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


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

<table>
<thead>
<tr>
<th>Item</th>
<th>NET MOVEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5/132</td>
<td>-450</td>
</tr>
<tr>
<td>A5/197</td>
<td>1760</td>
</tr>
<tr>
<td>B4/728</td>
<td>7</td>
</tr>
</tbody>
</table>

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


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.
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 ten minutes.

```sql
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

Avoid the overhead of data serialization and transport.

5.1. Example
Consider a security application that continuously scrutinizes system logs to detect security breaches. The application contains an operator \( A \) that parses the log messages, followed by a selection operator \( B \) that uses a simple heuristic to filter out log messages that are irrelevant for the security breach detection. Assume that the two operators run on separate cores, and that the selection operator \( B \) is lightweight compared to the cost of transferring a data item from \( A \) to \( B \) and firing \( B \). Fusing \( A \) and \( B \) prevents the unnecessary data transfer and operator firing. The fusion removes the pipeline parallelism between \( A \) and \( B \), but since \( B \) is lightweight, the savings outweigh the lost benefits from pipeline parallelism.

5.2. Profitability
Fusion trades communication cost against pipeline parallelism. When two operators are fused, the communication between them is cheaper. But without fusion, in a multithreaded system, they can have pipeline parallelism: the upstream operator already works on the next data item while, simultaneously, the downstream operator is still working on the previous data item. The chart shows throughput given two operators of equal cost. The cost of the operators is normalized to a communication cost of 1 for sending a data item between nonfused operators. When the operators are not fused, there are two cases: if operator cost is lower than communication cost, throughput is bounded by communication cost; otherwise, it is determined by operator cost. When the operators are fused, performance is determined by operator cost alone. The break-even point is when the cost per operator equals the communication cost, because the fused operator is \( 2 \times \) as expensive as each individual operator.

5.3. Safety
Fusion is safe if the following conditions hold:

— Ensure resource kinds. The fused operators must only rely on resources, including logical resources such as local files and physical resources such as GPUs, that are all available on a single host.
Fusion in 1963

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

define $\alpha$ arr $= A \times \text{int} \rightarrow \alpha$

let to_arr : $\alpha$ array $\rightarrow$ $\alpha$ arr $=$ fun a $\rightarrow$
\hspace{1cm} $A$ (Array.length a, Array.get a)
Array Programming with Fusion

type \( \alpha \) arr = A of int * (int \rightarrow \alpha) 

let to_arr : \( \alpha \) array \rightarrow \( \alpha \) arr = fun a \rightarrow
  A (Array.length a, Array.get a)

let map : (\( \alpha \) \rightarrow \( \beta \)) \rightarrow \( \alpha \) arr \rightarrow \( \beta \) arr = fun f (A (n,ix)) \rightarrow
  A(n, ix \triangleright f)

let (\triangleright\ ) f g = fun x \rightarrow f x \triangleright 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

```ocaml
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 \(\alpha\) arr = A of int * (int → \alpha)
let to\_arr : \(\alpha\) array → \(\alpha\) arr = fun a →
A (Array.length a, Array.get a)

let map : (\(\alpha\) → \(\beta\)) → \(\alpha\) arr → \(\beta\) 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 ▶ filter Float.is_finite ▶ map sqr ▶ sum

Arrays with missing elements

\[
\text{type } \alpha \text{ option } = \text{None } | \text{Some of } \alpha \\
\text{type } \alpha \text{ arr } = A \text{ of int } \times (\text{int} \to \alpha \text{ option})
\]

let to_arr : \alpha \text{ array } \to \alpha \text{ arr } = \ldots
Array Programming with Filtering and Fusion

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

\[
\text{type } \alpha \text{ arr } = \text{A of int } * (\text{int} \rightarrow \alpha \text{ option})
\]

\[
\text{let map } : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ arr } \rightarrow \beta \text{ arr } = \text{fun } f (\text{A (n,ix)}) \rightarrow \\
\text{A(n, fun i } \rightarrow \text{match ix i with Some y } \rightarrow \text{Some (f y) } \mid \_ \rightarrow \text{None)}
\]
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 : (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)
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 ▷ filter Float.is_finite ▷ map sqr ▷ sum

Arrays with missing elements, in CPS

\[
\text{type } \alpha \text{ arr } = A \text{ of int } \times (\text{int } \rightarrow (\alpha \rightarrow \text{unit}) \rightarrow \text{unit})
\]
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

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

type \( \alpha \) arr = \( A \) of int * (int → (\( \alpha \) → unit) → unit)

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

let filter : (\( \alpha \) → bool) → \( \alpha \) arr → \( \alpha \) 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 \( \alpha \) arr = A of int * (int → (\( \alpha \) → unit) → unit)

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

let filter : (\( \alpha \) → bool) → \( \alpha \) arr → \( \alpha \) 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
Array Programming with Complete Fusion

Staged Arrays with missing elements

type \( \alpha \) cde = string

type \( \alpha \) arr =
    A of int cde * (int cde \to (\alpha \text{ cde} \to \text{unit cde}) \to \text{unit cde})
Array Programming with Complete Fusion

\[
\text{type } \alpha \text{ arr } = \\
A \text{ of } \text{int cde } * (\text{int cde } \rightarrow (\alpha \text{ cde } \rightarrow \text{unit cde}) \rightarrow \text{unit cde})
\]

let \text{to}_\text{arr} : \alpha \text{ array } \rightarrow \alpha \text{ arr } = \text{fun a } \rightarrow \\
A (\text{Array.length a}, \text{fun i k } \rightarrow \text{Array.get a i } \triangleright k)

Before (unstaged)
Array Programming with Complete Fusion

type \( \alpha \) arr =
    \( A \) of int cde * (int cde \rightarrow (\alpha \ cde \rightarrow \text{unit cde}) \rightarrow \text{unit cde})

let to_arr : \( \alpha \) array cde \rightarrow \( \alpha \) arr = fun a \rightarrow
    \( A \) (sprintf "Array.length %s" a,
    fun i k \rightarrow sprintf "(Array.get %s %s)" a i \triangleright k)

Generate the code to evaluate \text{Array.length} and \text{Array.get} later
type $\alpha$ arr =
    A of int cde * (int cde $\to$ ($\alpha$ cde $\to$ unit cde) $\to$ unit cde)

let to_arr : $\alpha$ array cde $\to$ $\alpha$ arr = fun a $\to$
    A (sprintf ”Array.length %s” a,
        fun i k $\to$ sprintf ”(Array.get %s %s)” a i ▷ k)

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

```ocaml
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)
type $\alpha$ arr =
  A of int cde * (int cde $\rightarrow$ ($\alpha$ cde $\rightarrow$ unit cde) $\rightarrow$ unit cde)

let to_arr : $\alpha$ array cde $\rightarrow$ $\alpha$ arr = fun a $\rightarrow$
  A (sprintf "Array.length %s" a,
      fun i k $\rightarrow$ sprintf "(Array.get %s %s)" a i $\triangleright$ k)

let sum : float arr $\rightarrow$ float cde = fun (A (n,ix)) $\rightarrow$
  sprintf
    "let sum = ref 0. in
     for i = 0 to %s-1 do
       %s done; !sum"
    n
  (ix "i" (fun y $\rightarrow$ sprintf "sum := !sum +. %s" y))
Array Programming with Complete Fusion

type $\alpha$ arr =
  $\alpha$ array cde * (int cde $\rightarrow$ ($\alpha$ cde $\rightarrow$ unit cde) $\rightarrow$ unit cde)

let to_arr : $\alpha$ array cde $\rightarrow$ $\alpha$ arr = fun a $\rightarrow$
  A (sprintf "Array.length %s" a,
      fun i k $\rightarrow$ sprintf "(Array.get %s %s)" a i $\triangleright$ k)

let filter : ($\alpha$ $\rightarrow$bool) $\rightarrow$ $\alpha$ arr $\rightarrow$ $\alpha$ arr = fun f (A (n,ix)) $\rightarrow$
  A(n,fun i k $\rightarrow$ ix i (fun y $\rightarrow$ if f y then k y))

Before (unstaged)
Array Programming with Complete Fusion

\[
\text{type } \alpha \text{ arr } = \\
\quad \text{A of int cde } \to \text{ (int cde } \to \text{ (} \alpha \text{ cde } \to \text{ unit cde) } \to \text{ unit cde)}
\]

let to_arr : \(\alpha\) array cde \to \(\alpha\) arr = fun a \to \\
\quad \text{A (sprintf "Array.length %s" a,} \\
\quad \quad \text{fun i k \to sprintf "(Array.get %s %s)" a i } \triangleright k)

let filter : \((\alpha\) cde \to \text{bool cde}) \to \(\alpha\) arr \to \(\alpha\) arr = fun f (A (n,ix)) \to \\
\quad \text{A(n,fun i k \to ix i (fun y \to sprintf "if %s then %s" (f y) (k y)))}
Array Programming with Complete Fusion

type $\alpha$ arr =
   A of int cde * (int cde $\rightarrow$ ($\alpha$ cde $\rightarrow$ unit cde) $\rightarrow$ unit cde)

let to_arr : $\alpha$ array cde $\rightarrow$ $\alpha$ arr $=$ fun a $\rightarrow$
   A (sprintf "Array.length %s" a,
       fun i k $\rightarrow$ sprintf "(Array.get %s %s)" a i $\triangleright$ k)

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

let app : ($\alpha$ $\rightarrow$ $\beta$) cde $\rightarrow$ $\alpha$ cde $\rightarrow$ $\beta$ cde $=$ fun f x $\rightarrow$
   sprintf "(%s %s)" f x
to_arr a ▷ filter Float.is_finite ▷ map sqr ▷ sum

Before (unstaged)
let is\_finite = app "\ Float\_is\_finite"
let sqr = app "sqr"

let v2 = to\_arr "a" ▷ filter is\_finite ▷ map sqr ▷ sum
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

Introduction: What is Stream Processing

Stream Fusion

► Strymonas

Case Study: FM Radio
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

```ocaml
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

\[ T_1: \text{string} \times \text{int} \] table, \[ T_2: \text{int} \times \text{float} \] table

\[
\text{select } T_1.1, 2 \times T_2.2 \text{ from } T_1, T_2 \text{ where } T_1.2 = T_2.1 \text{ and } T_2.2 > 5.0
\]

\[
\text{let cart } (s1,s2) = \text{flat_map (fun e1 \to s2 \text{ where } Raw.map_raw' (fun e2 \to (e1,e2)))}
\]

\[
\text{let join } (t1,t2) = \text{cart (of_arr t1, of_arr t2)}
\]

\[
(* \text{ WHERE clauses } *)
\]
\[
\text{Raw.filter_raw C.(fun (e1,e2) \to snd e1 = fst e2)}
\]
\[
\text{Raw.filter_raw C.(fun (e1,e2) \to truncate (snd e2) > int 5)}
\]

\[
(* \text{ SELECTion } *)
\]
\[
\text{Raw.map_raw' C.(fun (e1,e2) \to pair (fst e1) (snd e2 \times \text{float 2.})}
\]

\[
(* \text{ Output } *)
\]
\[
\text{iter (fun (e1,e2) \to seq (print e1) (print_float e2))}
\]
A weird test

```plaintext
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

```ocaml
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)
  done
```

Stateful Streams

Difference encoder

\[
\text{let } \text{diff} : \text{int cstream } \rightarrow \text{int cstream} = \text{fun} \ st \rightarrow \\
\text{initializing}_\text{ref} \ C.(\text{int } 0) @@ \text{fun} \ z \rightarrow \\
\text{map} \ C.\left(\text{fun} \ e \rightarrow \text{letl} \ (e \rightarrow \text{dref} \ z) @@ \text{fun} \ v \rightarrow \text{seq} \ (z := e) \ v\right) \ st
\]

\[
\text{take}_\text{while}
\]

\[
\text{let } \text{take}_\text{while} : (\alpha \ cde \rightarrow \text{bool} \ cde) \rightarrow \alpha \ \text{cstream} \rightarrow \alpha \ \text{cstream} = \text{fun} \ f \ st \rightarrow \\
\text{initializing}_\text{ref} \ C.(\text{bool } \text{true}) @@ \text{fun} \ zr \rightarrow \\
st \triangleright \text{map}_\text{raw} \ C.\left(\text{fun} \ e \ k \rightarrow \text{if} \ (f \ e) \ (k \ e) \ (zr := \text{bool} \ \text{false})\right) \triangleright \text{guard} \ C.(\text{dref} \ zr)
\]
Results: JVM

![Graph showing performance results for different operations and techniques in JVM]

- **Operations**:
  - sum
  - sumOfSquares
  - sumOfSquaresEven
  - cart
  - dotProduct
  - filters
  - maps
  - flatMap_after_zipWith
  - zipWith_after_flatMap
  - flat_map_take

- **Techniques**:
  - baseline_java
  - baseline_scala
  - streams_java
  - staged_scala

The graph compares the average execution time (ms/op) for each operation using different techniques.
Results: C
Outline

Introduction: What is Stream Processing

Stream Fusion

Strymonas

▶ Case Study: FM Radio
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

```ocaml
let lowPassFilter : float → float → int → int → float cstream → float cstream =
  fun rate cutoff taps decimation st →
  let mk_coef_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_coef_arr cutoff) C.(+.)(.*.win)))
```
Conclusions

▶ Stream processing is varied: EE, CS, MBA,…
▶ Stream fusion is important and nontrivial especially complete stream fusion
▶ Strymonas can do it
Joint work with
Aggelos Biboudis, Tomoaki Kobayashi, Nick Palladinos, and
Yannis Smaragdakis