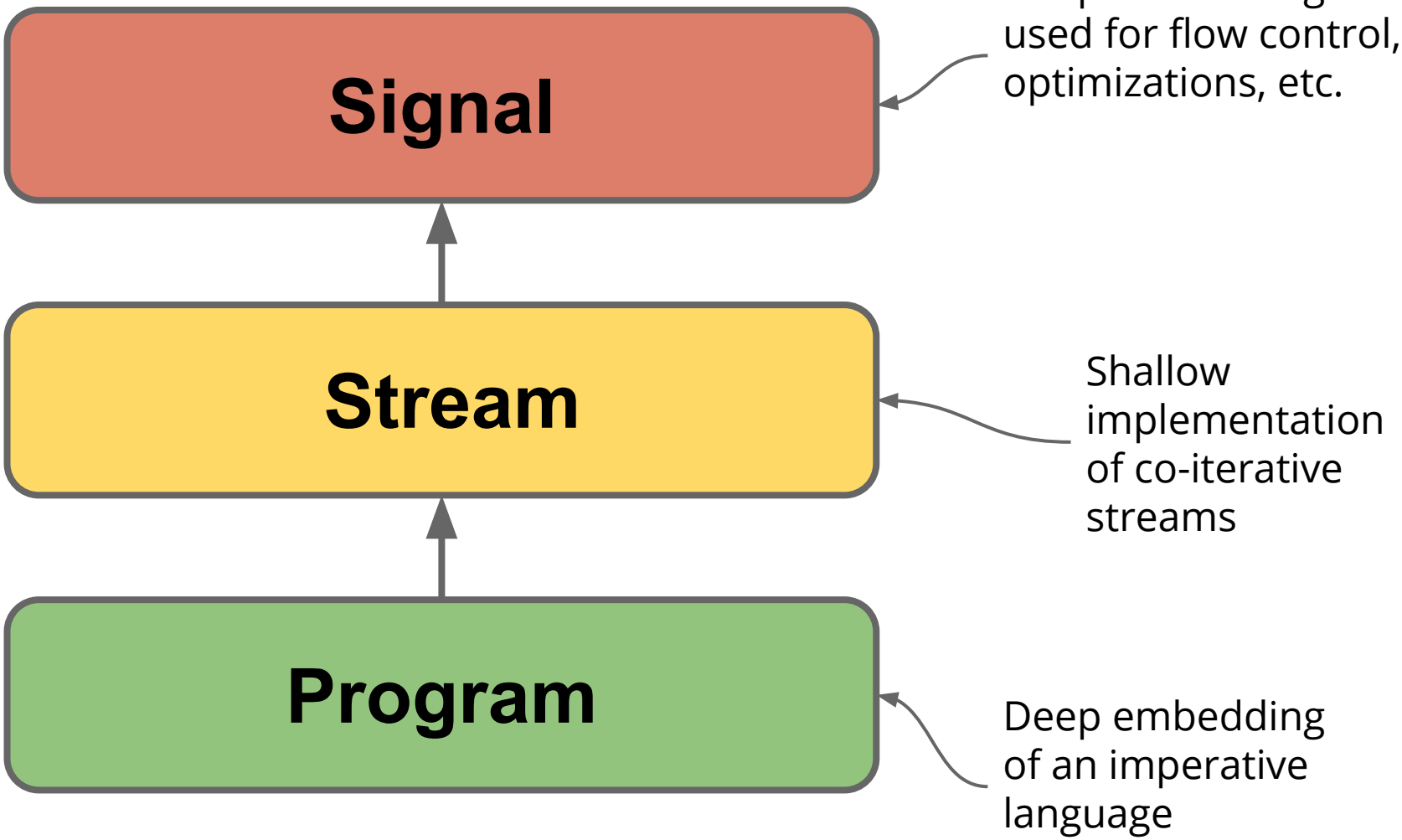


IIR Filter

```
iir :: Num a ⇒ [a] → [a] → Sig a → Sig a
iir as bs s = o
  where
    u = fir bs  s
    l = fir as' o  where as' = map negate as
    o = u + l
```

Implementation

Streams


Signal
Stream
Program
Deep embedding used for flow control, optimizations, etc.
Shallow implementation of co-iterative streams
Deep embedding of an imperative language

Programs

The abstract data type "`Program`" represents programs *, i.e. sequences of primitive instructions.

```
data Program instr a = ...
```

Where instructions are imperative commands of some language:

```
data CMD a
  where
    ReadFile :: FilePath      → CMD (Handle a)
    PutF     :: Handle a → a  → CMD ()
    GetF     :: Handle a      → CMD a
    ...
```

* Operational package by Heinrich Apfelmus

Programs

```
interpretWithMonad :: forall instr m b. Monad m
  ⇒ (forall a. instr a → m a)
  → (Program instr b → m b)
```

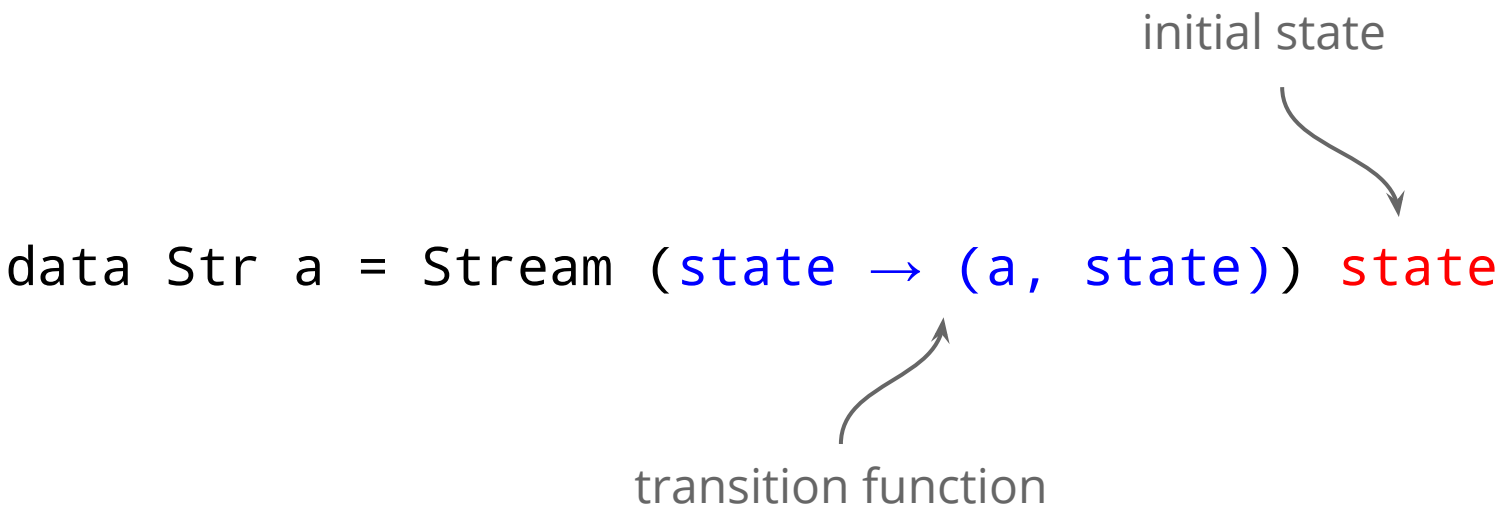
```
compile :: Comp C a ⇒ Program CMD a → C a
compile = interpretWithMonad compileCMD
```

```
run :: Eval a ⇒ Program CMD a → IO a
run = interpretWithMonad runCMD
```

Streams - Original

Co-iteration* consists of associating to a stream

- A transition function from state to pairs of values and state
- An initial state.

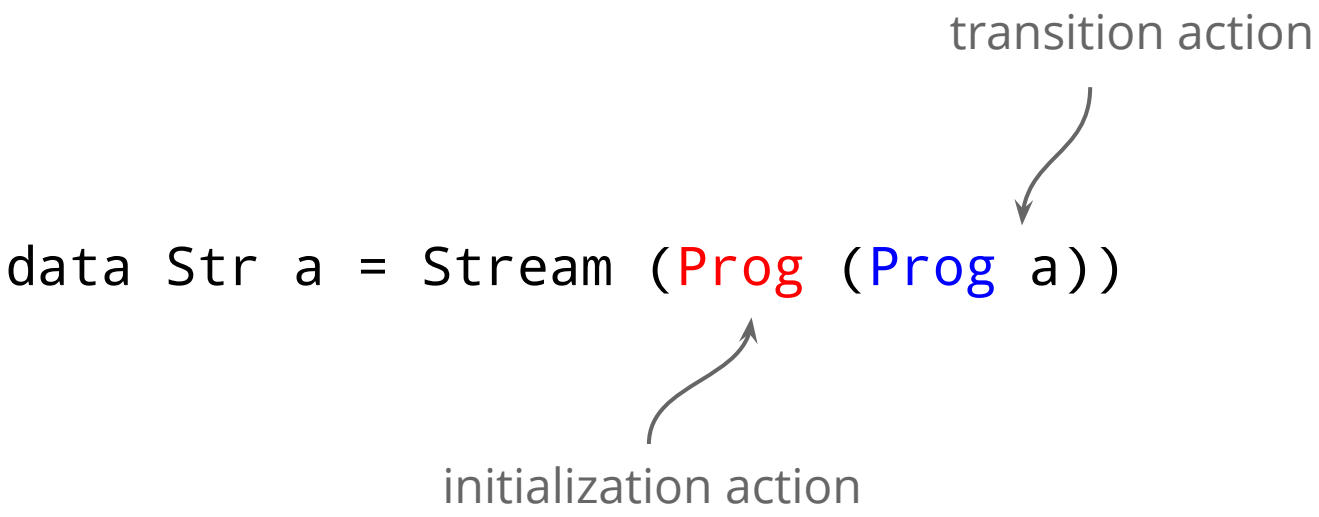


* A Co-iterative Characterization of Synchronous Stream Functions by Paul Caspi

Streams - Our version

Our co-iteration consists of associating to a stream

- A transition action from state to pairs of values and state
- An initialization action.



Streams

```
data Str a = Stream (Prog (Prog a))
```

```
repeat :: a → Str a
repeat a = Stream $ return $ return a
```

```
map :: (a → b) → Str a → Str b
map f (Stream init) = Stream $ do
  next ← init
  return $ do
    v ← next
    return $ f v
```

Signals

```
data Sig a
  where
    ...
    Const ::  Str a → Sig a
    Lift  :: (Str a → Str b) → Sig a → Sig b
```

```
repeat :: a → Sig a
repeat a = Const (Str.repeat a)
```

```
map :: (a → b) → Sig a → Sig b
map f = Lift (Str.map f)
```

More Signals

```
data Sig a
  where
    ...
    Zip :: Sig a → Sig b → Sig (a, b)
    Fst :: Sig (a, b) → Sig a


    zip :: Sig a → Sig b → Sig (a, b)
    zip = Zip

    fst :: Sig (a, b) → Sig a
    fst = Fst

    zipWith :: (a → b → c) → Sig a → Sig b → Sig c
    zipWith f = curry $ lift $ uncurry f
```

Streams - The Sequential Side

```
data Str a = Stream (Prog (Prog a))
```

```
delay :: VarPred a ⇒ a → Str a → Str a
delay v (Stream init) = Stream $ do
  next ← init
  r    ← newRef v
  return $ do
    a ← next
    b ← getRef r
    setRef r a
    return b
```

Can’t we already do that using Streams?

Well yes, but …

- It either requires all feedback networks to make use of a dedicated function for recurrence equations:

```
iir :: Fractional a, VarPred a
                ⇒ a → Vector a → Vector a
                    → Stream a → Stream a
iir a0 a b inp =
  recurrenceIO (replicate1 (length b) 0) inp
               (replicate1 (length a) 0)
               (\i o -> 1 / a0 * ( scalarProd b i
                                 - scalarProd a o)
               )
```

- Or we need to expose “ugly” implementation details

Even more Signals

```
data Sig a
  where
    ...
    Delay :: a → Sig a → Sig a
```

```
delay :: a → Sig a → Sig a
delay = Delay
```

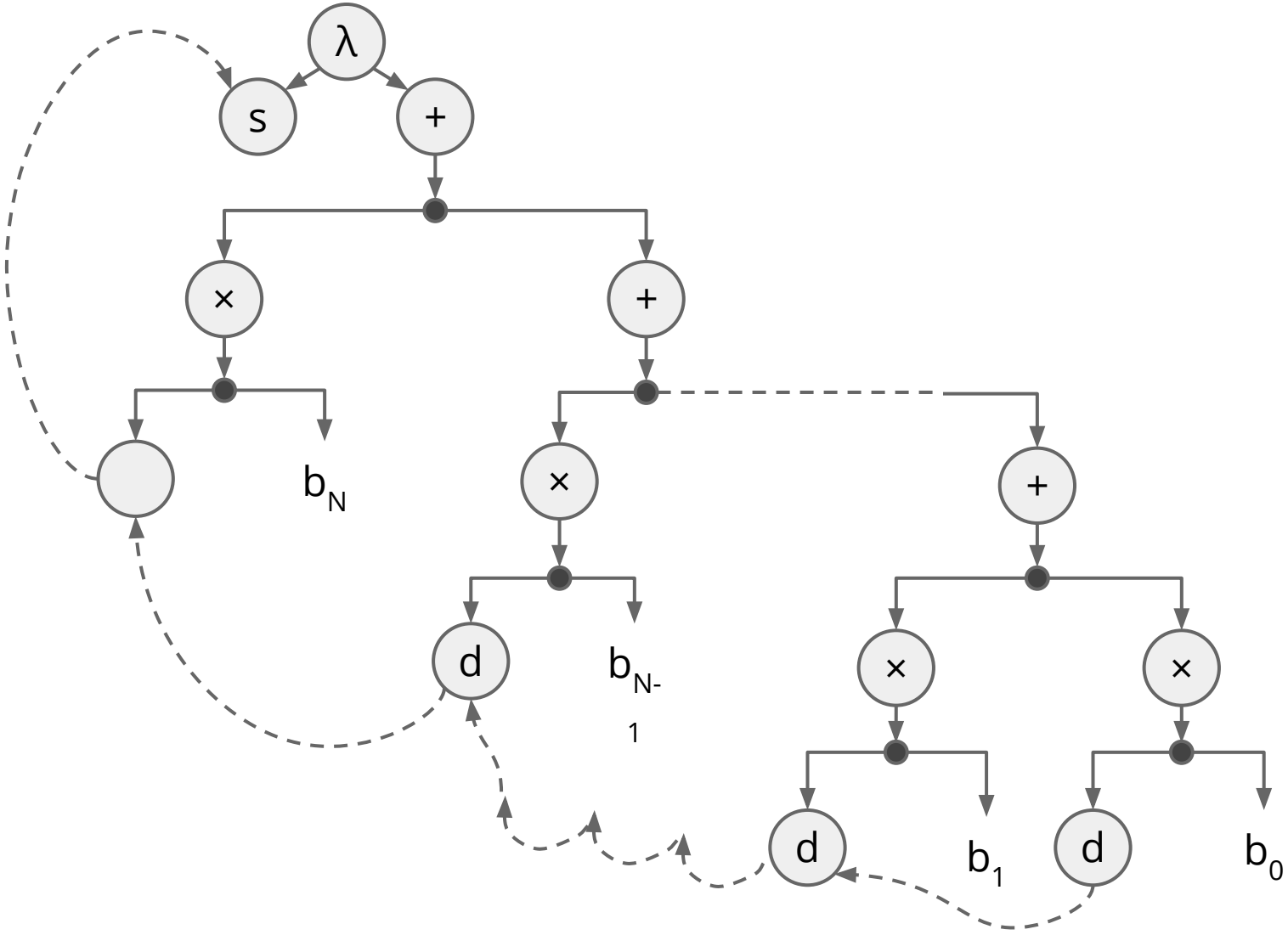
Compiling Signals

```
compile
  :: (Typeable a , Typeable b)
  ⇒    ( Sig a → Sig b )
  → IO ( Str a → Str b )
```

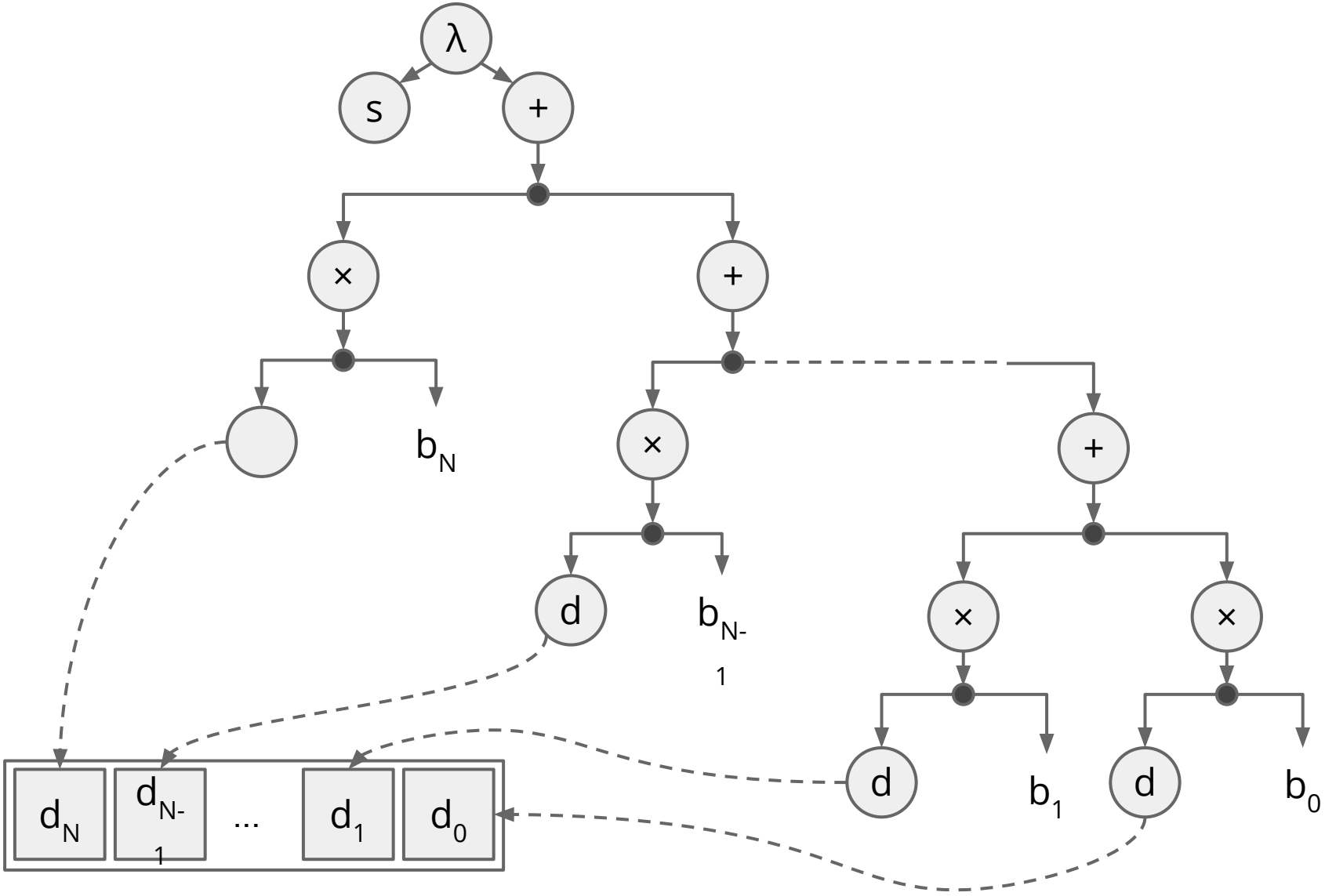
Due to Observable Sharing*

* Type-safe observable sharing in Haskell by Andy Gill

FIR Tree



FIR Tree



```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main();
void main()
{
    FILE* v0;
    FILE* v1;

    float a2[3];
    int r3;

    v0 = fopen("input", "r+");
    v1 = fopen("output", "r+");

    memset(a2, 0.0, sizeof(a2));
    r3 = 0;

    while (true) {

      fprintf(v1, "%f ", r9 + r16);

    }

    fclose(v0);
    fclose(v1);
}
```

```c
fscanf(v0, "%f", &v4);
v5 = feof(v0);
if (v5) { break; }
r6 = v4;
a2[r3] = r6;
if (r3 == 3 - 1) {
  r3 = 0;
} else {
  r3 = r3 + 1;
}
r7 = 1.1;
a8 = a2[(3 + r3 - 1) % 3];
r9 = r7 * a8;
r10 = 1.2;
a11 = a2[(3 + r3 - (1 + 1)) % 3];
r12 = r10 * a11;
r13 = 1.3;
a14 = a2[(3 + r3 - (1 + 1 + 1)) % 3];
r15 = r13 * a14;
r16 = r12 + r15;
```

Conclusion

- Our general idea is that :

  ```
  Deep Embedding + Observable Sharing ⇒ Efficient code
  ```

- Signals are independent of the underlying expression language

  Sig a ~ Signal exp a

- The current compiler is very much in its early stages, as, for example, there is nothing to prevent users from defining malformed signals.

  ```
  bad :: Sig Int
  bad = let s = 1 + s in s
  ```

  Furthermore, while the types support it idea, we have yet to implement signals carrying vectors of values.

- Performance?...