Even Better Stream Fusion

Oleg Kiselyov

Tohoku University, Japan

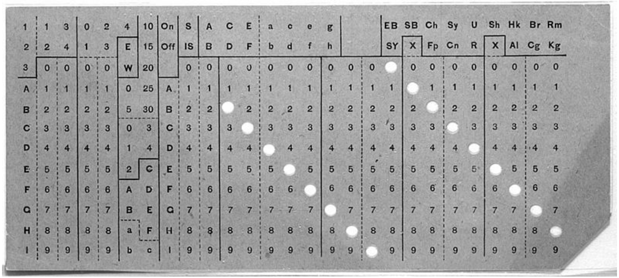
Tensor seminar, 18 February 2022

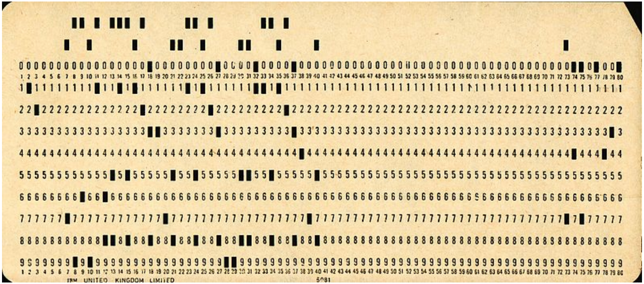
Outline

- **Introduction: What is Stream Processing**
- Stream Fusion
- Strymonas
- Case Study: FM Radio

Tabulating Machine

Punchcard





Tabulating Machine

Stream Processing

- Sequential
- Incremental
- Unbounded amount of data
- Limited memory

The Michael Jackson Design Technique

The Michael Jackson Design Technique: A study of the theory with applications. C.A.R.Hoare, 1977

4.2 Text - Correspondence

The following is a simple problem involving two data structures - one input data structure and one output data structure.

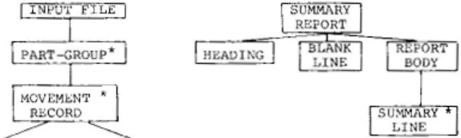
'The stores section in a factory issues and receives parts. Each issue and each receipt is recorded on a punched card: the card contains the part-number, the movement type (I for issue, R for receipt) and the quantity. The cards have already been copied to magnetic tape and sorted into part-number order. The program to be written will produce a simple summary of the net movement of each part. The format of the summary is:

```
STORES MOVEMENTS SUMMARY

A5/132       NET MOVEMENT      -450
A5/197       NET MOVEMENT      1760
B41/728      NET MOVEMENT         7
                    .
                    .
                    .
```

No attention need be paid to such refinements as skipping over the perforations at the end of each sheet of paper.'

The first step of the design procedure, the data step, is to draw data structures of all the files in the problem. The result of the data step is:



Origins of streams in CS

Melvin E. Conway: Design of a Separable Transition-diagram Compiler. Commun. ACM, July *1963*, 396–408

> *A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large sub- set of COBOL on a moderately large computer [10,000-16,000 words]. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.*

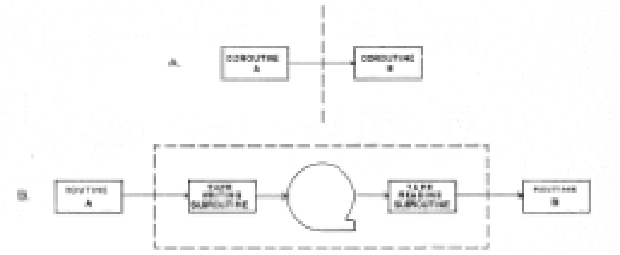
The compiler is written in Assembly by two people in less than a year

Origins of streams in CS

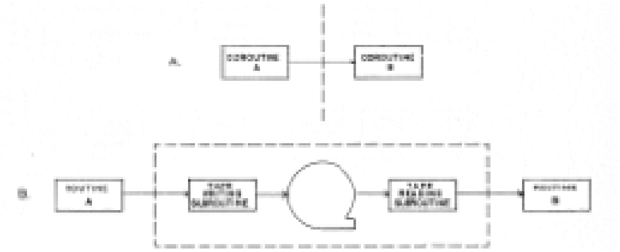
Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its separability. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

Origins of streams in CS



Origins of streams in CS



Can you tell that Jackson wasn't an EE but Conway was?

Stream Processing

Box, with one input and one output

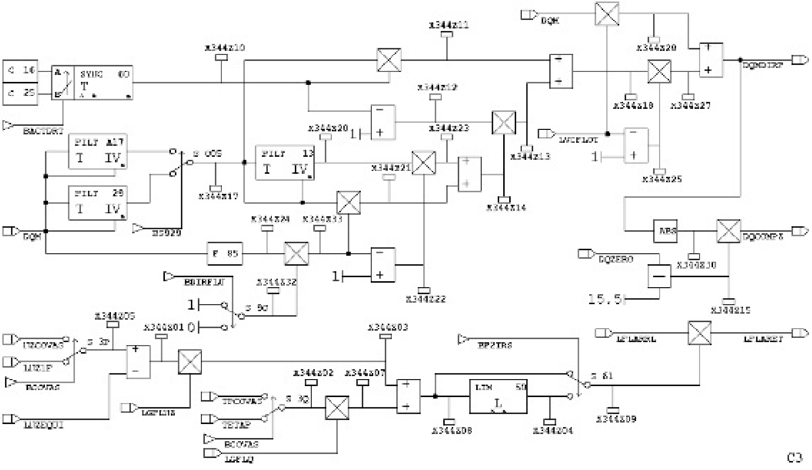
- Sequential
- Incremental
- Unbounded amount of data
- Limited memory

Diagrams

- Connecting the above boxes
- Finite buffering

Sample Diagram

SAO (Spécification Assistée par Ordinateur) — Airbus 80's



Event processing

NEXMark benchmark query 7

Query 7 monitors the highest price items currently on auction. Every ten minutes, this query returns the highest bid (and associated itemid) in the most recent ten minutes.

```
SELECT Rstream(B.price , B.itemid)
FROM
Bid [RANGE 10 MINUTE SLIDE 10 MINUTE] B
WHERE
B.price = (SELECT MAX(B1.price)
FROM BID [RANGE 10 MINUTE SLIDE 10 MINUTE] B1)
LIMIT 1;
```

Window processing

What is Stream Processing

- Record (punchcard) In/Record Out processing
  COBOL-like processing
- Co-routines
- Digital signal processing
- Event processing/correlation
  window processing

Can be represented as a diagram of connected boxes with dataflow left-to-right

What is Stream Processing

- Record (punchcard) In/Record Out processing
  COBOL-like processing
- Co-routines
- Digital signal processing
- Event processing/correlation
  window processing

Can be represented as a diagram of connected boxes with dataflow left-to-right

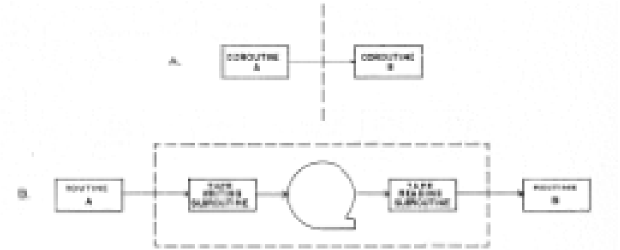
Intuitive design v. performance

Outline

- Introduction: What is Stream Processing
- **Stream Fusion**
- Strymonas
- Case Study: FM Radio

Fusion

Fusion in 1963



Melvin E. Conway: Design of a Separable Transition-diagram Compiler. Commun. ACM, July 1963, 396–408

Pipes

```
cat simple.ml | tr -d "_" | tr "[A-Z]" "[a-z]" |
    grep flatmap | wc -l
```

Pipes

```
cat simple.ml | tr -d "␣" | tr "[A-Z]" "[a-z]" |
    grep flatmap | wc -l
```

```
cat simple.ml |
awk '/[Ff]␣*[Ll]␣*[Aa]␣*[Tt]␣*[Mm]␣*[Aa]␣*[Pp]/ {c++}
     END {print c}'
```

Pipes

```
cat simple.ml | tr -d "_" | tr "[A-Z]" "[a-z]" |
    grep flatmap | wc -l
```

```
cat simple.ml |
awk '/[Ff]_*[Ll]_*[Aa]_*[Tt]_*[Mm]_*[Aa]_*[Pp]/ {c++}
     END {print c}'
```

Perl

Array Programming

$$\sum_{i=0}^{n-1} a_i^2$$

Array Programming

$$\sum_{i=0}^{n-1} a_i^2$$

```
let a = ...
let a2 = map sqr a
sum a2
```

where

```
let sqr : float → float = fun x → x *. x
let map : (α → β) → α array → β array = Array.map
let sum : float array → float = Array.fold_left (+.) 0.
```

Array Programming

$$\sum_{i=0}^{n-1} a_i^2$$

```
let a = ...
sum (map sqr a)
```

where

```
let sqr : float → float = fun x → x *. x
let map : (α → β) → α array → β array = Array.map
let sum : float array → float = Array.fold_left (+.) 0.
```

Array Programming

$$\sum_{i=0}^{n-1} a_i^2$$

```
let a = ...
a ▷ map sqr ▷ sum
```

where

```
let sqr : float → float = fun x → x *. x
let map : (α → β) → α array → β array = Array.map
let sum : float array → float = Array.fold_left (+.) 0.
let (▷) x f = f x
```

Array Programming

$$\sum_{i=0}^{n-1} a_i^2$$

```
let a = ...
a ▷ filter Float.is_finite ▷ map sqr ▷ sum
```

where

```
let sqr : float → float = fun x → x *. x
let map : (α → β) → α array → β array = Array.map
let sum : float array → float = Array.fold_left (+.) 0.
let (▷) x f = f x
let filter : (α →bool) → α array → α array =
fun f x → x ▷ Array.to_list ▷ List.filter f ▷ Array.of_list
```

Array Programming with Fusion

type α arr = A of int * (int$\rightarrow$ α)
let to_arr : α array $\rightarrow$ α arr = fun a $\rightarrow$
A (Array.length a, Array.get a)

Array Programming with Fusion

```
type α arr = A of int * (int→ α)
let to_arr : α array → α arr = fun a →
  A (Array.length a, Array.get a)

let map : (α → β) → α arr → β arr = fun f (A (n,ix)) →
  A(n, ix ▹ f)
let (▹) f g = fun x → f x ▷ g
```

- map is constant time and space

Array Programming with Fusion

```
type α arr = A of int * (int→ α)
let to_arr : α array → α arr = fun a →
   A (Array.length a, Array.get a)

let map : (α → β) → α arr → β arr = fun f (A (n,ix)) →
   A(n, ix ▷ f)

let sum : float arr → float = fun (A (n,ix)) →
  let rec loop acc i = if i ≥ n then acc else loop (acc +. ix i) (i+1)
  in loop 0. 0
```

- map is constant time and space

Array Programming with Fusion

to_arr a ▷ map sqr ▷ sum

```
type α arr = A of int * (int→ α)
let to_arr : α array → α arr = fun a →
  A (Array.length a, Array.get a)

let map : (α → β) → α arr → β arr = fun f (A (n,ix)) →
  A(n, ix ▷ f)

let sum : float arr → float = fun (A (n,ix)) →
 let rec loop acc i = if i ≥ n then acc else loop (acc +. ix i) (i+1)
 in loop 0. 0
```

- map is constant time and space
- No longer any intermediary arrays created

Array Programming with Fusion

to_arr a ▷ map sqr ▷ sum

??? to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

```
type α arr = A of int * (int→ α)
let to_arr : α array → α arr = fun a →
  A (Array.length a, Array.get a)

let map : (α → β) → α arr → β arr = fun f (A (n,ix)) →
  A(n, ix ▷ f)

let sum : float arr → float = fun (A (n,ix)) →
 let rec loop acc i = if i ≥ n then acc else loop (acc +. ix i) (i+1)
 in loop 0. 0
```

- map is constant time and space
- No longer any intermediary arrays created

Array Programming with Filtering and Fusion

to_arr a $\triangleright$ filter Float.is_finite $\triangleright$ map sqr $\triangleright$ sum

Arrays with missing elements

type α option = None | Some of α
type α arr = A of int * (int$\rightarrow$ α option)

let to_arr : α array $\rightarrow$ α arr = ...

Array Programming with Filtering and Fusion

$\text{to\_arr a} \triangleright \text{filter Float.is\_finite} \triangleright \text{map sqr} \triangleright \text{sum}$

type α arr = A of int * (int$\rightarrow$ α option)

let map : $(\alpha \rightarrow \beta) \rightarrow \alpha$ arr $\rightarrow \beta$ arr = fun f (A (n,ix)) $\rightarrow$
  A(n, fun i $\rightarrow$ match ix i with Some y $\rightarrow$ Some (f y) | _ $\rightarrow$ None)

Array Programming with Filtering and Fusion

to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

```
type α arr = A of int * (int→ α option)

let map : (α → β) → α arr → β arr = fun f (A (n,ix)) →
  A(n, fun i → match ix i with Some y → Some (f y) | _ → None)

let sum : float arr → float = ...
```

Array Programming with Filtering and Fusion

to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

type α arr = A of int * (int→ α option)

let map : ($\alpha \to \beta$) → α arr → β arr = fun f (A (n,ix)) →
  A(n, fun i → match ix i with Some y → Some (f y) | _ → None)

let filter : (α →bool) → α arr → α arr = fun f (A (n,ix)) →
  A(n,fun i →
    match ix i with Some y when f y → Some y | _ → None)

Array Programming with Filtering and Fusion

to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

```
type α arr = A of int * (int→ α option)

let map : (α → β) → α arr → β arr = fun f (A (n,ix)) →
  A(n, fun i → match ix i with Some y → Some (f y) | _ → None)

let filter : (α →bool) → α arr → α arr = fun f (A (n,ix)) →
  A(n,fun i →
     match ix i with Some y when f y → Some y | _ → None)
```

The fusion: no unbounded intermediate data structures

Array Programming with Filtering and Fusion

to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

```
type α arr = A of int * (int→ α option)

let map : (α → β) → α arr → β arr = fun f (A (n,ix)) →
  A(n, fun i → match ix i with Some y → Some (f y) | _ → None)

let filter : (α →bool) → α arr → α arr = fun f (A (n,ix)) →
  A(n,fun i →
     match ix i with Some y when f y → Some y | _ → None)
```

The fusion is incomplete

- constant (de)construction of α option (per element)
- overhead of many function calls (per operator)
- higher-order: how to do it in first-order language

Towards complete fusion

to_arr a $\triangleright$ filter Float.is_finite $\triangleright$ map sqr $\triangleright$ sum

Arrays with missing elements, in CPS

type α arr = A of int * (int $\to$ (α $\to$unit) $\to$ unit)

Towards complete fusion

to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

type α arr = A of int * (int → (α →unit) → unit)

let to_arr : α array → α arr = fun a →
  A (Array.length a, fun i k → Array.get a i ▷ k)

Towards complete fusion

to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

```
type α arr = A of int * (int → (α →unit) → unit)

let map : (α → β) → α arr → β arr = fun f (A (n,ix)) →
  A(n, fun i k → ix i (f ▹ k))
```

Towards complete fusion

to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

```
type α arr = A of int * (int → (α →unit) → unit)

let map : (α → β) → α arr → β arr = fun f (A (n,ix)) →
   A(n, fun i k → ix i (f ▷ k))

let sum : float arr → float = fun (A (n,ix)) →
   let sum = ref 0. in
   for i = 0 to n-1 do
      ix i (fun y → sum := !sum +. y)
   done; !sum
```

Towards complete fusion

$\text{to\_arr a} \triangleright \text{filter Float.is\_finite} \triangleright \text{map sqr} \triangleright \text{sum}$

type α arr = A of int * (int → (α →unit) → unit)

let map : ($\alpha \to \beta$) → α arr → β arr = fun f (A (n,ix)) →
A(n, fun i k → ix i (f ▷ k))

let filter : (α →bool) → α arr → α arr = fun f (A (n,ix)) →
A(n,fun i k → ix i (fun y → if f y then k y))

Towards complete fusion

to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

type α arr = A of int * (int → (α →unit) → unit)

let map : ($\alpha \to \beta$) → α arr → β arr = fun f (A (n,ix)) →
  A(n, fun i k → ix i (f ▹ k))

let filter : (α →bool) → α arr → α arr = fun f (A (n,ix)) →
  A(n,fun i k → ix i (fun y → if f y then k y))

The fusion is still incomplete, even got worse

Staged Arrays with missing elements

$\mathsf{type}\ \alpha\ \mathsf{cde} = \mathsf{string}$

$\mathsf{type}\ \alpha\ \mathsf{arr} =$

$\quad \mathsf{A\ of\ int\ cde\ *\ (int\ cde} \to (\alpha\ \mathsf{cde} \to \mathsf{unit\ cde}) \to \mathsf{unit\ cde})$

Array Programming with Complete Fusion

```
type α arr =
     A of int cde * (int cde → (α cde → unit cde) → unit cde)

let to_arr : α array → α arr = fun a →
   A (Array.length a, fun i k → Array.get a i ▷ k)
```

Before (unstaged)

Array Programming with Complete Fusion

```
type α arr =
    A of int cde * (int cde → (α cde → unit cde) → unit cde)

let to_arr : α array cde → α arr = fun a →
  A (sprintf "Array.length %s" a,
    fun i k → sprintf "(Array.get %s %s)" a i ▷ k)
```

Generate the code to evaluate Array.length and Array.get later

Array Programming with Complete Fusion

```
type α arr =
    A of int cde * (int cde → (α cde → unit cde) → unit cde)

let to_arr : α array cde → α arr = fun a →
  A (sprintf "Array.length %s" a,
    fun i k → sprintf "(Array.get %s %s)" a i ▷ k)

let map : (α → β) → α arr → β arr = fun f (A (n,ix)) →
  A(n, fun i k → ix i (f ▷ k))
```

Before (unstaged)

Array Programming with Complete Fusion

```
type α arr =
     A of int cde * (int cde → (α cde → unit cde) → unit cde)

let to_arr : α array cde → α arr = fun a →
   A (sprintf "Array.length %s" a,
     fun i k → sprintf "(Array.get %s %s)" a i ▷ k)

let map : (α cde → β cde) → α arr → β arr = fun f (A (n,ix)) →
   A(n, fun i k → ix i (f ▹ k))
```

Array Programming with Complete Fusion

```
type α arr =
    A of int cde * (int cde → (α cde → unit cde) → unit cde)

let to_arr : α array cde → α arr = fun a →
  A (sprintf "Array.length %s" a,
    fun i k → sprintf "(Array.get %s %s)" a i ▷ k)

let sum : float arr → float = fun (A (n,ix)) →
  let sum = ref 0. in
  for i = 0 to n-1 do
    ix i (fun y → sum := !sum +. y)
  done; !sum
```

Before (unstaged)

Array Programming with Complete Fusion

```
type α arr =
     A of int cde * (int cde → (α cde → unit cde) → unit cde)

let to_arr : α array cde → α arr = fun a →
   A (sprintf "Array.length %s" a,
     fun i k → sprintf "(Array.get %s %s)" a i ▷ k)

let sum : float arr → float cde = fun (A (n,ix)) →
   sprintf
   "let sum = ref 0. in
    for i = 0 to %s-1 do
      %s done; !sum" n
    (ix "i" (fun y → sprintf "sum := !sum +. %s" y))
```

Array Programming with Complete Fusion

```
type α arr =
    A of int cde * (int cde → (α cde → unit cde) → unit cde)

let to_arr : α array cde → α arr = fun a →
  A (sprintf "Array.length %s" a,
    fun i k → sprintf "(Array.get %s %s)" a i ▷ k)

let filter : (α →bool) → α arr → α arr = fun f (A (n,ix)) →
  A(n,fun i k → ix i (fun y → if f y then k y))
```

Before (unstaged)

Array Programming with Complete Fusion

```
type α arr =
    A of int cde * (int cde → (α cde → unit cde) → unit cde)

let to_arr : α array cde → α arr = fun a →
  A (sprintf "Array.length %s" a,
    fun i k → sprintf "(Array.get %s %s)" a i ▷ k)

let filter : (α cde → bool cde) → α arr → α arr = fun f (A (n,ix))
→
  A(n,fun i k → ix i (fun y → sprintf "if %s then %s" (f y) (k y)))
```

Array Programming with Complete Fusion

```
type α arr =
    A of int cde * (int cde → (α cde → unit cde) → unit cde)

let to_arr : α array cde → α arr = fun a →
  A (sprintf "Array.length %s" a,
    fun i k → sprintf "(Array.get %s %s)" a i ▷ k)

let filter : (α cde → bool cde) → α arr → α arr = fun f (A (n,ix))
→
  A(n,fun i k → ix i (fun y → sprintf "if %s then %s" (f y) (k y)))

let app : (α → β) cde → α cde → β cde = fun f x →
  sprintf "(%s %s)" f x
```

Array Programming with Complete Fusion

to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

Before (unstaged)

Array Programming with Complete Fusion

```
let is_finite = app "Float.is_finite"
let sqr = app "sqr"

let v2 = to_arr "a" ▷ filter is_finite ▷ map sqr ▷ sum
```

Array Programming with Complete Fusion

```
let is_finite = app "Float.is_finite"
let sqr = app "sqr"

let v2 = to_arr "a" ▷ filter is_finite ▷ map sqr ▷ sum
```

```
let sum = ref 0. in
for i = 0 to Array.length a-1 do
  if (Float.is_finite (Array.get a i)) then
     sum := !sum +. (sqr (Array.get a i))
done; !sum
```

Outline

Examples of Strymonas

Sum of even squares: sum of squares with filtering

Strymonas

```
C.one_arg_fun @@ fun arr →
    of_arr arr
    ▷ filter C.(fun x → x mod (int 2) = int 0)
    ▷ map C.(fun x → x * x)
    ▷ sum_int
```

generated code

```
fun arg1_49 →
  let t_50 = (Stdlib.Array.length arg1_49) − 1 in
  let v_51 = Stdlib.ref 0 in
  for i_52 = 0 to t_50 do
    (let el_53 = Stdlib.Array.get arg1_49 i_52 in
     if (el_53 mod 2) = 0
     then let t_54 = el_53 * el_53 in v_51 := ((! v_51) + t_54))
  done;
  ! v_51
```

Combinators in two different namespaces

Another simple example

```
let ex1 = iota C.(int 1) ▷ map C.(fun e → e * e)
(* val ex1 : int cstream = <abstr> *)

let sum_int = fold C.(+) C.(int 0)
(* val sum_int : int cstream → int cde = <fun> *)

let ex2 = ex1 ▷ filter C.(fun e → e mod (int 17) > int 7)
              ▷ take C.(int 10) ▷ sum_int
```

generates

Another simple example

```
let ex1 = iota C.(int 1) ▷ map C.(fun e → e ∗ e)
(∗ val ex1 : int cstream = <abstr> ∗)

let sum_int = fold C.(+) C.(int 0)
(∗ val sum_int : int cstream → int cde = <fun> ∗)

let ex2 = ex1 ▷ filter C.(fun e → e mod (int 17) > int 7)
              ▷ take C.(int 10) ▷ sum_int
```

generates

```
let v_1 = Stdlib.ref 0 in
(let v_2 = Stdlib.ref 10 in
 let v_3 = Stdlib.ref 1 in
 while (! v_2) > 0 do
   let t_4 = ! v_3 in
   Stdlib.incr v_3;
   (let t_5 = t_4 ∗ t_4 in
    if (t_5 mod 17) > 7 then (Stdlib.decr v_2; v_1 := ((! v_1) + t_5)))
   done);
! v_1
```

Another simple example

```
let ex1 = iota C.(int 1) ▷ map C.(fun e → e * e)
(* val ex1 : int cstream = <abstr> *)

let sum_int = fold C.(+) C.(int 0)
(* val sum_int : int cstream → int cde = <fun> *)

let ex2 = ex1 ▷ filter C.(fun e → e mod (int 17) > int 7)
              ▷ take C.(int 10) ▷ sum_int
```

generates

```
int cfun()
{ int v_1 = 0; int v_2 = 10; int v_3 = 1;
  while (v_2 > 0)
  { int t_4; int t_5;
    t_4 = v_3;
    v_3++;
    t_5 = t_4 * t_4;
    if ((t_5 % 17) > 7)
    { v_2--; v_1 = v_1 + t_5; }
  }
  return v_1;}
```

Database join

T_1: string * int table, T_2: int * float table
select T_1.1, 2*T_2.2 from T_1, T_2 where T_1.2=T_2.1 and T_2.2 > 5.0

```
let cart (s1,s2) =
 s1 ▷ flat_map (fun e1 → s2 ▷ Raw.map_raw' (fun e2 → (e1,e2))) in

let join (t1,t2) =
 cart (of_arr t1, of_arr t2) ▷

 (* WHERE clauses *)
 Raw.filter_raw C.(fun (e1,e2) → snd e1 = fst e2) ▷
 Raw.filter_raw C.(fun (e1,e2) → truncate (snd e2) > int 5) ▷

 (* SELECTion *)
 Raw.map_raw' C.(fun (e1,e2) → pair (fst e1) (snd e2 *. float 2.)) ▷

 (* Output *)
 iter (fun (e1,e2) → seq (print e1) (print_float e2))
```

A weird test

```
let square x = C.(x * x) and
    even x = C.(x mod (int 2) = int 0) in
Raw.zip_raw
  (* First stream to zip *)
  ([| 0;1;2;3|] ▷ of_int_array
    ▷ map square
    ▷ take (C.int 12)
    ▷ filter even
    ▷ map square)
  (* Second stream to zip *)
  (iota (C.int 1)
    ▷ flat_map (fun x →
        iota C.(x+int 1) ▷ take (C.int 3))
    ▷ filter even)
  ▷ iter C.(fun (x,y) → seq (print_int x) (print_int y))
```

A weird test: result

```
let t_71 = [| 0;1;2;3| ] in
let v_70 = ref 12 in
let v_72 = ref 0 in
let v_73 = ref 1 in
while ((! v_70) > 0) && ((! v_72) ≤ 3) do
  let t_77 = ! v_73 in
  incr v_73;
  (let v_78 = ref 3 in
   let v_79 = ref (t_77 + 1) in
   while ((! v_78) > 0) && (((! v_70) > 0) && ((! v_72) ≤ 3)) do
     decr v_78;
     (let t_80 = ! v_79 in
      incr v_79;
      if (t_80 mod 2) = 0
      then
        (let v_81 = ref true in
         while ! v_81 do
           (decr v_70;
            (let el_82 = Array.get t_71 (! v_72) in
             let t_83 = el_82 * el_82 in
             if (t_83 mod 2) = 0
             then
               let t_84 = t_83 * t_83 in
               (v_81 := false;
                (Format.print_int t_84;
                 Format.force_newline ());
                Format.print_int t_80;
                Format.force_newline ()));
            incr v_72);
           v_81 := ((! v_81) && (((! v_70) > 0) && ((! v_72) ≤ 3))) done))
     done)
  done
```

Stateful Streams

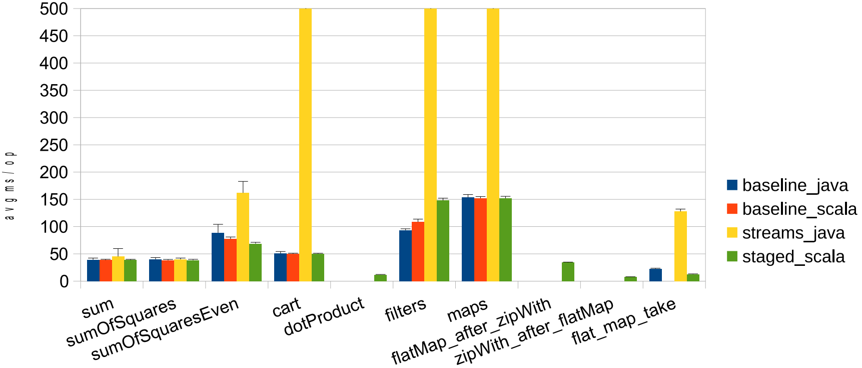
Difference encoder

```
let diff : int cstream → int cstream = fun st →
  initializing_ref C.(int 0) @@ fun z →
  map C.(fun e → letl (e − dref z) @@ fun v → seq (z := e) v) st
```

take_while

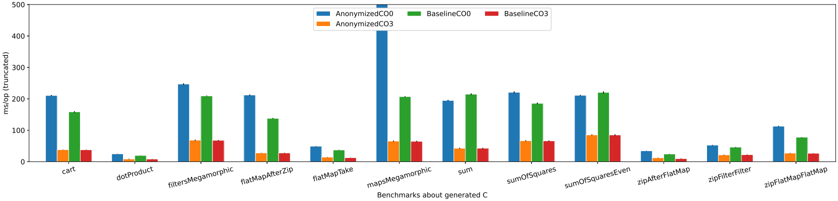
```
let take_while : (α cde → bool cde) → α cstream → α cstream = fun f st →
  initializing_ref C.(bool true) @@ fun zr →
  st ▷ map_raw C.(fun e k → if_ (f e) (k e) (zr := bool false)) ▷ guard C.(dref zr)
```

Results: JVM



500
450
400
350
300
250
200
150
100
50
0
avg ms/op
sum
sumOfSquares
sumOfSquaresEven
cart
dotProduct
filters
maps
flatMap_after_zipWith
zipWith_after_flatMap
flat_map_take
baseline_java
baseline_scala
streams_java
staged_scala

Results: C


AnonymizedCO0
AnonymizedCO3
BaselineCO0
BaselineCO3
ms/op (truncated)
0
100
200
300
400
500
cart
dotProduct
filtersMegamorphic
flatMapAfterZip
flatMapTake
mapsMegamorphic
sum
sumOfSquares
sumOfSquaresEven
zipAfterFlatMap
zipFilterFilter
zipFlatMapFlatMap
Benchmarks about generated C

Outline

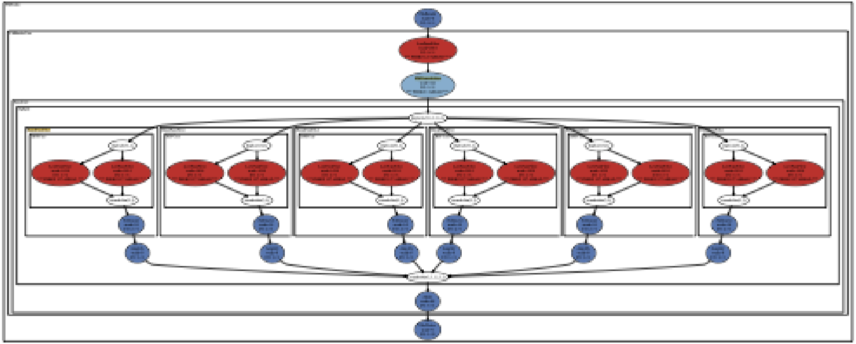
Introduction: What is Stream Processing

Stream Fusion

Strymonas

- **Case Study: FM Radio**

Software FM Radio



William Thies. PhD Thesis, MIT, 2009

Software FM Radio in Strymonas

```
let samplingRate = 250_000_000.
let cutoffFrequency = 108_000_000.
let numberOfTaps = 64
let maxAmplitude = 27_000.
let bandwidth     = 10_000.

let numIters = C.int 1_000_000

let () =
  C.newref C.(float 0.) (fun out →
    get_floats
    ▷ lowPassFilter samplingRate cutoffFrequency numberOfTaps 4
    ▷ fmDemodulator samplingRate maxAmplitude bandwidth
    ▷ equalizer samplingRate bands eqCutoff eqGain numberOfTaps
    ▷ take numIters
    ▷ iter C.(fun e → out:=e)
  )
  ▷ C.print ~name:"fmradio"
```

Basic idea: filtering

```
let lowPassFilter : float → float → int → int → float cstream → float cstream =
  fun rate cutoff taps decimation st →
    let mk_coeff_arr cutoff = ...
    in
    let (module Win) = Window1.make_window taps decimation in
    st
    ▷ Win.make_stream C.tfloat
    ▷ map_raw (fun win →
       C.letl (Win.dot C.tfloat (mk_coeff_arr cutoff) C.( +. ) C.( *. ) win))
```

Conclusions

- Stream processing is varied: EE, CS, MBA,...
- Stream fusion is important and nontrivial
  especially complete stream fusion
- Strymonas can do it

Team

Joint work with

Aggelos Biboudis, Tomoaki Kobayashi, Nick Palladinos, and Yannis Smaragdakis