ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

A Brief History of Streams

Aggelos Biboudis
Monday October 22, 2018
NII Shonan Meeting Seminar 136 on Functional Stream Libraries and Fusion

The goal of this talk

- A brief (or 10km-overview) of Streams
- Review the major areas
- Introduce the main terminology
- If something is overlooked, please mention
- Let’s discuss challenges (interactive)

Processing stream of tweets

```
tweets
        ▷ filter(t => t.contains("#phdlife"))
        ▷ filter(t => Sentiment.detectSentiment(t) == POSITIVE)
        ▷ map(t => t.User)
        ▷ take 15
        ▷ any(u => u.Followers > 1000)
```

Basics of a Streaming API

```
type α stream

Producers
val of_arr    : α array                          → α stream
val unfold    : (ζ → (α * ζ) option) → ζ → α stream

Transformers
val map       : (α → β)          → α stream → β stream
val filter    : (α → bool)       → α stream → α stream
val take      : int              → α stream → α stream
val flat_map  : (α → β stream) → α stream → β stream
val zip_with  : (α → β → γ) → (α stream → β stream → γ stream)

Consumer
val fold      : (ζ → α → ζ) → ζ → α stream → ζ
```

Stream Origins

- Melvin Conway, 1963: Coroutines
  "*separable programs*"

- Douglas McIlroy, 1964: Unix Pipes
  pipe() implemented by Ken Thompson in v3, 1973
  '|' leads to a "pipeline revolution" in v4

- Peter Landin, 1965: Streams
  "functional analogue of coroutines"

Coroutines: Conway, 1963

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. 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.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective COBOL except for automatic segmentation and its byproducts, such as those properties of the ALTER verb which are affected by segmentation. The verbs DEFINE, ENTER, USE and INCLUDE are accessible

to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

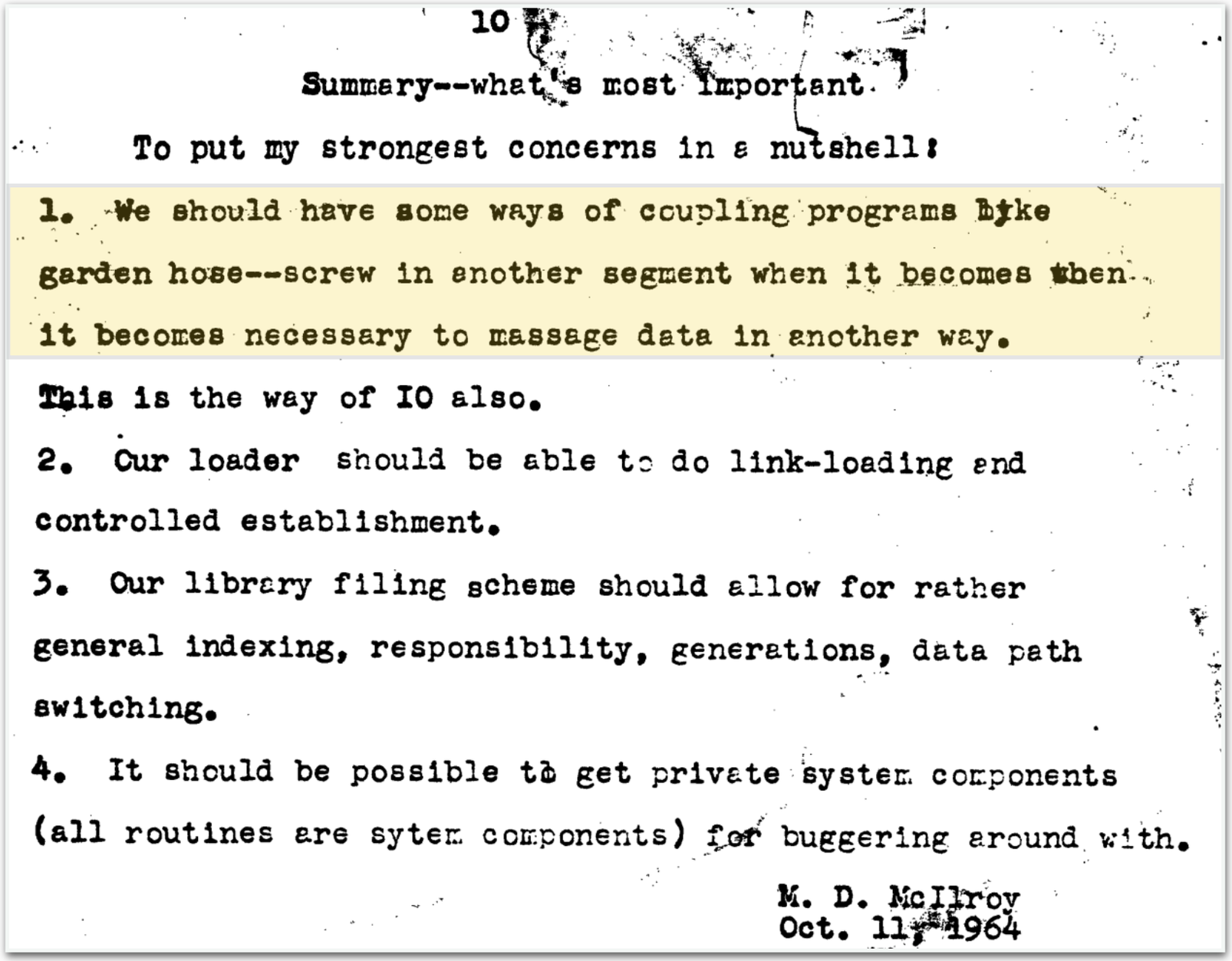
The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

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.

Unix Pipes: D. McIlroy, 1964

(implemented in 1973 by Ken Thomson more info at http://www.softpanorama.org/Scripting/Piporama/history.shtml)



10

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes then it becomes necessary to massage data in another way. This is the way of IO also.

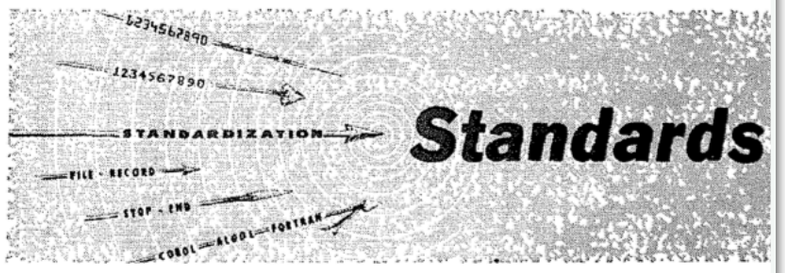
2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are syten components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

Streams: Landin, 1965



S. GORN, I

A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I*

By P. J. Landin†

This paper describes how some of the semantics of ALGOL 60 can be formalized by establishing a correspondence between expressions of ALGOL 60 and expressions in a modified form of Church's λ-notation. First a model for computer languages and computer behavior is described, based on the notions of functional application and functional abstraction, but also having analogues for imperative language features. Then this model is used as an "abstract object language" into which ALGOL 60 is mapped. Many of ALGOL 60's features emerge as particular arrangements of a small number of structural rules, suggesting new classifications and generalizations.

The correspondence is first described informally, mainly by illustrations. The second part of the paper gives a formal description, i.e. an "abstract compiler" into the "abstract object language." This is itself presented in a "purely functional" notation, that is one using only application and abstraction.

for $v :=$ a **step** b **until** c,	$for(v,$
d,	$\quad concatenate\ (step(a, b, c),$
e **while** p	$\quad\quad unitlist\ (d),$
do T	$\quad\quad while(e, p)),$
	$\quad T)$

where *for*, *concatenate*, *step* and *while* are defined as follows.[5]

rec $for(v, S, T)$ $=$ **if** $\neg$ $null\ S$ **then** $[v := hS;$
$T;$
$for(v, tS, T)]$

rec $concatenate\ S$ $=$ $null\ S \rightarrow (\ )$
$null(hS) \rightarrow concatenate\ (tS)$
else $\rightarrow h^2S : concatenate(t(hS) : tS)$

rec $step(a, b, c)$ $=$ $(a - c) \times sign(b) > 0 \rightarrow (\ )$
else $\rightarrow a : step(a+b, b, c)$

rec $while(e, p)$ $=$ $p \rightarrow e : while(e, p)$
else $\rightarrow (\ )$

However, these definitions fail to reflect the sequence of execution prescribed for ALGOL 60. When interpreted by the sharing machine they would lead to an attempt to evaluate the entire control-list before the first iteration of the loop. The inadequacy of this approach is especially flagrant in the case of *while*. We therefore consider **for**-list-elements as denoting not lists but a particular kind of function, called here a *stream*, that is like a list but has special properties related to the sequencing of evaluation. Principally, the items of an intermediately resulting stream need never exist simultaneously. So streams might have practical advantages when a list is subjected to a cascade of editing processes.[6]

---

[5] Following [MEE], an infixed colon indicates prefixing. Thus "$x:L$" is equivalent to "*prefix x L*."

[6] It appears that in stream-transformers we have a functional analogue of what Conway [12] calls "co-routines."

Fast-Forward 52 years

- iterators (‘`yield`’), generators as in Python, …
- LINQ, Java 8 Streams, …
- Lucid, LUSTRE, …
- Naiad, Flink, DryadLINQ, Spark Streaming, …
- Rx, Elm, …
- SIMD, …
- StreamIt, …
- Ziria, …

(Gilles) Kahn networks (1974)

- Infinite streams of data processed
- Kahn Process: a sequential program reading / writing to FIFOs channels
- Unbounded channels
- Determinism
- Monotonicity

Semantics of Kahn Process Networks

- Operational: a transition system
- Denotational: each process is a function on streams

    1. feedback loops correspond to fix points

Synchronous Data Flow (1987)

Synchronous Data Flow

EDWARD A. LEE, MEMBER, IEEE, AND DAVID G. MESSERSCHMITT, FELLOW, IEEE

Data flow is a natural paradigm for describing DSP applications for concurrent implementation on parallel hardware. Data flow programs for signal processing are directed graphs where each node represents a function and each arc represents a signal path. Synchronous data flow (SDF) *is a special case of data flow (either atomic or large grain) in which the number of data samples produced or consumed by each node on each invocation is specified* a priori. *Nodes can be scheduled statically (at compile time) onto single or parallel programmable processors so the run-time overhead usually associated with data flow evaporates. Multiple sample rates within the same system are easily and naturally handled. Conditions for correctness of SDF graph are explained and scheduling algorithms are described for homogeneous parallel processors sharing memory. A preliminary SDF software system for automatically generating assembly language code for DSP microcomputers is described. Two new efficiency techniques are introduced,* static buffering *and an extension to SDF to efficiently implement conditionals.*

I. Data Flow and Synchronous Data Flow: An Introduction

For concurrent implementation, a signal processing task is broken into subtasks which are then automatically, semi-automatically, or manually scheduled onto parallel processors, either at compile time (*statically*) or at run-time (*dynamically*). Automatic breakdown of an ordinary sequential computer program is an appealing concept [1], but the success of techniques based on traditional imperative programming is limited; imperative programs do not often exhibit the concurrency available in the algorithm. If the programmer provides the breakdown as a natural consequence of the programming methodology, we should expect more efficient use of concurrent resources.

Synchronous data flow (*SDF*) is a special case of data flow [2]–[6], a hardware and software methodology popular among computer scientists for parallel computation. Under the data flow paradigm, algorithms are described as directed graphs where the nodes represent computations (or functions) and the arcs represent data paths. A second-order recursive digital filter described as a data flow graph is shown in Fig. 1. In that example, an essentially infinite

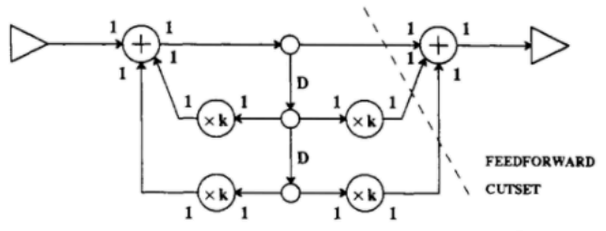


Fig. 1. A data flow graph for a second-order recursive digital filter. The empty circles are "fork" nodes, which simply replicate each input sample on all output paths. The "*D*" on two of the arcs indicates delay, and the "1" adjacent to each node input or output indicates that a single token is consumed or produced when the node fires.

stream of input data is expected, so the nodes specify computations performed infinitely often. This is typical of signal processing applications, and is an important property often lacking in more general applications.

The data flow principle is that any node can *fire* (perform its computation) whenever input data are available on its incoming arcs. A node with no input arcs may fire at any time. This implies that many nodes may fire simultaneously, hence the concurrency. Because the program execution is controlled by the availability of data, data flow programs are said to be *data-driven* [7]. To preserve the integrity of the computation, nodes must be free of *side effects*. For example, a node may not write to a memory location which is later read by another node unless the two are explicitly connected by an arc. The only influence one node has on another is the data passing through the arcs.

In Fig. 1, each input and output of each node has the number "1" adjacent to it, which in our notation indicates that when the node fires, a single sample (or *token*) will be consumed or produced on each arc. A synchronous data flow graph is one for which these numbers may be specified for every node *a priori*. That is, the number of tokens produced or consumed must be independent of the data. We expect that most nodes for signal processing applications will be

Stream Processing Functions (Burge, 1975)

Stream Processing Functions

Abstract: One principle of structured programming is that a program sh
which are then combined so that the relation of the parts to the whole can b
ways to compose programs. The main method used is to permit the prograr
on by a variable. The sequence is represented by a function called a *stre*
tional **while** and **for** loops of structured programming may be composed by
ing), which results in more structured programs than the originals. This te
way into its logically separate parts, which can then be considered independ

Introduction

One of the underlying principles of structured programming [1] is that the separation of the parts of a program, and the relation of the parts to the whole, should both be clearly apparent from its written form. A second principle is that the meaning of each part should depend in a simple way only on the meaning of its subparts, and not on any other properties. Programs written in this way are easy to understand and write, and the details of their operation are transparently clear. This principle of structured programming is epitomized in the expression for-

use
rat
for
be
sep
dep
of
dat
mir
tw

The data structure that is relevant is an *A-sequence*, defined as follows:

An *A-sequence* has a *hs*, which is an *A*,

and a *ts* which is an *A-sequence*.

Streams A sequence is therefore an infinite list, and the problem of conserving storage for its representation inside a computer becomes even more pressing. A sequence can be represented by a particular type of function, which is called a *stream function* or a *stream*. A *stream* is applicable to an empty list of arguments, and it produces a pair whose first is the next item in the sequence and whose second is a stream for the tail of the sequence. Thus

$$A\text{-}stream \subseteq (null\ list \rightarrow A \times A\text{-}stream)$$

Lists/Generators/Lazy Lists

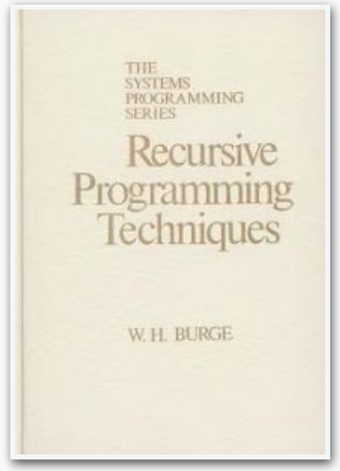
Principia Mathematica (Whitehead and Russell,**1910-1913**)

→ IPL (lists, generators, and more by Newell, Shaw and Simon, **1957**) /
Programming the logic theory machine (Newell and Shaw, **1962**)

→ Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I (McCarthy, **1960**)

→ Laziness appears with a quadruple attack:

The "*Procrastinating [SECD] Machine"* by (Burge, **1975**) ←
SASL's semantics get rewritten adopting laziness (Turner, **1975**)
CONS should not evaluate its arguments (Friedman et al., **1976**)
A Lazy Evaluator (Henderson et al., **1977**)



→ Iterators appear with a double attack:

Abstraction mechanisms in CLU (Liskov et al, CACM vol **20**, **1977**)
Abstraction and Verification in Alphard (M Shaw et al, CACM vol **20**, **1977**)

→The Semantic Elegance of Applicative Languages (Turner, **1981**)

Turner On Lazy Lists vs Coroutines

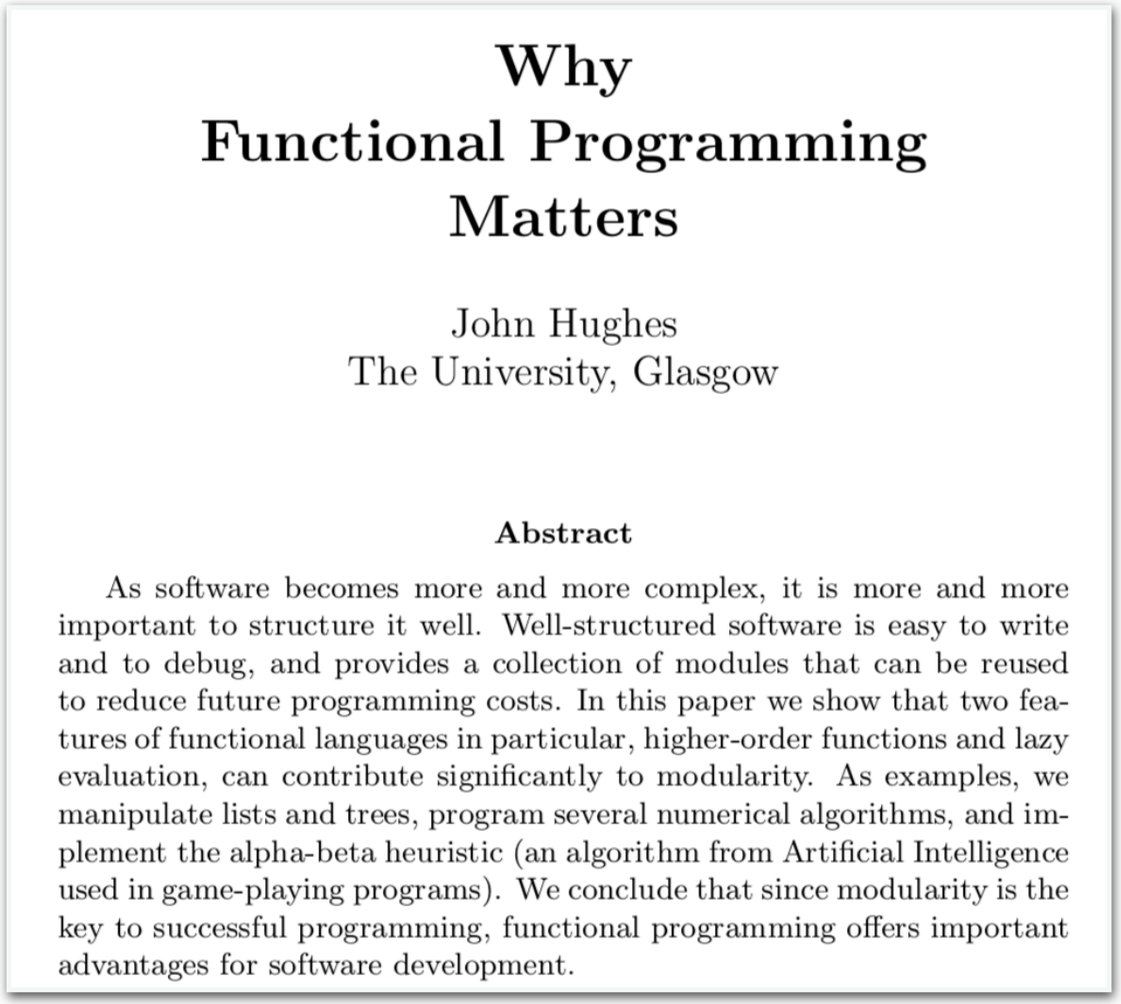
6.2 Advantages of Laziness

Two other projects independently developed lazy functional programming systems in the same year as SASL — Friedman & Wise (1976), Henderson & Morris (1976). Clearly laziness was an idea whose time had arrived.

My motives in changing to a lazy semantics in 1976 were

(i) on a sequential machine, consistency with the theory of (Church 1941) requires normal order reduction
(ii) a non-strict semantics is better for equational reasoning
(iii) allows interactive I/O via lazy lists — call-by-value SASL was limited to outputs that could be held in memory before printing.
(iv) I was coming round to the view that lazy data structures could replace exotic control structures, like those of J-PAL.
  (a) lazy lists replace coroutines (e.g. equal fringe problem)
  (b) the *list of successes*[12] method replaces backtracking

Lazy Lists



Why
Functional Programming
Matters

John Hughes
The University, Glasgow

Abstract

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write and to debug, and provides a collection of modules that can be reused to reduce future programming costs. In this paper we show that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute significantly to modularity. As examples, we manipulate lists and trees, program several numerical algorithms, and implement the alpha-beta heuristic (an algorithm from Artificial Intelligence used in game-playing programs). We conclude that since modularity is the key to successful programming, functional programming offers important advantages for software development.

```
data List a = Nil
            | Cons a (List a)
```

consumer $g \circ f$ producer

Lazy Lists

(in strict languages / OCaml)

```
(* lists in OCaml *)
type 'a list =
 | Nil
 | Cons of 'a * 'a list


(* infinite lists in OCaml with thunks *)
type 'a lazy_list =
 | Nil
 | Cons of 'a * (unit → 'a lazy_list)


(* list shape / step *)
type ('a,'z) list_shape =
 | Nil
 | Cons of 'a * 'z
```

Lazy Lists

(in strict languages / OCaml)

```
(*  compact form of a stream *)
type 'a stream = Cons of 'a * (unit → ('a, 'a stream) list_shape)


(*  compact form of stream with closure conversion to
    explicitly pass the state to the stepper function *)
type 'a stream = { stepper : 's * ('s → ('a,'s) list_shape) → 'a stream }
```

What happens today on the other side?
(strict languages with yield)

```
// LINQ query (C#)
from p in Enumerable.Range(0, int.MaxValue)
where p % 2 == 0
select p


// desugared
Enumerable.Range(0, int.MaxValue)
          .Where<int>((Func<int, bool>) (p => p % 2 == 0))

// implementation of operators
IEnumerable<TResult> SelectIterator<TSource, TResult>(
   IEnumerable<TSource> source,
   Func<TSource, int, TResult> selector) {
            int index = -1;
            foreach (TSource element in source) {
                checked { index++; }
                yield return selector(element, index);
            }
        }
```

IEnumerable / IEnumerator

Two styles of composition

(or “who controls my stack”?)

```
Pull<T> source(T[] arr) {
  return new Pull<T>() {
    boolean hasNext() {..}
    T next() {..}
  };
}



Pull<Integer> sIt =
  source(v).map(i->i*i);

while (sIt.hasNext()) {
  el = sIt.next();
  /* consume el */
}
```

(Scala/C#/F#)

```
Push<T> source(T[] arr) {
  return k -> {
   for (int i = 0;
         i < arr.length; i++)
     k(arr[i]); };
}



Push<Integer> sFn =
  source(v).map(i->i*i);

sFn(el -> /* consume el */);
```

(Java 8 Streams)

Two styles of composition

(F-algebra vs F-co algebra)

- Push-based design
  - inspired by folds
  - producer-driven
  - better inlining ✓
  - map, filter ✓
  - flat_map (laziness takes a hit combined with take)
  - no trivial take
  - no trivial zip
- Pull-based design
  - inspired by unfolds or generators (CLU/Alphard)
  - consumer-driven
  - map, filter ✓
  - flat_map ✓
  - infinite streams ✓
  - zip ✓
  - short-circuit (take) ✓

And, pull/push perspectives

(on hotspot-compiler-dev mailing list)

perspectives on streams performance

John Rose john.r.rose at oracle.com
Fri Mar 6 01:01:20 UTC 2015

- Previous message: RFR(S) 8074010: followup to 8072383
- Next message: perspectives on streams performance
- **Messages sorted by:** [date] [thread] [subject] [author]

```
In order to get the full benefit from JDK 8 streams we will need to make th

I think of streams as a more concise and orderly replacement of classic "fo

A classic "for" loop is a external iterator notation:  The iteration machin

External iterators are easier to optimize, because their crucial iteration

HotSpot are less good at internal iterators.  If the original point of the
```

(pull)

(push)

Lowering the Abstraction

- **excessive** construction and deconstruction of values (e.g., composing and decomposing in pattern matching);

- recursive calls when an environment does **not support specific optimizations** (e.g., tail-call optimization);

- **heap-allocated closures** (e.g., lambdas capturing free variables) when the program is higher-order;

- **multiple iterations over one sequence of data** that can be traversed once (e.g., a map of squares pipeline over a stream);

- **iterations over multiple sequences of data** that can be traversed at once (e.g., zipping a stream); and

- **highly polymorphic call-sites** (e.g., MoveNext() in iterators), also known as “megamorphic”, which incur dynamic dispatch (“virtual method”) overhead.

Lowering the Abstraction

A Transformation System for Developing Recursive Programs

R. M. BURSTALL AND JOHN DARLINGTON

University of Edinburgh, Edinburgh, Scotland

ABSTRACT A system of rules for transforming programs is described, with the programs in the form of recursion equations An initially very simple, lucid, and hopefully correct program is transformed into a more efficient one by altering the recursion structure Illustrative examples of program transformations are given, and a tentative implementation is described Alternative structures for programs are shown, and a possible initial phase for an automatic or semiautomatic program manipulation system is indicated

KEY WORDS AND PHRASES program transformation, program manipulation, optimization, recursion

CR CATEGORIES· 3 69, 4 12, 4 22, 5 24, 5 25

1. *Introduction*

We present here a system for transforming programs, where the programs are expressed as first order recursion equations. This recursive form seems well adapted to manipulation, much more so than the usual Algol-style form of program, and our transformation system consists of just a few simple rules together with a strategy for applying them. Despite their simplicity, these rules produce some interesting changes in the programs.

The overall aim of our investigation has been to help people to write correct programs which are easy to alter. To produce such programs it seems advisable to adopt a lucid, mathematical, and abstract programming style. If one takes this really seriously, attempting to free one's mind from considerations of computational efficiency, there may be a heavy penalty in program running time; in practice it is often necessary to adopt a more intricate version of the program, sacrificing comprehensibility for speed. The question then arises as to how a lucid program can be transformed into a more intricate but efficient one in a systematic way, or indeed in a way which could be mechanized.

(Stream) Fusion

Burstall R. M. and Darlington J.

→ Listlessness is Better Than Laziness (Wadler, 1984)

→ Shortcut fusion (foldr/build, Gill et al., 1993)

→ Shortcut Fusion for Accumulating Parameters & Zip-like Functions (unbuild/unfoldr, Svenningsson, 2002)

→ Coutts et al., Stream Fusion

(Stream) Fusion

```
data Stream a = ∃s. Stream (s → Step a s) s
data Step a s = Done
              | Yield a s
              | Skip s
```

(note the resemblance to) The Under-Appreciated Unfold

Jeremy Gibbons
School of Computing and Math. Sciences
Oxford Brookes University
Gipsy Lane, Headington,
Oxford OX3 0BP, UK.
Email: jgibbons@brookes.ac.uk

Geraint Jones
Oxford University Computing Lab
Wolfson Building, Parks Road
Oxford OX1 3QD, UK.
Email: geraint@comlab.ox.ac.uk

(Stream) Fusion

Burstall R. M. and Darlington J.

→ Listlessness is Better Than Laziness (Wadler, 1984)

→ Shortcut fusion (foldr/build, Gill et al., 1993)

→ Shortcut Fusion for Accumulating Parameters & Zip-like Functions (unbuild/unfoldr, Svenningsson, 2002)

→ Coutts et al., Stream Fusion

Staged Stream Fusion

```
of_arr .⟨arr⟩.
  ▷ map (fun x → .⟨~x * ~x⟩.)
  ▷ sum
```

staging ↓

```
let s_1 = ref 0 in
let arr_2 = arr in
 for i_3 = 0 to Array.length arr_2 -1 do
   let el_4 = arr_2.(i_3) in
   let t_5 = el_4 * el_4 in
   s_1 := t_5 +  !s_1
 done;
!s_1
```

Staged Stream Fusion

and much more complex...

```
zip_with (fun e1 e2 → .⟨(~e1,~e2)⟩.)
 (of_arr .⟨arr1⟩. (* 1st stream *)
   ▷ map (fun x → .⟨~x * ~x⟩.)
   ▷ take .⟨12⟩.
   ▷ filter (fun x → .⟨~x mod 2 = 0⟩.)
   ▷ map (fun x → .⟨~x * ~x⟩.))
(iota .⟨1⟩.        (* 2nd stream *)
   ▷ flat_map (fun x → iota .⟨~x+ 1⟩. ▷ take .⟨3⟩.)
   ▷ filter (fun x → .⟨~x mod 2 = 0⟩.))
▷  fold (fun z a → .⟨~a :: ~z⟩.) .⟨[]⟩.
```

Multi-Stage Programming

- manipulate code templates
- brackets to create well-{formed, scoped, typed} templates

```
let c = .< 1 + 2 >.
```

- create holes

```
let cf x = .< .~x + .~x >.
```

- synthesize code at staging-time (runtime)

```
cf c ~> .< (1 + 2) + (1 + 2) >.
```

Naive Staging

based on `unfoldr`:
functional analogue of iterators

```
type α stream = ∃σ. σ * (σ → (α,σ) stream_shape)
```

```
type ('a,'z) stream_shape =
    | Nil
    | Cons of 'a * 'z
```

Naive Staging

binding-time analysis

```
type α stream = ∃σ. σ code * (σ code → (α,σ) stream_shape code )
```

classify variables as static and dynamic

Naive Staging

```
let map : ('a code -> 'b code) -> 'a stream -> 'b stream =
  fun f (s, step) ->
    let new_step = fun s ->
      .< match .~(step s) with
          | Nil          -> Nil
          | Cons (a,t) -> Cons (.~(f .<a>.), t)>.
    in (s, new_step);;
```

Result

no intermediate ✓
function inlining ✓
various overheads ✗

```
let rec loop_1 z_2 s_3 =
 match match match s_3 with
         | (i_4, arr_5) ->
           if i_4 < (Array.length arr_5)
           then Cons ((arr_5.(i_4)),((i_4 + 1), arr_5))
           else Nil
   with
   | Nil  -> Nil
   | Cons (a_6,t_7) -> Cons ((a_6 * a_6), t_7)
 with
 | Nil  -> z_2
 | Cons (a_8,t_9) -> loop_1 (z_2 + a_8) t_9
```

✗ → match match match s_3 with

✗ → with

✗ → with

of_arr

map

sum

Step 1: fusing the stepper

- stepper has known structure though!

```
let map : ('a code -> 'b code) -> 'a st_stream -> 'b st_stream =
  fun f (s, step) ->
    let new_step s k = step s @@ function
    | Nil          -> k Nil
    | Cons (a,t) -> .<let a' = .~(f a) in .~(k @@ Cons (.<a'>., t))>.
    in (s, new_step)
;;
```

stream_shape is static and factored out of the dynamic code

* Anders Bondorf. 1992. Improving binding times without explicit CPS-conversion. In LFP '92
* Oleg Kiselyov, Why a program in CPS specializes better, http://okmij.org/ftp/meta-programming/#bti

Result
(after step 1)

stepper inlined ✓
pattern matching ✗

✗

```
let rec loop_1 z_2 s_3 =
  match s_3 with
    | (i_4, arr_5) ->
      if i_4 < (Array.length arr_5)
      then
        let el_6 = arr_5.(i_4) in
        let a'_7 = el_6 * el_6 in
        loop_1 (z_2 + a'_7) ((i_4 + 1), arr_5)
      else z_2
```

Step 2: fusing the state

- no pair-allocation in loop: state passed in and mutated

```
let of_arr : 'a array code -> 'a st_stream =
  let init arr k
    = .< let i = ref 0 and
         arr = .~arr in .~(k (.<i>.,.<arr>.))>.
and step (i,arr) k
  = .< if !(.~i) < Array.length .~arr
       then
         let el = (.~arr).(!(.~i)) in
         incr .~i;
         .~(k @@ Cons (.<el>., ()))
       else .~(k Nil)>.
in
fun arr -> (init arr,step)
```

(int * α array) code
~> int ref code * α array code

Result

no pattern matching ✓
recursion ✗

✗ →
```
let i_8 = ref 0
and arr_9 = [|0;1;2;3;4|] in
let rec loop_10 z_11 =
 if ! i_8 < Array.length arr_9
 then
   let el_12 = arr_9.(! i_8) in
   incr i_8;
   let a'_13 = el_12 * el_12 in
✗ →  loop_10 (z_11+a'_13)
 else z_11
```

Factor out static knowledge:
After 3 key domain-specific optimizations*

1. The structure of the stepper is known:
   use that at staging time!

2. The structure of the state is known:
   use that at staging time, too!

3. Tail recursion vs Iteration:
   modularize the loop structure (`for` vs `while`)

* 6 domain-specific optimizations in total, accommodating linearity (filter and flat_map), sub-ranging, infinite streams (take and unfold), and parallel stream fusion (zip)

Step 3: generating imperative loops

```
let of_arr : 'a array code -> 'a stream = fun arr ->
  let init k
      = .<let arr = .~arr in .~(k .<arr>.)>.
  and upper_bound arr
      = .<Array.length .~arr - 1>.
  and index arr i k
      = .<let el = (.~arr).(.~i) in .~(k .<el>.)>.
  in (init, For {upb;index})
```

start with For-form and if needed transform to Unfold

Result

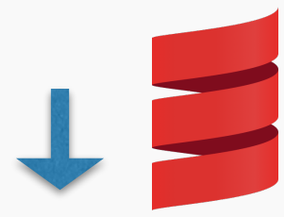
loop-based/fused ✔

```
let s_1 = ref 0 in
let arr_2 = [|0;1;2;3;4|] in
for i_3 = 0 to (Array.length arr_2) - 1 do
  let el_4 = arr_2.(i_3) in
  let t_5 = el_4 * el_4 in s_1 := !s_1 + t_5
done;
!s_1
```

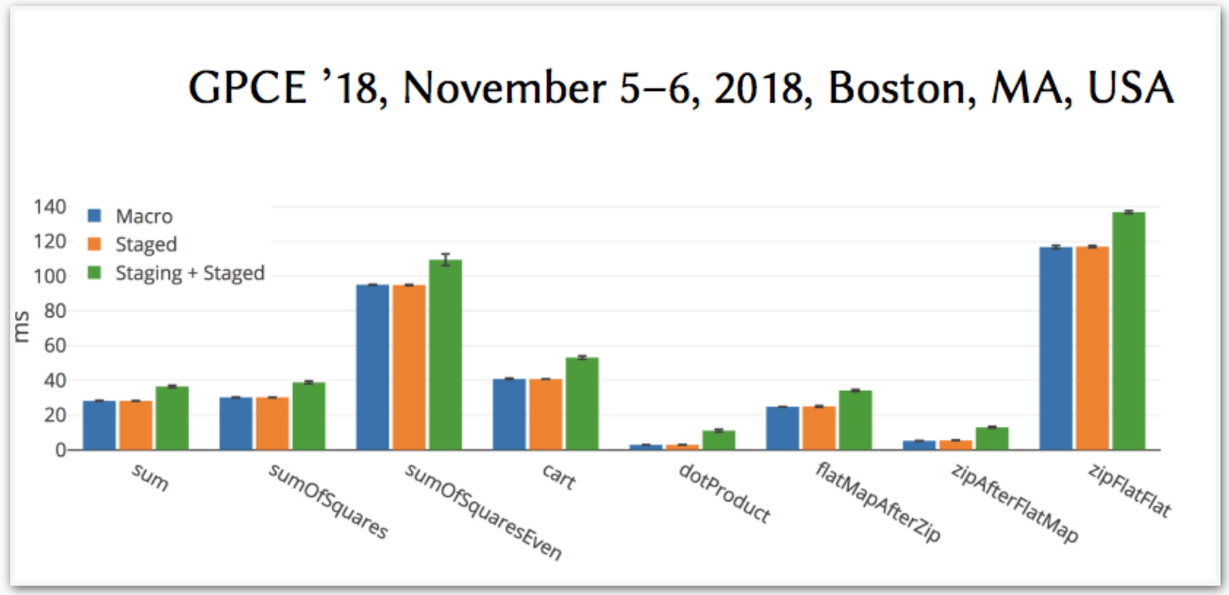
Now in Dotty (Scala 3) too

```
def sum() = '{ (array: Array[Int]) =>
   ~Stream.of('(array)) .fold('{0}, ((a, b) => '{ ~a + ~b }))
 }
```

staging or macros

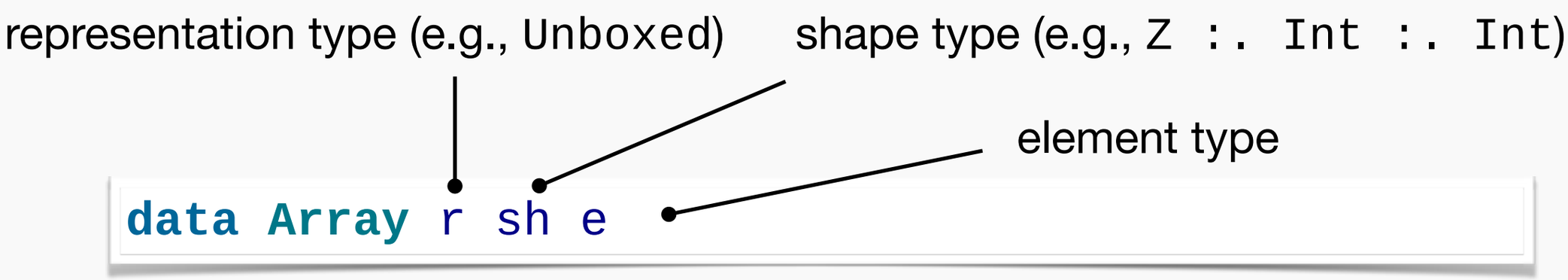


```
({(array: scala.Array[scala.Int]) => {
  var x: scala.Int = 0
  var x$2: scala.Int = array.length
  var x$3: scala.Int = 0
  while (x$3.<(x$2)) {
    val el: scala.Int = array.apply(x$3)
    x$3 = x$3.+(1)
    x = x.+(el)
  }
  x
})
```



Data parallelism over large arrays

(with Repa)



- Delayed representation for fusion (fusion is merely a nested function composition)

- Operations like computeP and foldP parallelize the computations automatically

Data parallelism over large arrays

(with Accelerate)

a function from `Exp Int -> Exp Int`

`Exp` vs `Acc`

we are not in the Haskell world anymore,
`use` represents arrays in the Accelerate world
copying may occur here ;-)

```
> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
> run $ A.map (+1) (use arr)
Array (Z :. 3 :. 5) [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

- GPU backends
- Delayed representation for fusion again, but hidden
- Arrays cannot be nested

NESL (1995)

- NESL Blelloch, Guy (1995). "NESL: A Nested Data-Parallel Language" well suited for irregular algorithms (trees, graphs or sparse matrices)
- A combination of SETL and ML
- Nested-ness eliminated in maps by a flattening transformation (vectorization using *segment-descriptors*)
([[10], [20, 30], [40, 50, 60]] -> ([1, 2, 3], [10, 20, 30, 40, 50, 60]))

```
// apply-to-each
{a * a : a in [3, -4, -9, 5] | a > 0};

// replication of 1
{a * 1 : a in [3, -4, -9, 5] | a > 0};

// nested parallelism
{sum(a) : a in [[2,3], [8,3,9], [7]]};
```

Influences

- Data Parallel Haskell (incorporating higher-order functions)

```
— rewrite rules like
mapᵛ f x → fᵛ x

(fᵛ)ᵛ x → segmentᵛ x (fᵛ (concatᵛ x))
```

- Nessie: A NESL to CUDA Compiler (Reppy & Sandler, 2015)

- Streaming NESL, Madsen & Filinksi, 2016 (dealing with space consumption by incorporating chunking)

Modularity in Array Processing

- Futhark: a stand-alone language for array programming
  https://futhark-lang.org

- LIFT: a functional intermediate representation based on lambda calculus
  http://www.lift-project.org/

Lift, Gist

```
dot(x, y) = reduce (+, 0, map(×, zip(x, y)))
```

```
partialDot (x: [float]N , y: [float]N ) =
(join ◦ mapWrg0(
   join ◦ toGlobal(mapLcl0(mapSeq(id)))
        ◦ split1
        ◦ iterate6(join
           ◦ mapLcl0(toLocal(mapSeq(id))
                     ◦ reduceSeq (add , 0))
                     ◦ split2)
        ◦ join ◦ mapLcl0(toLocal(mapSeq(id))
                         ◦ reduceSeq (multAndSumUp, 0))
               ◦ split2) ◦ split128)(zip(x, y))
```

Futhark, Gist

- Futhark is based on Second Order Array Combinators (R. S. Bird, Algebraic Identities for Program Calculation, 1989) and supports:
- nested parallelism
- in-place array updates
- fusion through rewrite rules

```
fun main (matrix : [n][m]f32): ([n][m]f32, [n]f32) =
  map (λrow : ([m]f32, f32) →
        let row' = map (λx : f32 → x+1.0) row
        let s = reduce (+) 0 row
        in (row',s))
  matrix
```

Query Engine Optimizations

- Many ideas are shared
- (physical) operators consume data from tables
- produce streams of tuples
- cost analysis to compute efficient scans

Query Engine Optimizations

- Volcano model (Graefe, 1994), pull
- DataPath (Arumugam, 2010), push
- HyPer model (Neumann, 2011), code generation, why the shift?

1. next is called for every tuple.
2. each call raises a performance hit (one virtual call per element followed by branch prediction degradation)
3. this model promotes poor code locality

Push versus pull-based loop fusion in query engines

Push versus pull-based loop fusion in query engines

AMIR SHAIKHHA, MOHAMMAD DASHTI
and CHRISTOPH KOCH

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
(*e-mails:* amir.shaikhha@epfl.ch, mohammad.dashti@epfl.ch, christoph.koch@epfl.ch)

Abstract

Database query engines use pull-based or push-based approaches to avoid the materialization of data across query operators. In this paper, we study these two types of query engines in

Haskell libraries for lazy-IO

- https://hackage.haskell.org/package/iteratee
- https://hackage.haskell.org/package/streaming
- https://hackage.haskell.org/package/pipes
- https://hackage.haskell.org/package/conduit

e.g. pipes

(https://hackage.haskell.org/package/pipes-4.3.9/docs/Pipes-Tutorial.html)

- Pipes have three features: effects, streaming and composability

  ➡ Producers can only yield values and they model streaming sources

  ➡ Consumers can only await values and they model streaming sinks

  ➡ Pipes can both yield and await values and they model stream transformations

  ➡ Effects can neither yield nor await and they model non-streaming components

Major Challenges

- Let’s discuss