HaPoC
2021

All Things Flow

Unfolding the History of Streams

Aggelos Biboudis

swisscom

Jeremy Gibbons

UNIVERSITY OF OXFORD

Oleg Kiselyov

TOHOKU UNIVERSITY

Thursday, October 28, 2021
6th International Conference on the History and Philosophy of Computing

Philosophy of motion

- "whether motion exists as we perceive it, what is it, and, if it exists, how does it occur."
  - pre-Socratics: Heraclitus (535 BC), Democritus
  - Parmenides: motion is only perceived but cannot actually exist (relativity for motion)
  - Zeno of Elea: infinite continuous matter, space (and time)
  - Democritus: matter and or space (and time) are discrete and finite
  - Plato, Aristotle, the Sanlun school of Mahayana Buddhism and Sengzhao (The Immutability of Things-3rd century CE), Aztecs, ...

The goal of this talk

- We lay the ground for a holistic discussion behind **streams of information**
- Motivate **cross-disciplinary curiosity**
- Target audience for an upcoming paper:
  - an emerging computer science researcher,
  - a curious software engineer; and
  - a database query optimisation specialist

von Neumann bottleneck

- a CPU
- a store
- a connecting tube that can transmit word-at-a-time rate between CPU and the store

1977 ACM Turing Award Lecture

programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specifications of programming languages.'

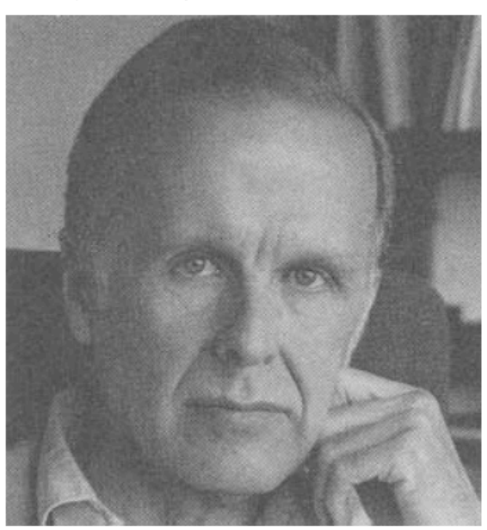
The most significant part of the full citation is as follows:

'. . . Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical com-

Thus, Backus has contributed strongly both to the pragmatic world of problem-solving on computers and to the theoretical world existing at the interface between artificial languages and computational linguistics. Fortran remains one of the most widely used programming languages in the world. Almost all programming languages are now described with some type of formal syntactic definition.' "

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating

John Backus. 1978. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. Commun. ACM 21, 8 (Aug. 1978)

Is it still applicable? (2021)

- Not word-at-a-time (out-of-order, a stream of instructions, superscalar, native SIMD, etc)
- The data bus bandwidth problem is solved by L1D and L1I, L2/L3 caches
- 64 bits nowadays
- **Backus was visionary**: how many times our mind goes to the array representation first and we think in terms of processing X-at-a-time aka the von Neumann machine-style - a style of **no equational reasoning** and **complex semantics trying to capture effects**?

Stream—a term historically used to denote:

1. a means of processing lots of data in **limited** memory;
2. capturing the **semantics** of I/O;
3. **event** processing and correlation; and
4. **iteration** abstractions.

Conway's design for a **one-pass** COBOL compiler (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.

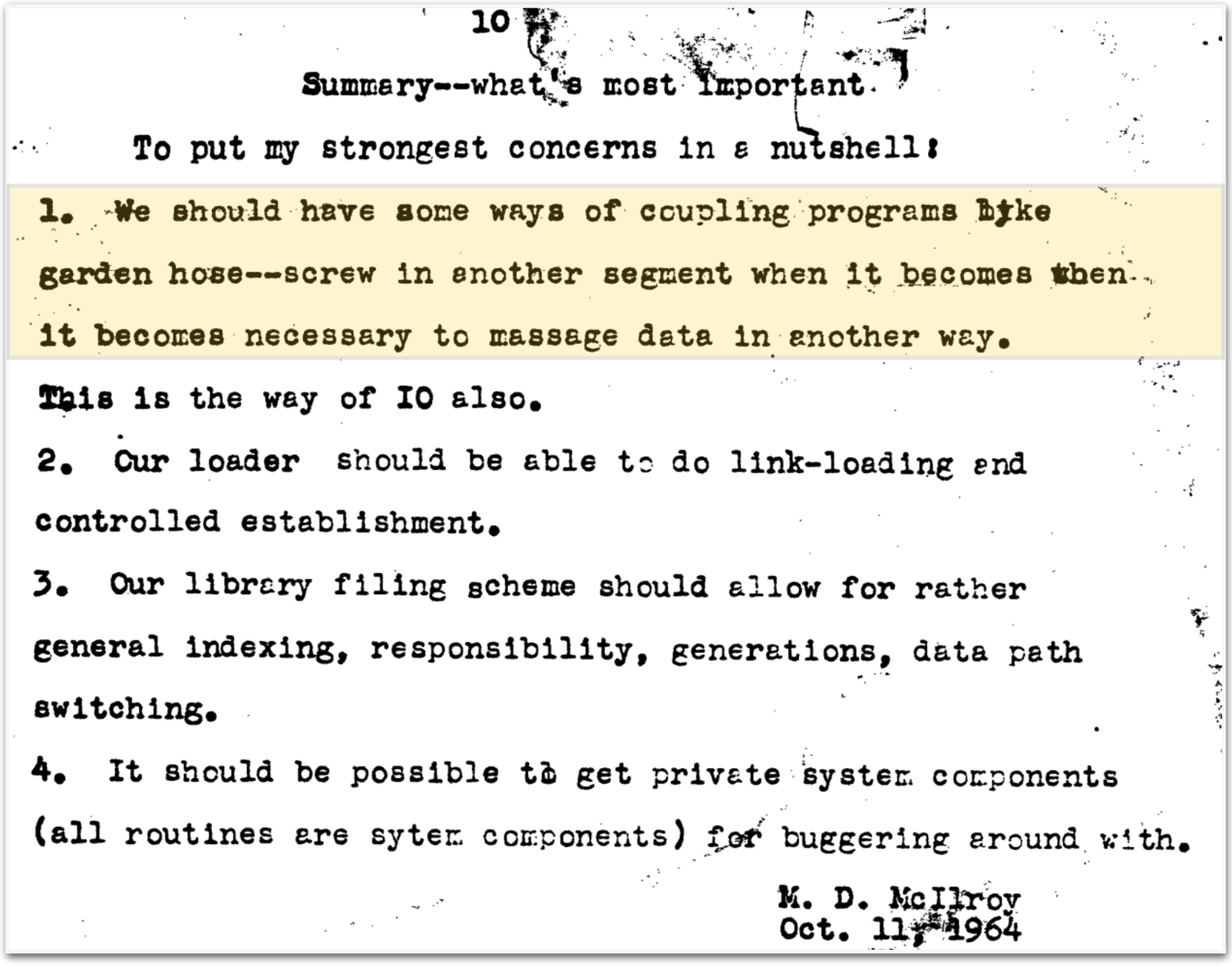
Coroutines and Separable Programs

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.

Melvin E. Conway. 1963. Design of a separable transition-diagram compiler. Commun. ACM 6, 7 (July 1963),

Streams for data processing in sub-linear space

(Doug. 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.

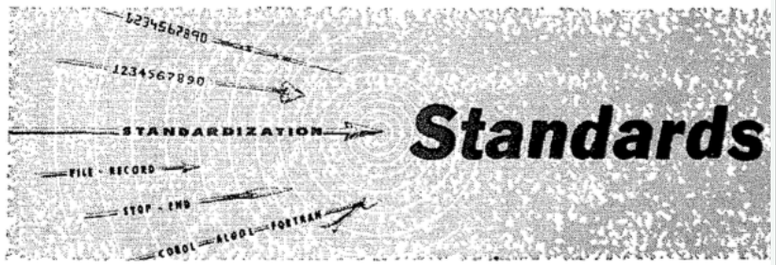
2. Our loader should be able to do link-loading and controlled establishment.

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

Capturing the semantics of for-loops and I/O



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

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

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

$$\textbf{rec}\ concatenate\ S = null\ S \rightarrow (\,)\ |\ null(hS) \rightarrow concatenate\ (tS)\ |\ \textbf{else} \rightarrow h^2S{:}concatenate(t(hS){:}tS)$$

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

$$\textbf{rec}\ while(e, p) = p \rightarrow e{:}while(e, p)\ |\ \textbf{else} \rightarrow (\,)$$

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."

P. J. Landin. 1965. Correspondence between ALGOL 60 and Church's Lambda-notation: part I. Commun. ACM 8, 2 (Feb. 1965)

Streams as event processing and correlation

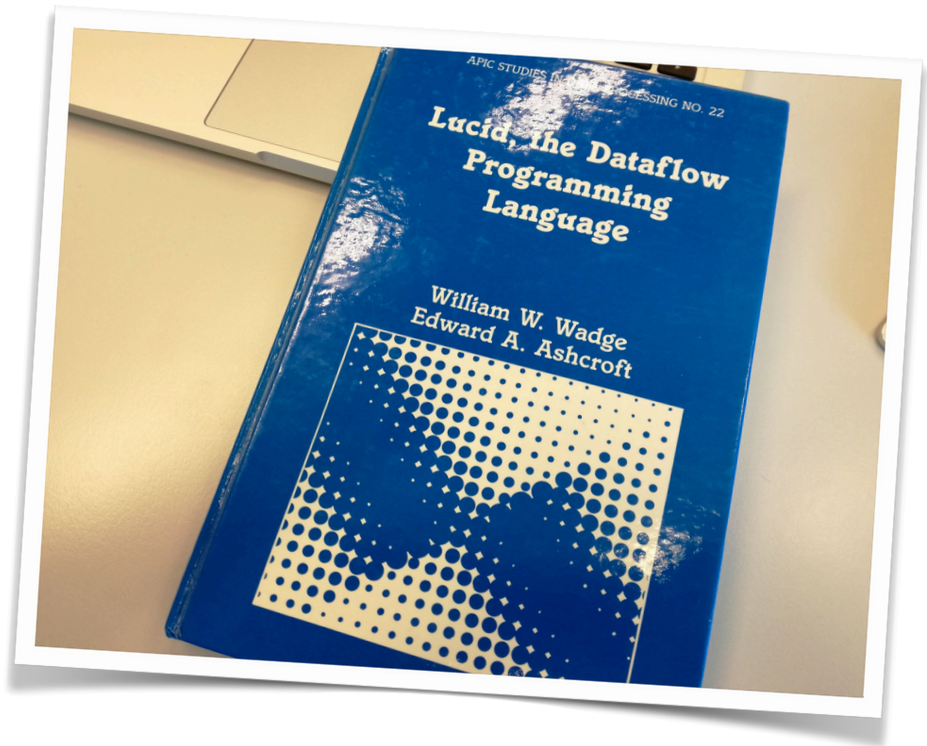
- Information flow processing: data stream processing (DSMS) vs complex event processing systems

- Events (e.g., sensor readings), triggers

Lucid (1976)

- Expressions only; **no control statements**
- Instead of “fetching” data, processing on the **flow of data**
- Network of transformations in applicative fashion
- Values of expressions: **sequences (streams) only**
- Inspired by Peter Landin’s ISWIM (1966)

```
fac
  where
    n = 0 fby (n + 1);
    fac = 1 fby (fac * (n + 1) );
  end
```



Spark Streaming, Flink, Kafka Streams, Samza, …

- Not far from the ideas of Lucid and Unix Pipes et al.
  - Distributed & publish subscribe
  - Fault Tolerance (checkpoints)
  - Delivery Guarantees (such as at-most-once, exactly-once, at-least-once)
  - State management (such as counts on records)
  - Performance

Streams as iteration abstractions

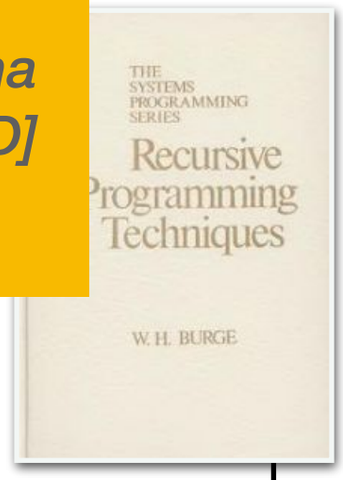
- If we can't get away from the von-Neumann philosophy lets attempt to **tame the control flow**
- **PL constructs for streaming computations** (full co-routines, yield (semi-co-routines), iterators)
- If there is a next element, transform and propagate with the **minimal** memory footprint
- Streaming libraries emerge

From Generators to Iterators

IPL (Allen Newell, Cliff Shaw, Herbert A. Simon)

1956

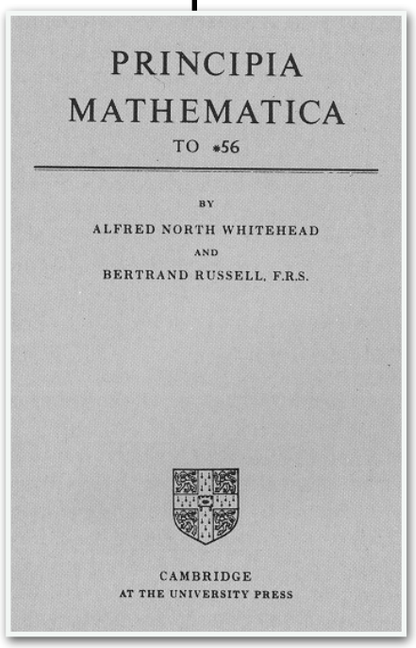
Procrastinating [SECD] Machine



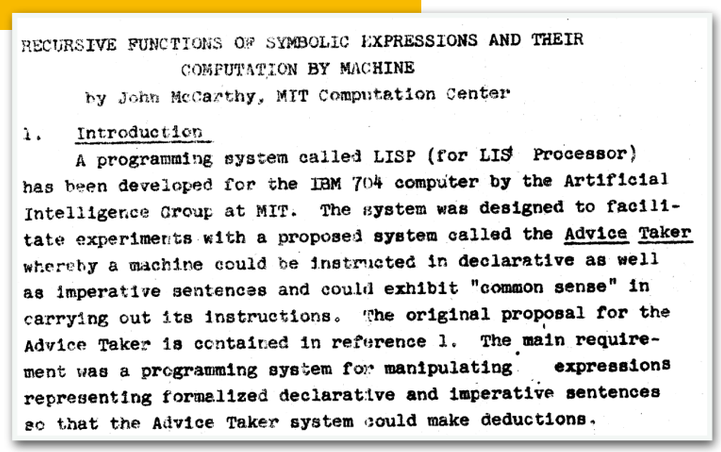
1959

CLU (Barbara Liskov)

1910-1913



LISP



RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS AND THEIR COMPUTATION BY MACHINE

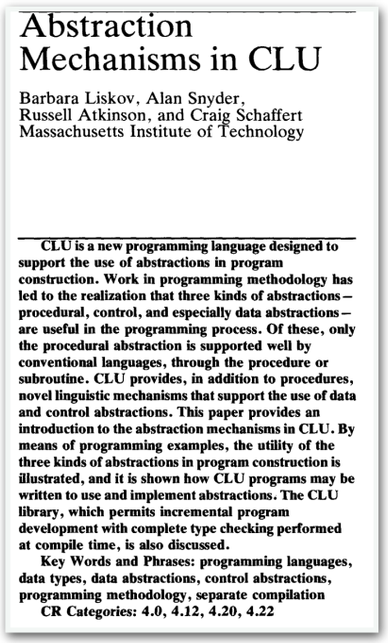
by John McCarthy, MIT Computation Center

1. Introduction

A programming system called LISP (for LIS Processor) has been developed for the IBM 704 computer by the Artificial Intelligence Group at MIT. The system was designed to facilitate experiments with a proposed system called the Advice Taker whereby a machine could be instructed in declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal for the Advice Taker is contained in reference 1. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions.

1975

1977



Abstraction Mechanisms in CLU

Barbara Liskov, Alan Snyder,
Russell Atkinson, and Craig Schaffert
Massachusetts Institute of Technology

CLU is a new programming language designed to support the use of abstractions in program construction. Work in programming methodology has led to the realization that three kinds of abstractions—procedural, control, and especially data abstractions—are useful in the programming process. Of these, only the procedural abstraction is supported well by conventional languages, through the procedure or subroutine. CLU provides, in addition to procedures, novel linguistic mechanisms that support the use of data and control abstractions. This paper provides an introduction to the abstraction mechanisms in CLU. By means of programming examples, the utility of the three kinds of abstractions in program construction is illustrated, and it is shown how CLU programs may be written to use and implement abstractions. The CLU library, which permits incremental program development with complete type checking performed at compile time, is also discussed.

Key Words and Phrases: programming languages, data types, data abstractions, control abstractions, programming methodology, separate compilation
CR Categories: 4.0, 4.12, 4.20, 4.22

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

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

W. H. Burge. 1975. Stream processing functions. IBM J. Res. Dev. 19, 1 (January 1975)

Motivating example 1: Streaming APIs

(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 */
}
```

```
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 */);
```

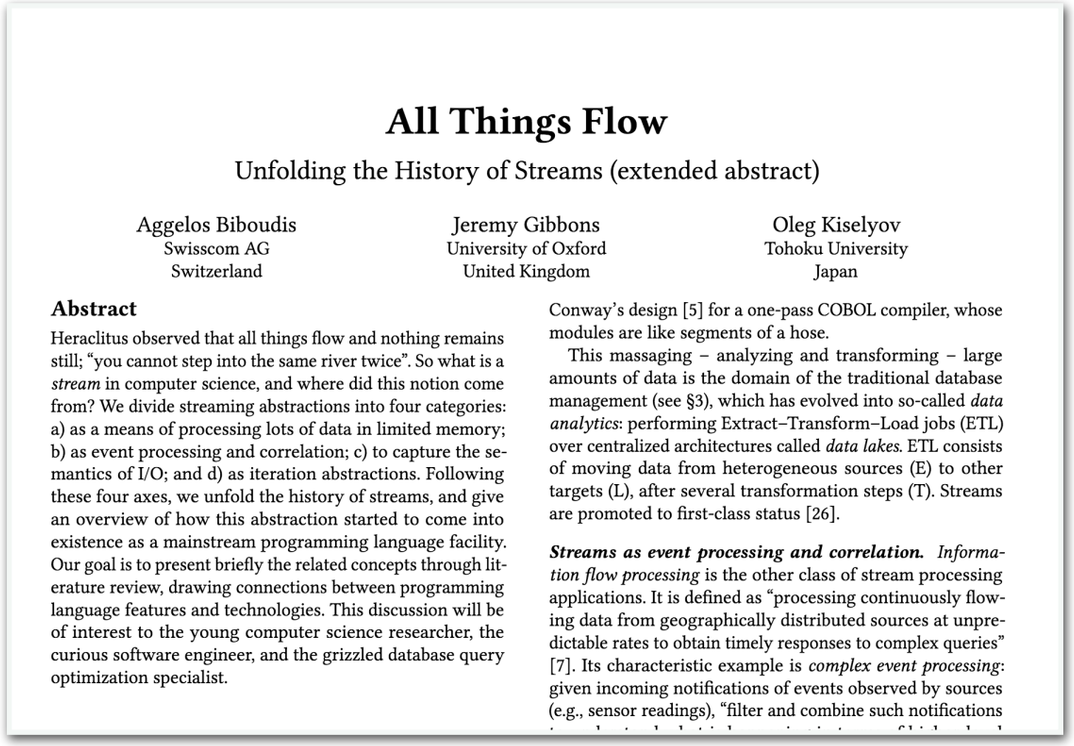
Motivating Example 2: Database Systems

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

Take-aways

- Conway, Backus, Landin, McIlroy were visionaries
- Now more than ever we are coming to appreciate their perspective
- Emerging streaming applications such as **6G/Edge networks** and **streaming tensor computations** will rely on the same principles; we lay the ground for a holistic discussion behind streams

Thank you!



All Things Flow

Unfolding the History of Streams (extended abstract)

Aggelos Biboudis
Swisscom AG
Switzerland

Jeremy Gibbons
University of Oxford
United Kingdom

Oleg Kiselyov
Tohoku University
Japan

Abstract

Heraclitus observed that all things flow and nothing remains still; "you cannot step into the same river twice". So what is a *stream* in computer science, and where did this notion come from? We divide streaming abstractions into four categories: a) as a means of processing lots of data in limited memory; b) as event processing and correlation; c) to capture the semantics of I/O; and d) as iteration abstractions. Following these four axes, we unfold the history of streams, and give an overview of how this abstraction started to come into existence as a mainstream programming language facility. Our goal is to present briefly the related concepts through literature review, drawing connections between programming language features and technologies. This discussion will be of interest to the young computer science researcher, the curious software engineer, and the grizzled database query optimization specialist.

Conway's design [5] for a one-pass COBOL compiler, whose modules are like segments of a hose.

This massaging – analyzing and transforming – large amounts of data is the domain of the traditional database management (see §3), which has evolved into so-called *data analytics*: performing Extract–Transform–Load jobs (ETL) over centralized architectures called *data lakes*. ETL consists of moving data from heterogeneous sources (E) to other targets (L), after several transformation steps (T). Streams are promoted to first-class status [26].

Streams as event processing and correlation. *Information flow processing* is the other class of stream processing applications. It is defined as "processing continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries" [7]. Its characteristic example is *complex event processing*: given incoming notifications of events observed by sources (e.g., sensor readings), "filter and combine such notifications