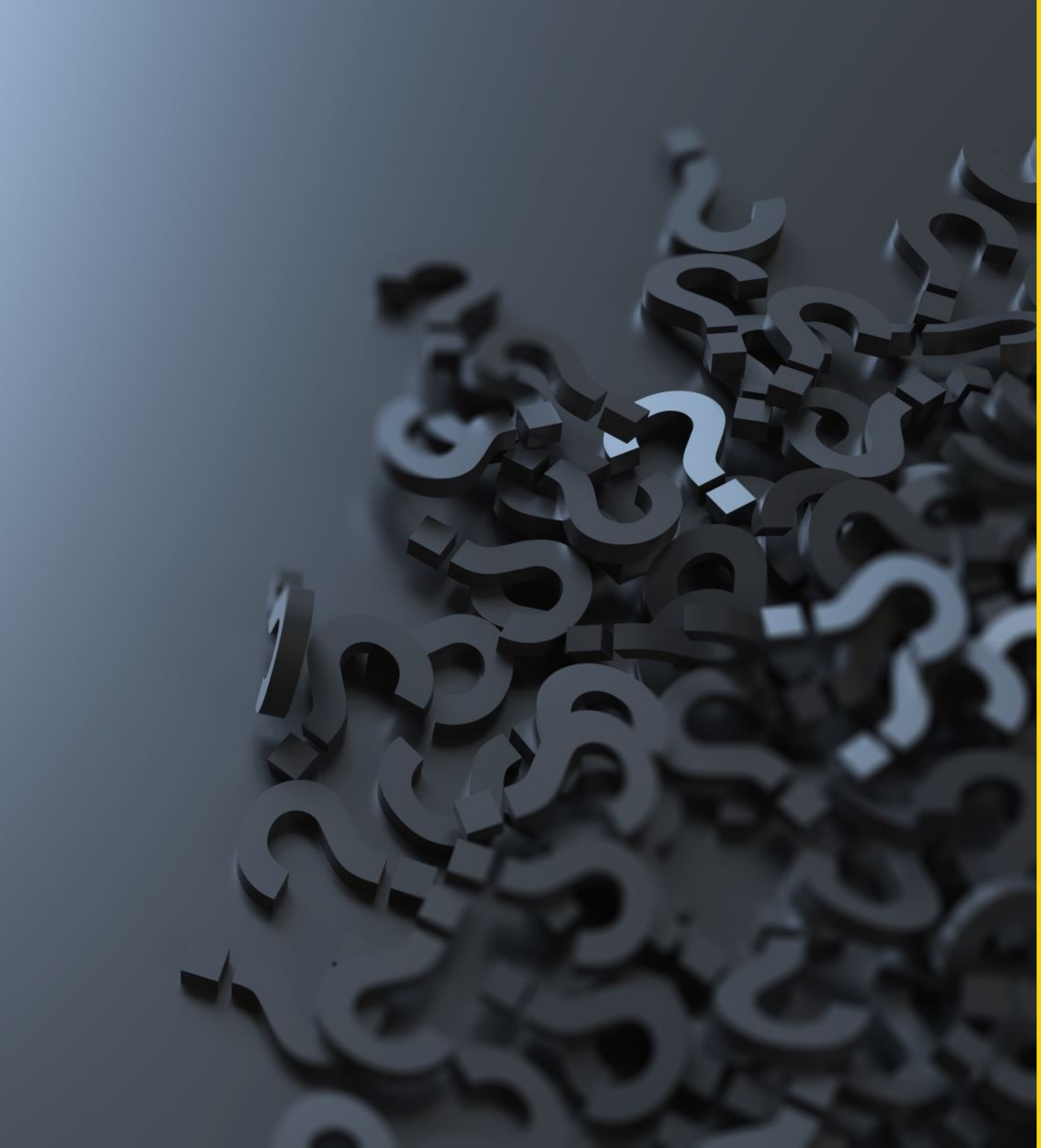
CS 295B/CS 395B Systems for Knowledge Discovery

Sound Data Collection

The University of Vermont

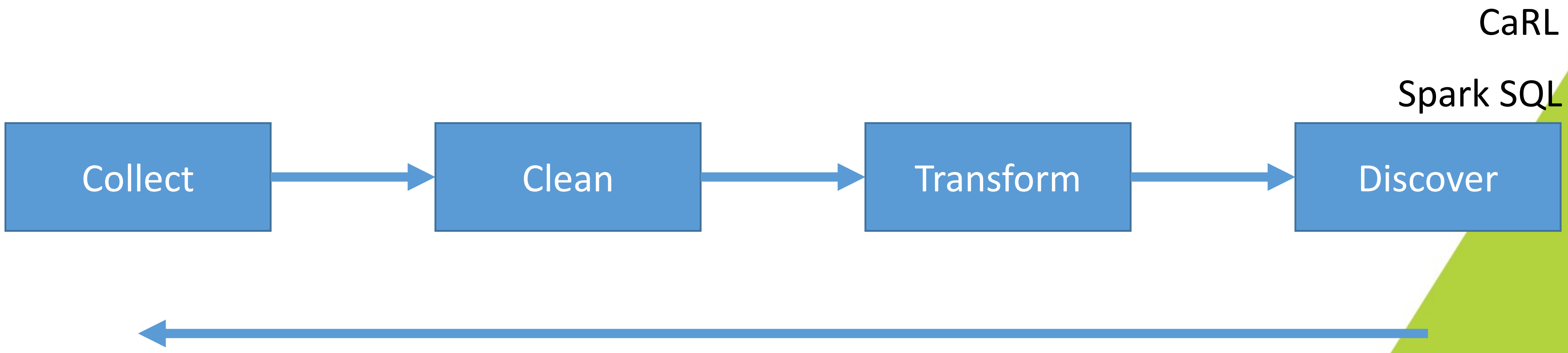


Topics

- Data collection
- Correctness
- Program Synthesis
- AI-powered inference
- Paper context

So far in this course….

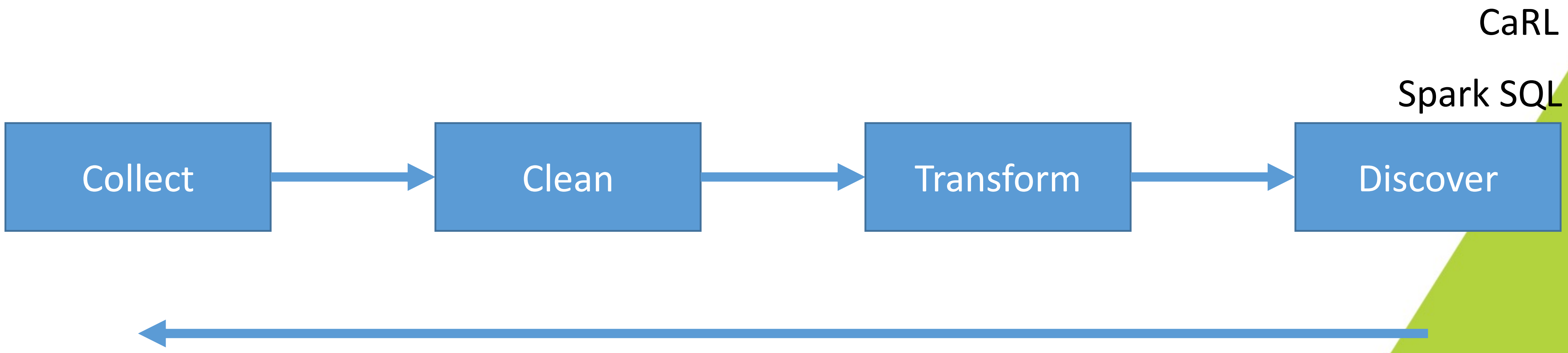
Recall the diagram from day one:



Started with query languages over structured data

So far in this course….

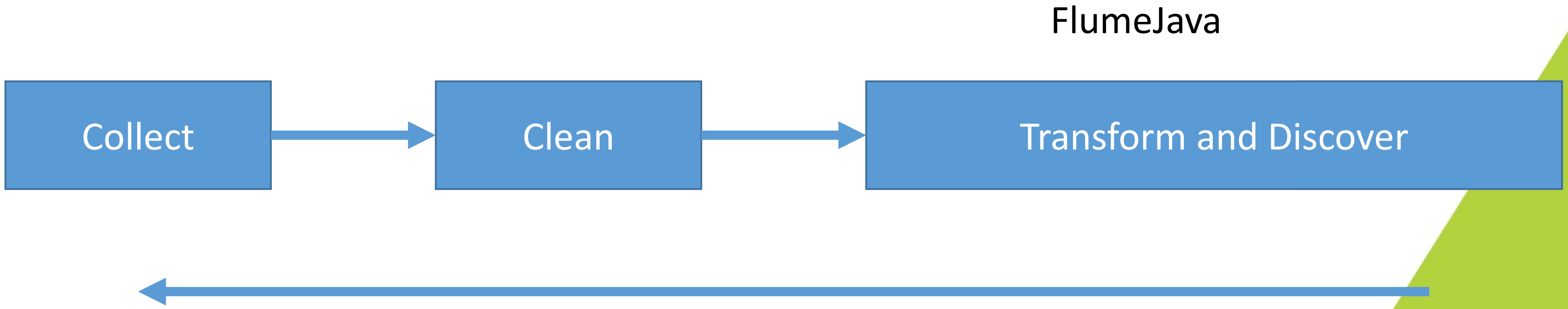
Recall the diagram from day one:



Then moved on to processing less-processed data

So far in this course….

Recall the diagram from day one:



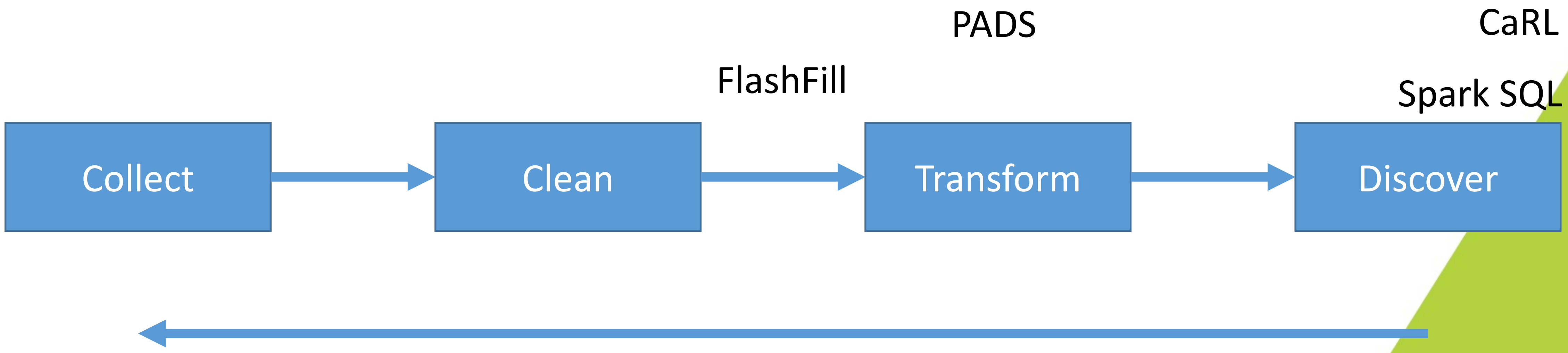
So far in this course….

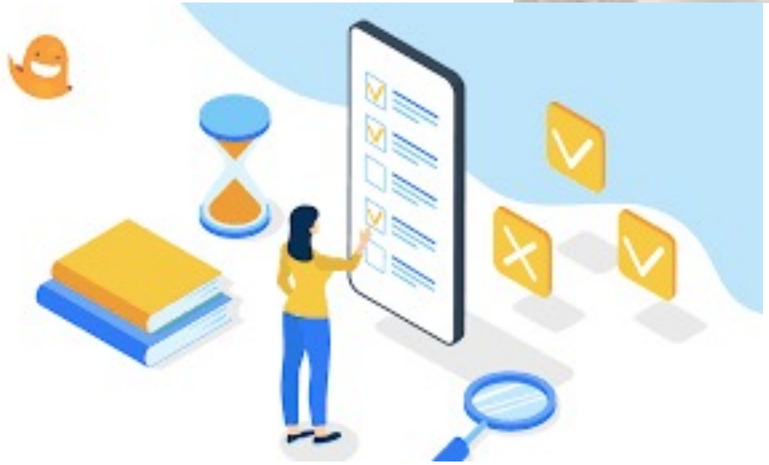
Recall the diagram from day one:

Indri
Collect and Discover

So far in this course….

Recall the diagram from day one:





What is data collection?

Typically think: active data collection

- Scraping information from the web
- Designing software for measuring phenomena
- Running surveys, collecting labels, etc.

There is a lot of passively collected data out here

Why “big data” is/was A Thing

Passive data collection

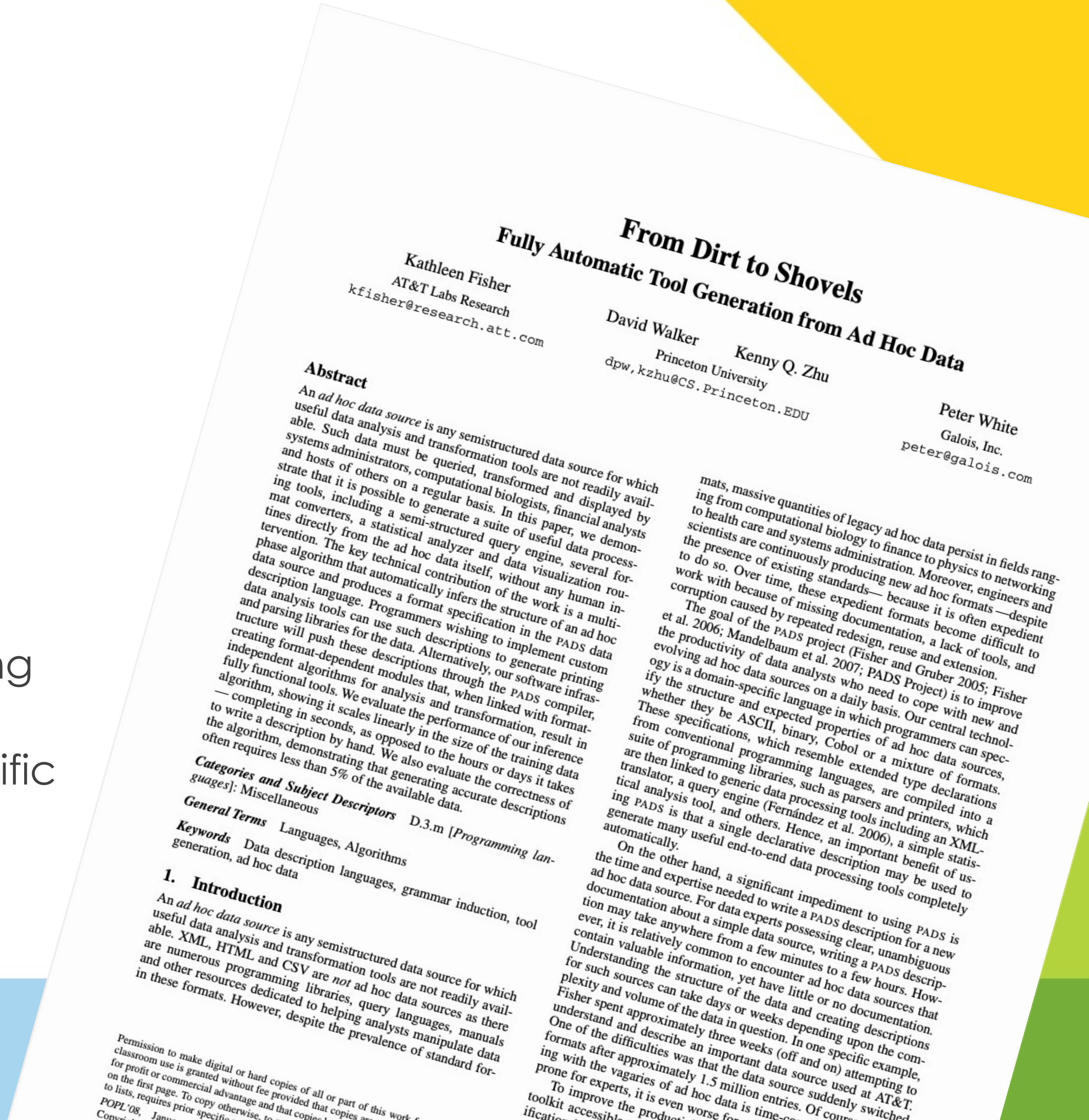
Most abundant source example:

- Logs from software systems

Problems:

- Logging typically used for debugging or auditing
- Logs are only semi-structured, application-specific

Transform



From Dirt to Shovels
Fully Automatic Tool Generation from Ad Hoc Data

Kathleen Fisher
AT&T Labs Research
kfisher@research.att.com

David Walker Kenny Q. Zhu
Princeton University
dpw,kzhu@CS.Princeton.EDU

Peter White
Galois, Inc.
peter@galois.com

Abstract

An *ad hoc data source* is any semistructured data source for which useful data analysis and transformation tools are not readily available. Such data must be queried, transformed and displayed by systems administrators, computational biologists, financial analysts and hosts of others on a regular basis. In this paper, we demonstrate that it is possible to generate a suite of useful data processing tools, including a semi-structured query engine, several format converters, a statistical analyzer and data visualization routines directly from the ad hoc data itself, without any human intervention. The key technical contribution of the work is a multi-phase algorithm that automatically infers the structure of an ad hoc data source and produces a format specification in the PADS data description language. Programmers wishing to implement custom data analysis tools can use such descriptions to generate printing and parsing libraries for the data. Alternatively, our software infrastructure will push these descriptions through the PADS compiler, creating format-dependent modules that, when linked with format-independent algorithms for analysis and transformation, result in fully functional tools. We evaluate the performance of our inference algorithm, showing it scales linearly in the size of the training data — completing in seconds, as opposed to the hours or days it takes to write a description by hand. We also evaluate the correctness of the algorithm, demonstrating that generating accurate descriptions often requires less than 5% of the available data.

Categories and Subject Descriptors D.3.m [*Programming languages*]: Miscellaneous

General Terms Languages, Algorithms

Keywords Data description languages, grammar induction, tool generation, ad hoc data

1. Introduction

An *ad hoc data source* is any semistructured data source for which useful data analysis and transformation tools are not readily available. XML, HTML and CSV are *not* ad hoc data sources as there are numerous programming libraries, query languages, manuals and other resources dedicated to helping analysts manipulate data in these formats. However, despite the prevalence of standard formats, massive quantities of legacy ad hoc data persist in fields ranging from computational biology to finance to physics to networking to health care and systems administration. Moreover, engineers and scientists are continuously producing new ad hoc formats —despite the presence of existing standards— because it is often expedient to do so. Over time, these expedient formats become difficult to work with because of missing documentation, a lack of tools, and corruption caused by repeated redesign, reuse and extension.

The goal of the PADS project (Fisher and Gruber 2005; Fisher et al. 2006; Mandelbaum et al. 2007; PADS Project) is to improve the productivity of data analysts who need to cope with new and evolving ad hoc data sources on a daily basis. Our central technology is a domain-specific language in which programmers can specify the structure and expected properties of ad hoc data sources, whether they be ASCII, binary, Cobol or a mixture of formats. These specifications, which resemble extended type declarations from conventional programming languages, are compiled into a suite of programming libraries, such as parsers and printers, which are then linked to generic data processing tools including an XML-translator, a query engine (Fernández et al. 2006), a simple statistical analysis tool, and others. Hence, an important benefit of using PADS is that a single declarative description may be used to generate many useful end-to-end data processing tools completely automatically.

On the other hand, a significant impediment to using PADS is the time and expertise needed to write a PADS description for a new ad hoc data source. For data experts possessing clear, unambiguous documentation about a simple data source, writing a PADS description may take anywhere from a few minutes to a few hours. However, it is relatively common to encounter ad hoc data sources that contain valuable information, yet have little or no documentation. Understanding the structure of the data and creating descriptions for such sources can take days or weeks depending upon the complexity and volume of the data in question. In one specific example, Fisher spent approximately three weeks (off and on) attempting to understand and describe an important data source used at AT&T. One of the difficulties was that the data source suddenly switched formats after approximately 1.5 million entries. Of cours
ing with the vagaries of ad hoc data is time-co
prone for experts, it is even worse fo

To improve the producti
toolkit accessibl
ification

Working with ad hoc data sources

Most popular programming environment:

- Spreadsheet programs (e.g., Excel)

Problems:

- Spreadsheets typically mix data organization, cleaning, and querying
- Spreadsheets only semi-structured, application-specific

Clean

Transform

Automating String Processing in Spreadsheets Using Input-Output Examples

Sumit Gulwani
Microsoft Research, Redmond, WA, USA
sumitg@microsoft.com

Abstract

We describe the design of a string programming/expression language that supports restricted forms of regular expressions, conditionals and loops. The language is expressive enough to represent a wide variety of string manipulation tasks that end-users struggle with. We describe an algorithm based on several novel concepts for synthesizing a desired program in this language from input-output examples. The synthesis algorithm is very efficient taking a fraction of a second for various benchmark examples. The synthesis algorithm is interactive and has several desirable features: it can rank multiple solutions and has fast convergence, it can detect noise in the user input, and it supports an active interaction model wherein the user is prompted to provide outputs on inputs that may have multiple computational interpretations.

The algorithm has been implemented as an interactive add-in for Microsoft Excel spreadsheet system. The prototype tool has met the golden test - it has synthesized part of itself, and has been used to solve problems beyond author's imagination.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Program Synthesis

General Terms Algorithms, Human Factors

Keywords Program Synthesis, User Intent, Programming by Example (PBE), Version Space Algebra, Spreadsheet Programming, String Manipulation

1. Introduction

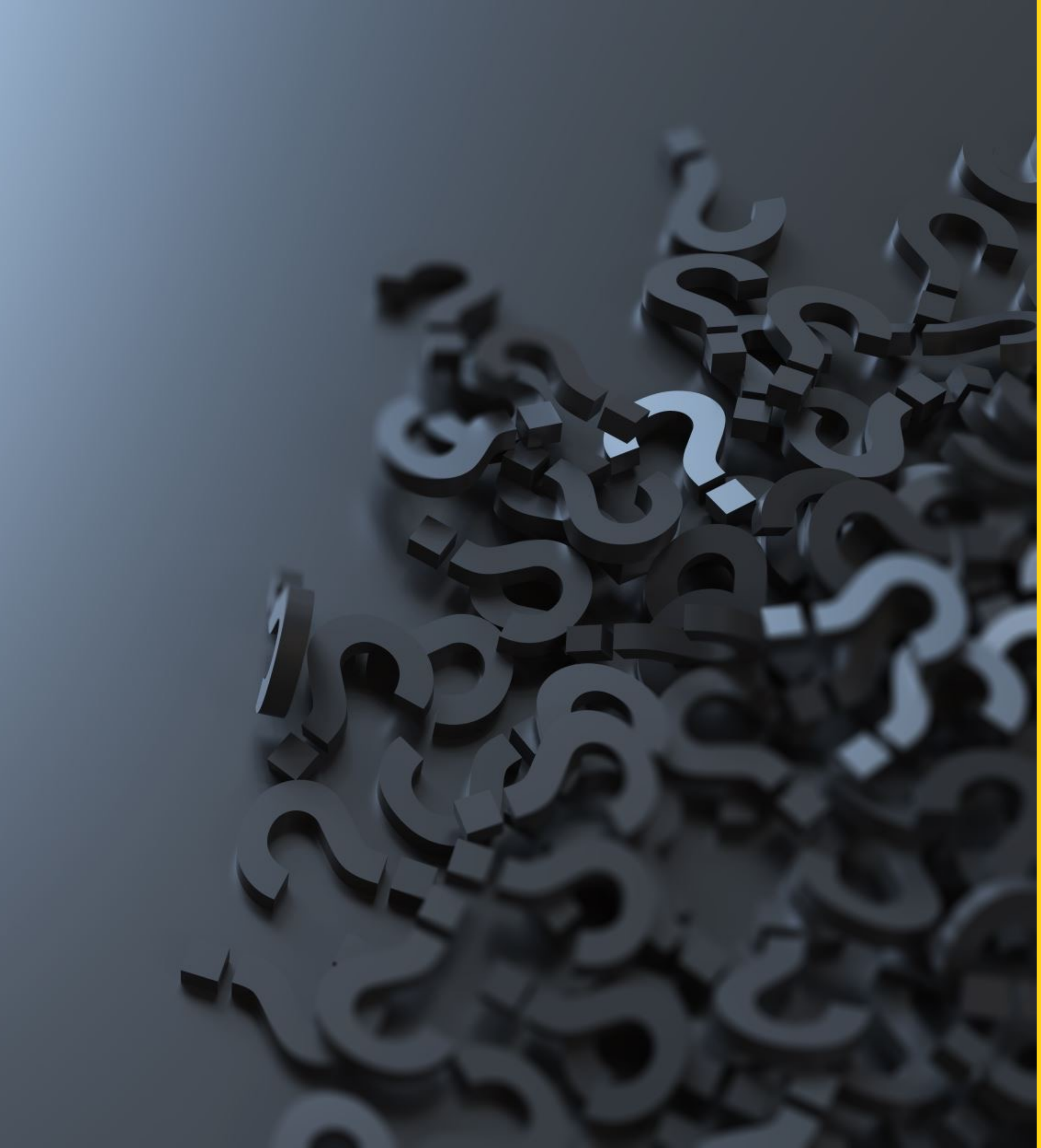
More than 500 million people worldwide use spreadsheets. These
iness *end-users* have myriad diverse backgrounds and include
ity traders, graphic designers, chemists, human resource
ce pros, marketing managers, underwriters, com-
ven mailroom clerks – they are not profes-
y need to create small, *often one-off*,
nctions [5].
adsheet programming is
e with tons of fea-
re or succes-
mplish

their task [9]. More significantly, programming is still required to perform tedious and repetitive tasks such as transforming entities like names/phone-numbers/dates from one format to another, data cleansing, extracting data from several text files or web pages into a single document, etc. Spreadsheet systems like Microsoft Excel allow users to write macros using a rich inbuilt library of string and numerical functions, or to write arbitrary scripts using a variety of programming languages like Visual Basic, or .Net. Since end-users are not proficient in programming, they find it too difficult to write desired macros or scripts.

We have performed an extensive case study of spreadsheet help forums and identified that string processing is one of the most common class of programming problems that end-users struggle with. This is not surprising given that languages like Perl, Awk, Python came into existence to support string/text processing, and that new languages like Java/C# provide a rich support for string processing. During our study of help forums, we also carefully studied how these users were describing the specification of th
desired program to the experts on the other side of the help forum
It turns out that the most common form of specification was in
output examples. Since input-output examples may lead to u
specification, the interaction between the user and the expert
involved a few rounds of communication (over multiple day

We describe a program synthesis system that is capable
thesizing a wide range of string processing programs i
sheets from input-output examples. The synthesizer aim
the role of the forum expert, which not only removes a
the loop, but also enables users to solve their probl
seconds as opposed to a few days. Our synthesis
is deployment ready, has the following important
ties.

- Fully Automated: We do not require non-
users to provide annotations/hints of any fo
- Real Time: Our system takes less than 0
per interactive round.
- Easy Interaction: Programming by exa
process where examples are added in
specification more precise. Our syste
for which the user should provide e
- Fast Convergence: Our system ty
iteration for convergence in practi
- Noise Handling: If the user ma
correct specification, our syst
and report the likely
the following



Topics

- ~~Data collection~~
- Correctness
- Program Synthesis
- AI-powered inference
- Paper context

Correctness of data collection and processing

Collection

- Correctness is statistical (usu. w.r.t. underlying population)
- Discussions later in the course

Transformation

- What kinds of guarantees can we make ahead of time?
    - Be on the lookout for correctness proofs

What are correctness proofs?

Algorithmic:

- **Soundness**
  - All returned instances have property P (100% precision)
- **Completeness**
  - Finds all instances of property P (100% recall)

What is completeness (in practice)?

When do we want it?

- Most algorithmic contexts
- Often elided

When might we not want it?

- Sometimes we just can't have it!
- Sometimes returning everything is too much

contributed articles

DOI:10.1145/1646353.1646374

How Coverity built a bug-finding tool, and a business, around the unlimited supply of bugs in software systems.

BY AL BESSEY, KEN BLOCK, BEN CHELF, ANDY CHOU, BRYAN FULTON, SETH HALLEM, CHARLES HENRI-GROS, ASYA KAMSKY, SCOTT MCPEAK, AND DAWSON ENGLER

A Few Billion Lines of Code Later Using Static Analysis to Find Bugs in the Real World

like all static bug finders, the fact that programming map clearly to source cod inspection can find man lations. For example, to "acquired locks must b checker would look for tions (such as lock() and inspect the code p rule disobedience (su no unlock() and d
For those who things, checkers in typically traverse sensitive) in a for across function while keeping t information toward the en of the suppor path was inf
A glance veals many finding.1,2, ligion wa good, an million manua numb mini muc not ma

What is soundness (in practice)?

Domain-specific

Something the researcher decides

- Do the papers provide soundness theorems?
- If they do not, why not?

Example: Soundness in type systems

Type systems: a branch of PL

Examples of types:

- Primitives/base types: Booleans, characters/strings, numbers (integers, ratios, floats)
- Tagged unions
- Records
- Tuples
- Parametric types (lists, maps, etc.)

Challenges in type systems

Two sources:

1. Can make arbitrarily complex types due to parametric types
2. Can have arbitrarily complex expressions, each subexpression having a type

How do we establish soundness in type systems?

Type soundness established through 2 properties:

1. Progress (never get stuck)

2. Preservation (types obey the rules)

Recap: Recipe for lang + type system

Misconception: that we are “proving the language correct”

Truth: Some part of the language is decidable

1. Define your language
2. Encode property P in the type system
3. Write out typing rules
4. Prove typing rules sound

Types of semantics

Typing is only meaningful in the context of syntax and semantics

Syntax

Previously: what it is

Now: what it does

Tells you how to ***build up*** *programs*

Semantic

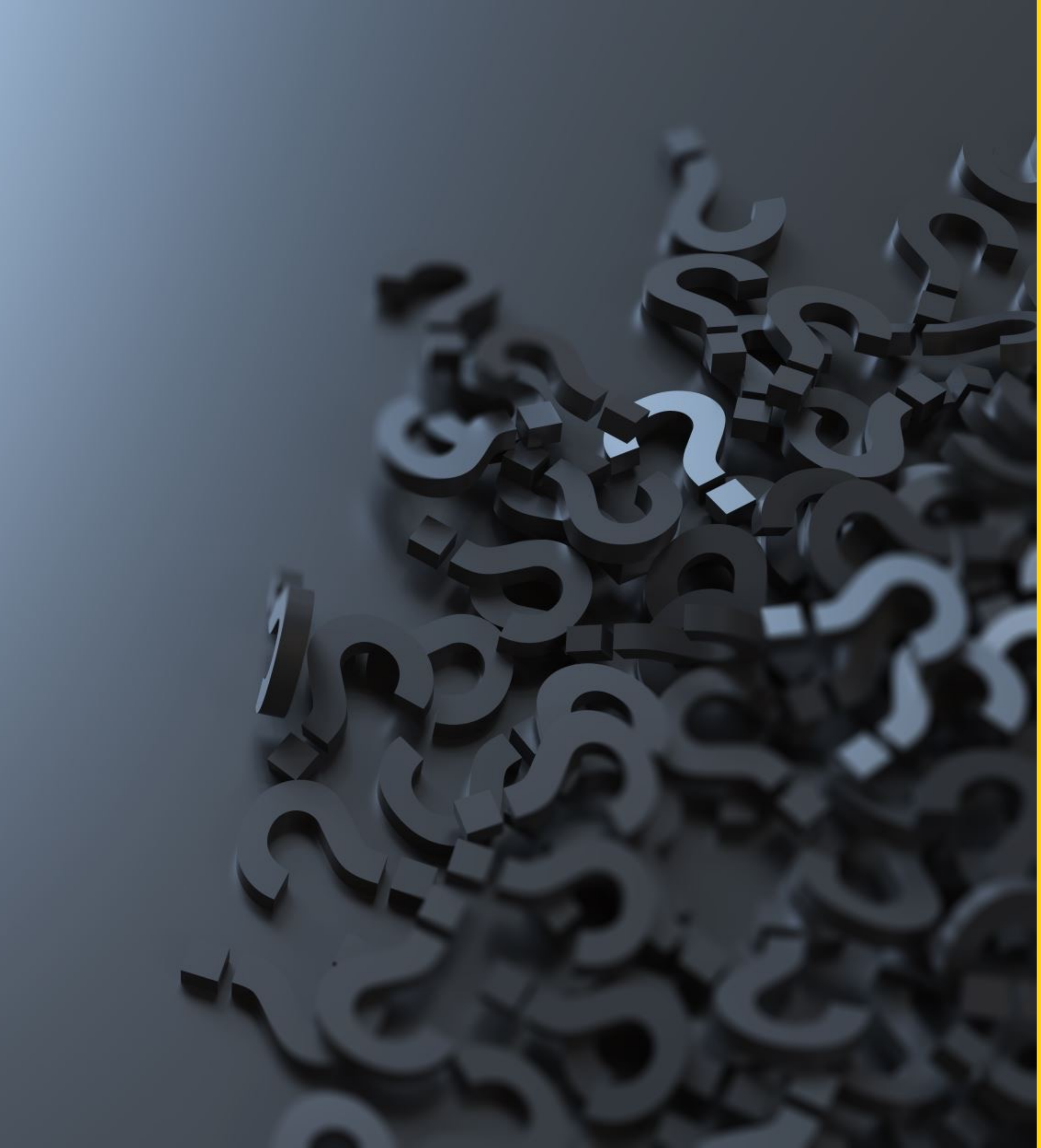
Previously: what it is

Now: what it does

Tells you how to ***evaluate down*** *programs.*

How to read denotational semantics

Do stuff on the board



Topics

- ~~Data collection~~
- ~~Correctness~~
- Program Synthesis
- AI-powered inference
- Paper context

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

John McCarthy, *Massachusetts Institute of Technology, Cambridge, Mass.*

1. Introduction

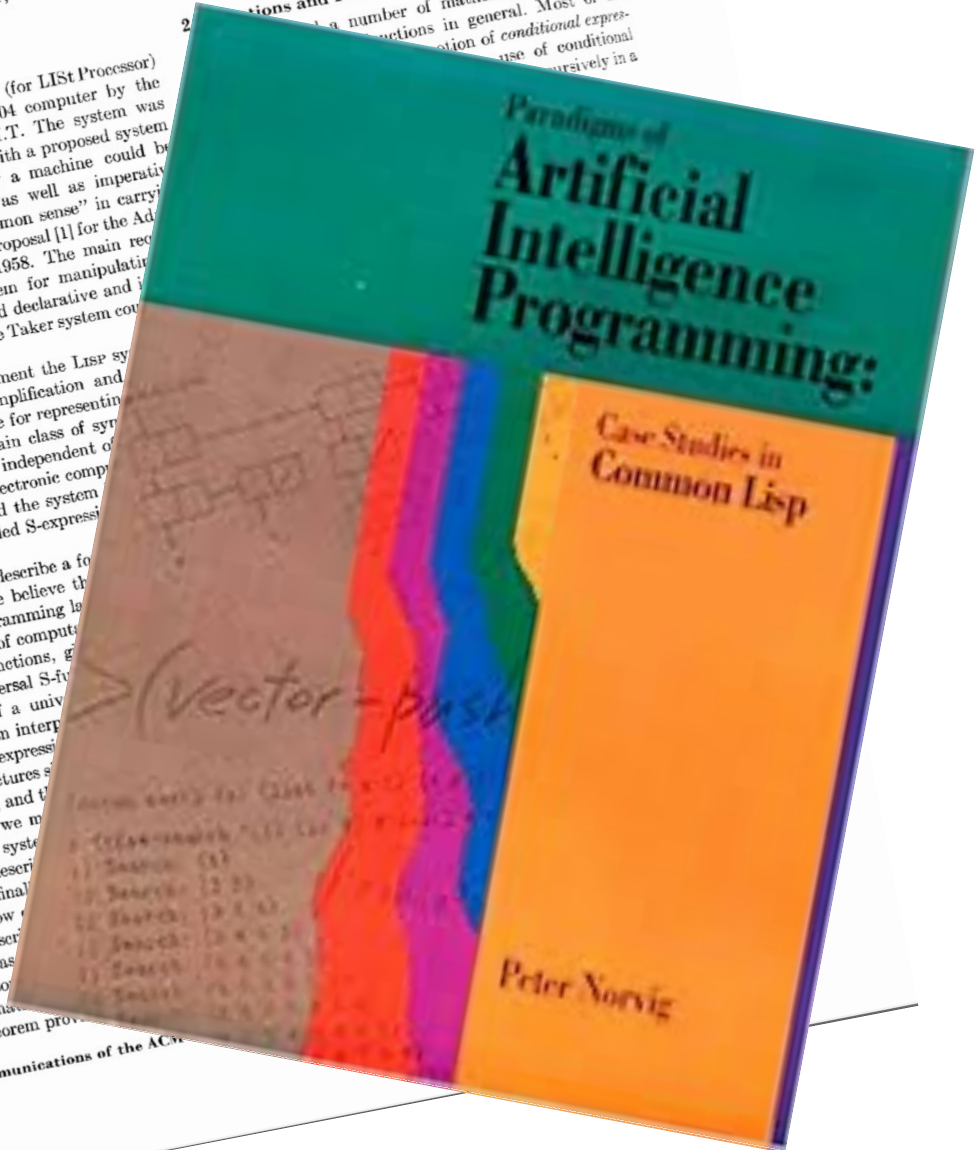
A programming system called LISP (for LISt Processor)
has been developed for the IBM 704 computer by the
Artificial Intelligence group at M.I.T. The system was
designed to facilitate experiments with a proposed system
called the Advice Taker, whereby a machine could be
instructed to handle declarative as well as imperati
sentences and could exhibit "common sense" in carryi
out its instructions. The original proposal [1] for the Ad
Taker was made in November 1958. The main rec
ment was a programming system for manipulatin
pressions representing formalized declarative and i
tive sentences so that the Advice Taker system cou
deductions.

In the course of its development the Lisp sy
through several stages of simplification and
came to be based on a scheme for representin
recursive functions of a certain class of sym
sions. This representation is independent o
computer, or of any other electronic comp
seems expedient to expound the system
the class of expressions called S-expressi
tions called S-functions.

In this article, we first describe a fo
functions recursively. We believe th
vantages both as a programming la
for developing a theory of comput
S-expressions and S-functions, g
then describe the universal S-fu
the theoretical role of a univ
the practical role of an interp
representation of S-expressi
IBM 704 by list structures s
Shaw and Simon [2], and t
by program. Then we m
Lisp programming syste
another way of descri
expressions, and final
terpretation of flow
We hope to descr
for which Lisp has
give elsewhere so
formalism to ma
mechanical theorem pro

2. ...ions and Function Definitions

...a number of mathematical ideas and
...ctions in general. Most of the
...tion of *conditional expres-*
...use of conditional
...rsively in a



Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp

Peter Norvig

Program Synthesis

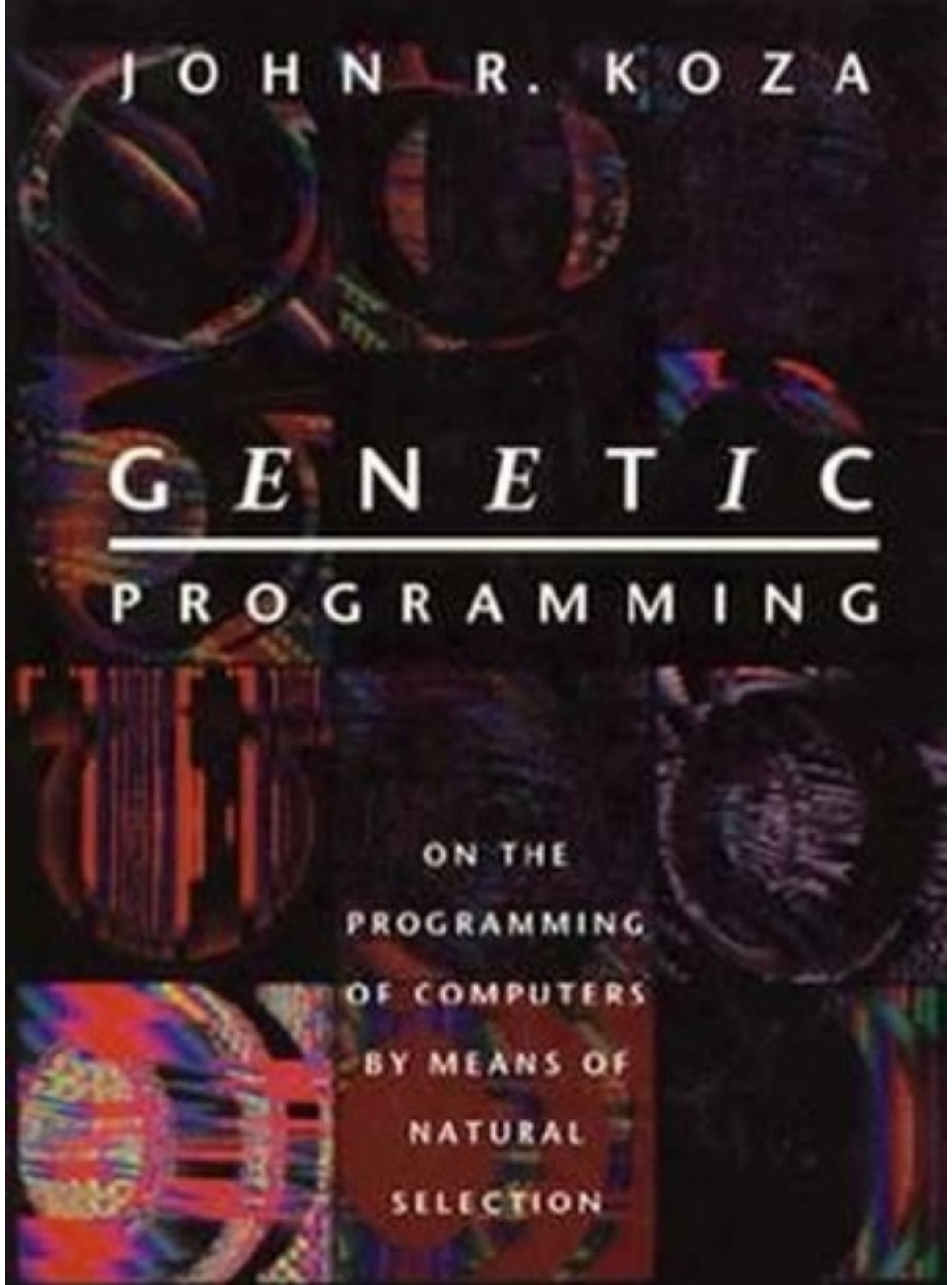
Long-desired goal of computer science (& AI)

- LISP: code is data
- Contrast with code generation
- Genetic Programming (stochastic program synthesis)
  - A kind of evolutionary method

ds

algorithms

ion +


JOHN R. KOZA
GENETIC
PROGRAMMING
ON THE PROGRAMMING OF COMPUTERS BY MEANS OF NATURAL SELECTION

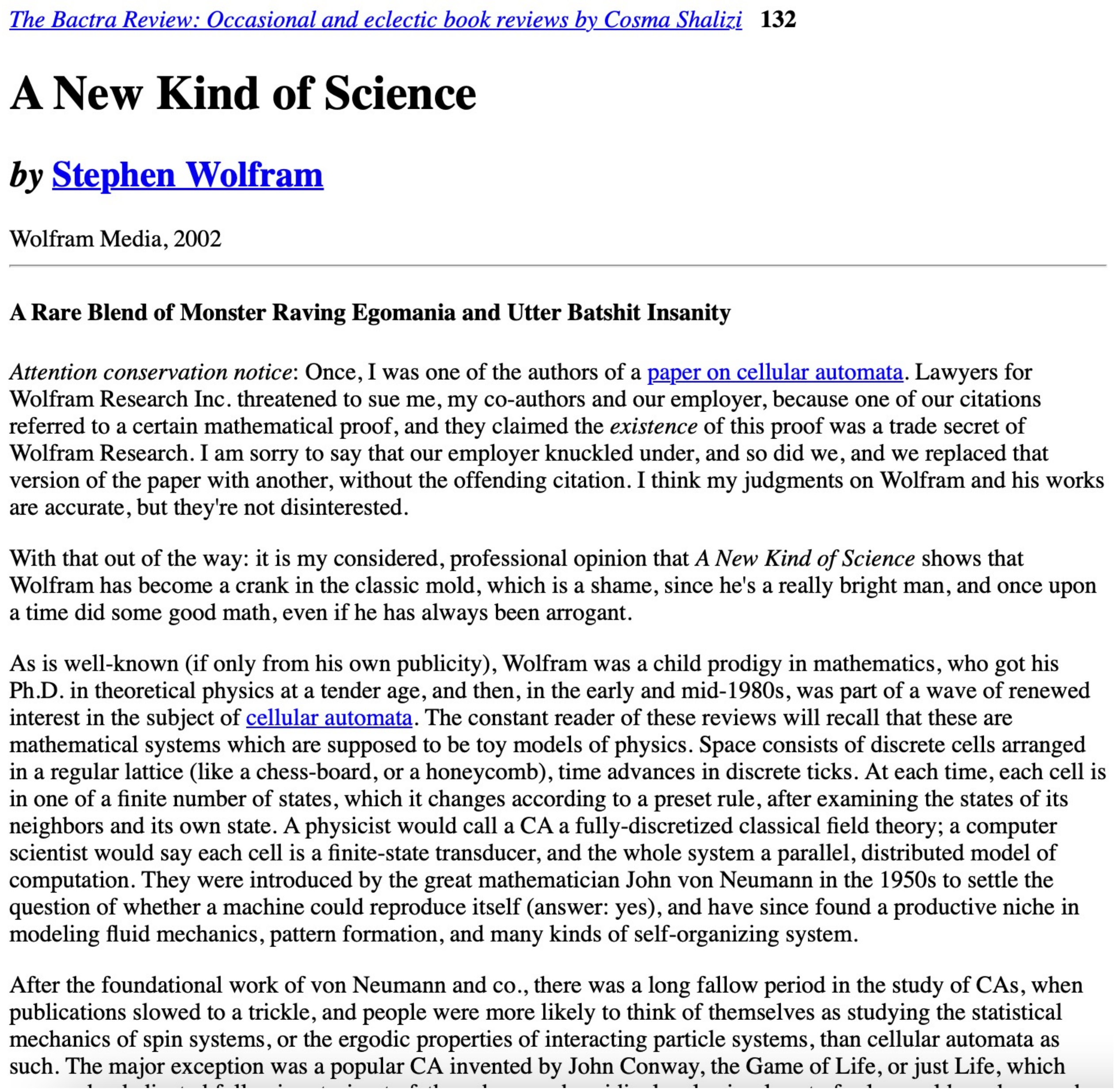
Genetic Programming

Idea: Apply principles from genetic algorithms to programs, not just vectors of numbers.

What is it good for?

- Domains requiring structured output
- Domains where you have very little knowledge *a priori*
- Domains where you just need a solution

Warning!



The Bactra Review: Occasional and eclectic book reviews by Cosma Shalizi **132**

A New Kind of Science

by Stephen Wolfram

Wolfram Media, 2002

A Rare Blend of Monster Raving Egomania and Utter Batshit Insanity

Attention conservation notice: Once, I was one of the authors of a paper on cellular automata. Lawyers for Wolfram Research Inc. threatened to sue me, my co-authors and our employer, because one of our citations referred to a certain mathematical proof, and they claimed the *existence* of this proof was a trade secret of Wolfram Research. I am sorry to say that our employer knuckled under, and so did we, and we replaced that version of the paper with another, without the offending citation. I think my judgments on Wolfram and his works are accurate, but they're not disinterested.

With that out of the way: it is my considered, professional opinion that *A New Kind of Science* shows that Wolfram has become a crank in the classic mold, which is a shame, since he's a really bright man, and once upon a time did some good math, even if he has always been arrogant.

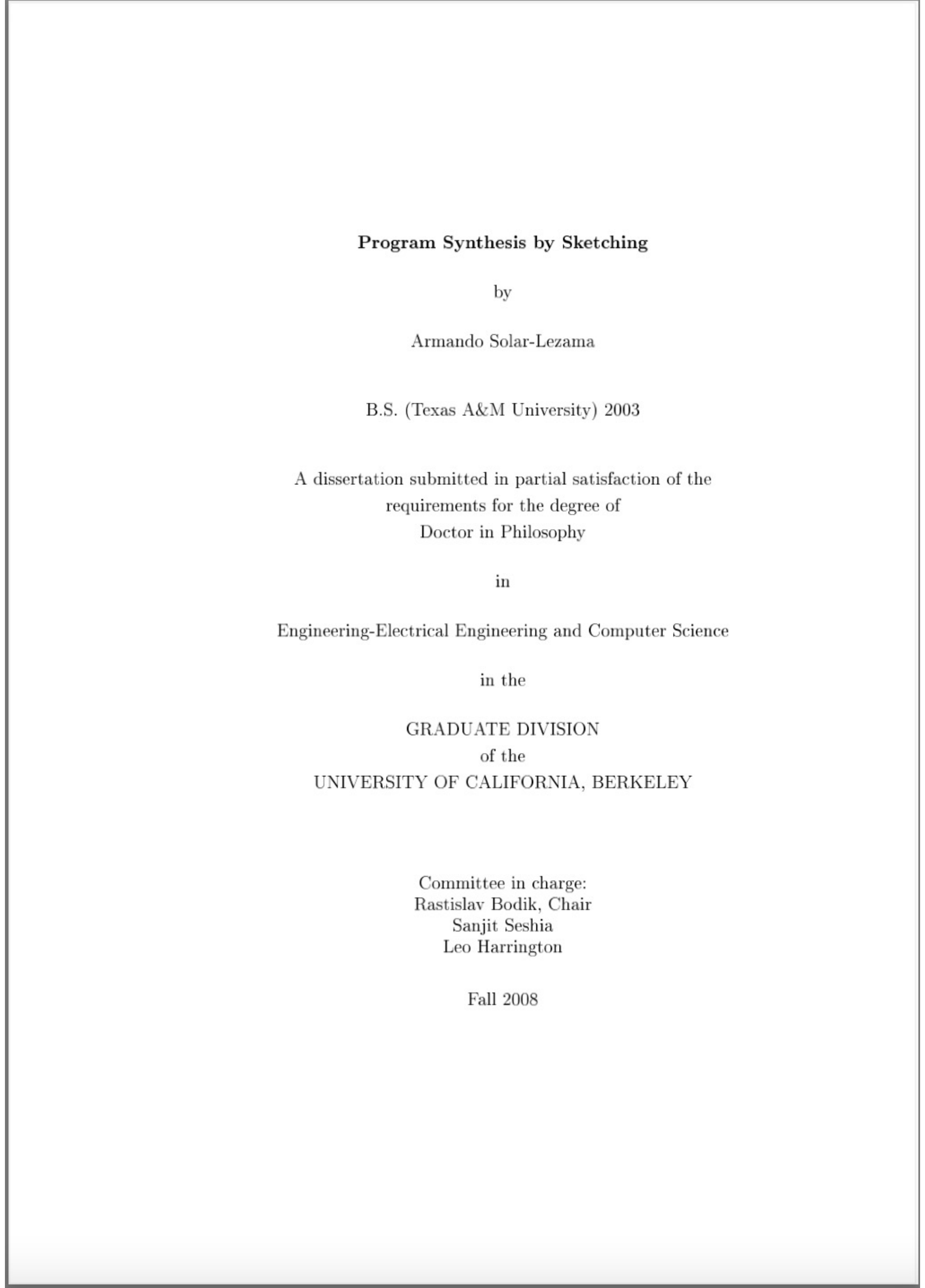
As is well-known (if only from his own publicity), Wolfram was a child prodigy in mathematics, who got his Ph.D. in theoretical physics at a tender age, and then, in the early and mid-1980s, was part of a wave of renewed interest in the subject of cellular automata. The constant reader of these reviews will recall that these are mathematical systems which are supposed to be toy models of physics. Space consists of discrete cells arranged in a regular lattice (like a chess-board, or a honeycomb), time advances in discrete ticks. At each time, each cell is in one of a finite number of states, which it changes according to a preset rule, after examining the states of its neighbors and its own state. A physicist would call a CA a fully-discretized classical field theory; a computer scientist would say each cell is a finite-state transducer, and the whole system a parallel, distributed model of computation. They were introduced by the great mathematician John von Neumann in the 1950s to settle the question of whether a machine could reproduce itself (answer: yes), and have since found a productive niche in modeling fluid mechanics, pattern formation, and many kinds of self-organizing system.

After the foundational work of von Neumann and co., there was a long fallow period in the study of CAs, when publications slowed to a trickle, and people were more likely to think of themselves as studying the statistical mechanics of spin systems, or the ergodic properties of interacting particle systems, than cellular automata as such. The major exception was a popular CA invented by John Conway, the Game of Life, or just Life, which

Counter-example guided inductive synthesis (CEGIS)

GOAL: LEARN GENERATING PROGRAM FROM INPUT-OUTPUT PAIRS

USE FORMAL METHODS TO FIND



Program Synthesis by Sketching

by

Armando Solar-Lezama

B.S. (Texas A&M University) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor in Philosophy

in

Engineering-Electrical Engineering and Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Rastislav Bodik, Chair
Sanjit Seshia
Leo Harrington

Fall 2008

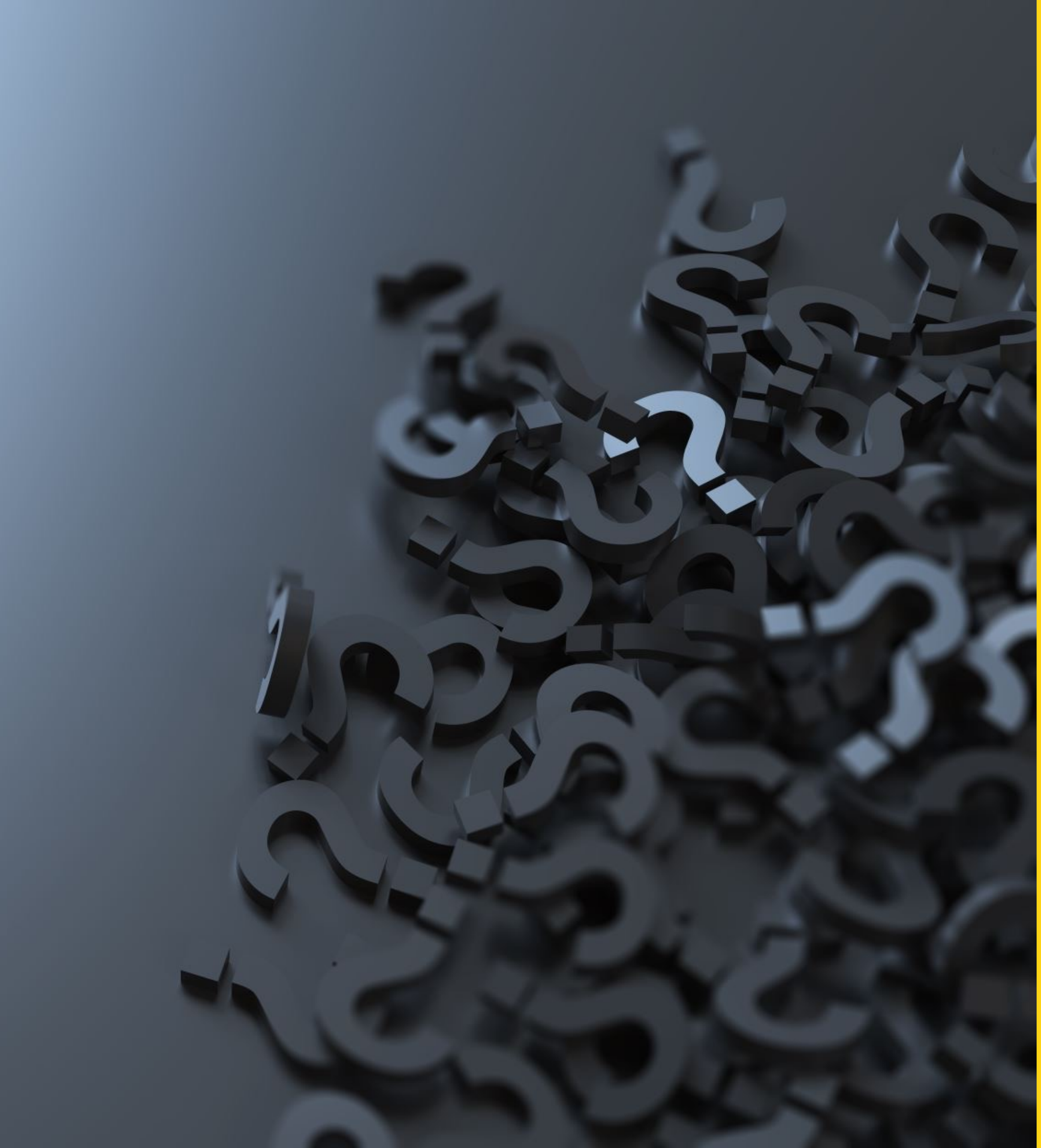
CEGIS: Origins

"Sketch" a program

- i.e., write a program that has "holes" in it
  - Leverage well-known techniques in PL for reasoning about "partial programs"

Use an SMT solver

- Previously an "AI thing"



Topics

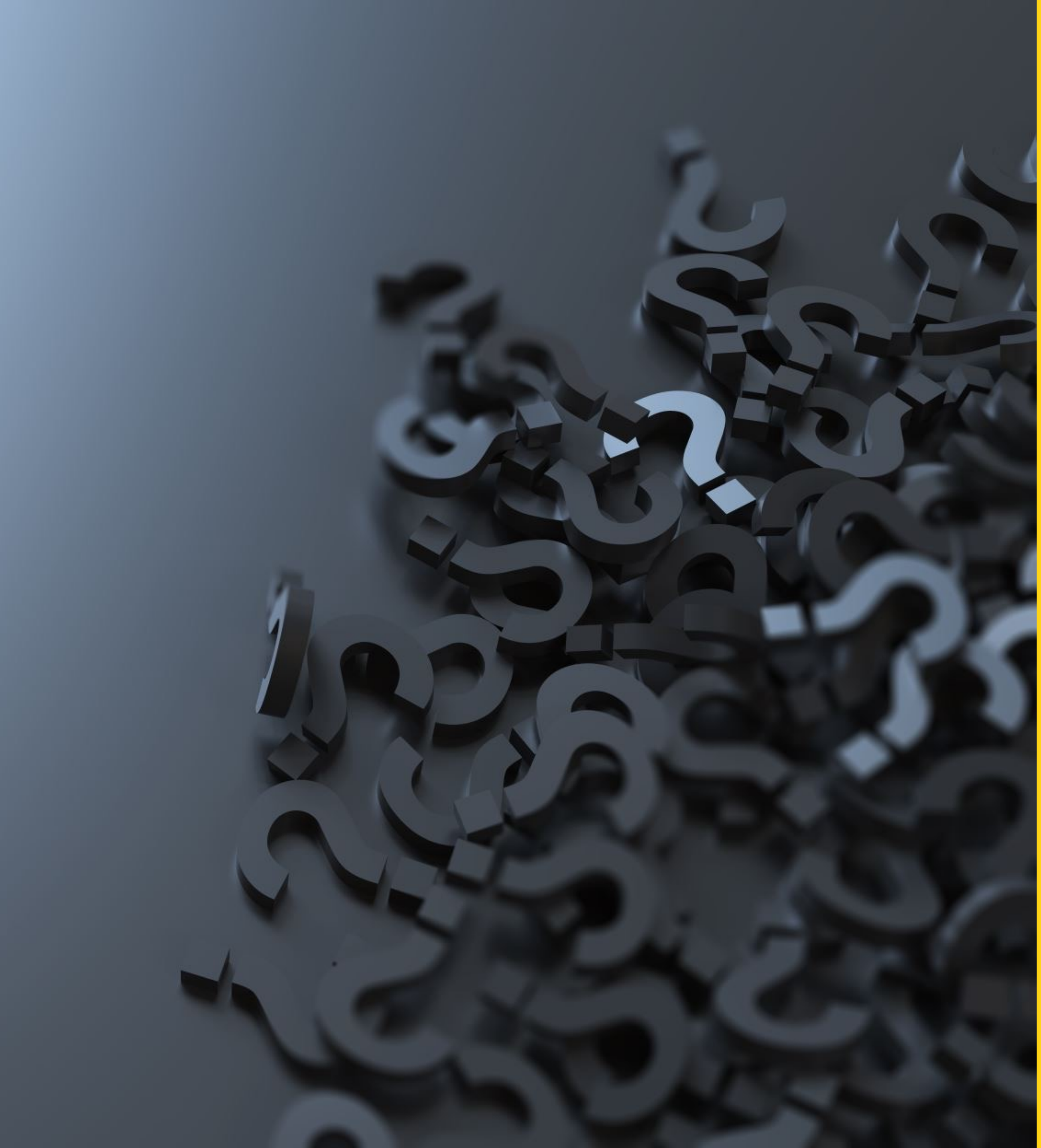
- ~~Data collection~~
- ~~Correctness~~
- ~~Program Synthesis~~
- AI-powered inference
- Paper context

AI → PL

Both papers feature *completely different* techniques from AI

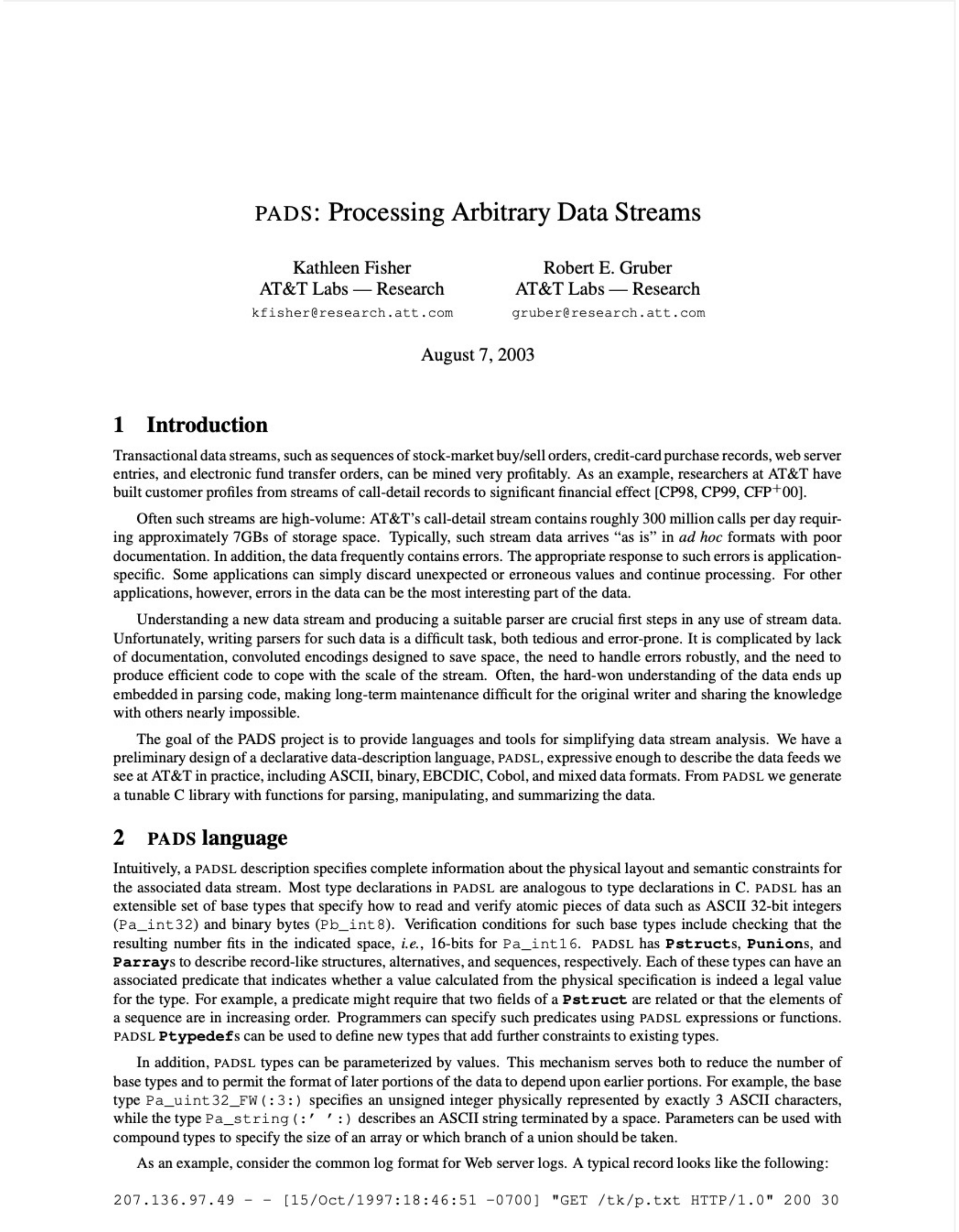
- PADS: Structure Discovery
- FlashFill: Version Space Algebras

Will see more from our presenters on Monday



Topics

- ~~Data collection~~
- ~~Correctness~~
- ~~Program Synthesis~~
- ~~AI-powered inference~~
- Paper context



PADS: Processing Arbitrary Data Streams

Kathleen Fisher
AT&T Labs — Research
kfisher@research.att.com

Robert E. Gruber
AT&T Labs — Research
gruber@research.att.com

August 7, 2003

1 Introduction

Transactional data streams, such as sequences of stock-market buy/sell orders, credit-card purchase records, web server entries, and electronic fund transfer orders, can be mined very profitably. As an example, researchers at AT&T have built customer profiles from streams of call-detail records to significant financial effect [CP98, CP99, CFP+00].

Often such streams are high-volume: AT&T's call-detail stream contains roughly 300 million calls per day requiring approximately 7GBs of storage space. Typically, such stream data arrives "as is" in *ad hoc* formats with poor documentation. In addition, the data frequently contains errors. The appropriate response to such errors is application-specific. Some applications can simply discard unexpected or erroneous values and continue processing. For other applications, however, errors in the data can be the most interesting part of the data.

Understanding a new data stream and producing a suitable parser are crucial first steps in any use of stream data. Unfortunately, writing parsers for such data is a difficult task, both tedious and error-prone. It is complicated by lack of documentation, convoluted encodings designed to save space, the need to handle errors robustly, and the need to produce efficient code to cope with the scale of the stream. Often, the hard-won understanding of the data ends up embedded in parsing code, making long-term maintenance difficult for the original writer and sharing the knowledge with others nearly impossible.

The goal of the PADS project is to provide languages and tools for simplifying data stream analysis. We have a preliminary design of a declarative data-description language, PADSL, expressive enough to describe the data feeds we see at AT&T in practice, including ASCII, binary, EBCDIC, Cobol, and mixed data formats. From PADSL we generate a tunable C library with functions for parsing, manipulating, and summarizing the data.

2 PADS language

Intuitively, a PADSL description specifies complete information about the physical layout and semantic constraints for the associated data stream. Most type declarations in PADSL are analogous to type declarations in C. PADSL has an extensible set of base types that specify how to read and verify atomic pieces of data such as ASCII 32-bit integers (`Pa_int32`) and binary bytes (`Pb_int8`). Verification conditions for such base types include checking that the resulting number fits in the indicated space, *i.e.*, 16-bits for `Pa_int16`. PADSL has **`Pstruct`**s, **`Punion`**s, and **`Parray`**s to describe record-like structures, alternatives, and sequences, respectively. Each of these types can have an associated predicate that indicates whether a value calculated from the physical specification is indeed a legal value for the type. For example, a predicate might require that two fields of a **`Pstruct`** are related or that the elements of a sequence are in increasing order. Programmers can specify such predicates using PADSL expressions or functions. PADSL **`Ptypedef`**s can be used to define new types that add further constraints to existing types.

In addition, PADSL types can be parameterized by values. This mechanism serves both to reduce the number of base types and to permit the format of later portions of the data to depend upon earlier portions. For example, the base type `Pa_uint32_FW(:3:)` specifies an unsigned integer physically represented by exactly 3 ASCII characters, while the type `Pa_string(:' ':)` describes an ASCII string terminated by a space. Parameters can be used with compound types to specify the size of an array or which branch of a union should be taken.

As an example, consider the common log format for Web server logs. A typical record looks like the following:

```
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700] "GET /tk/p.txt HTTP/1.0" 200 30
```

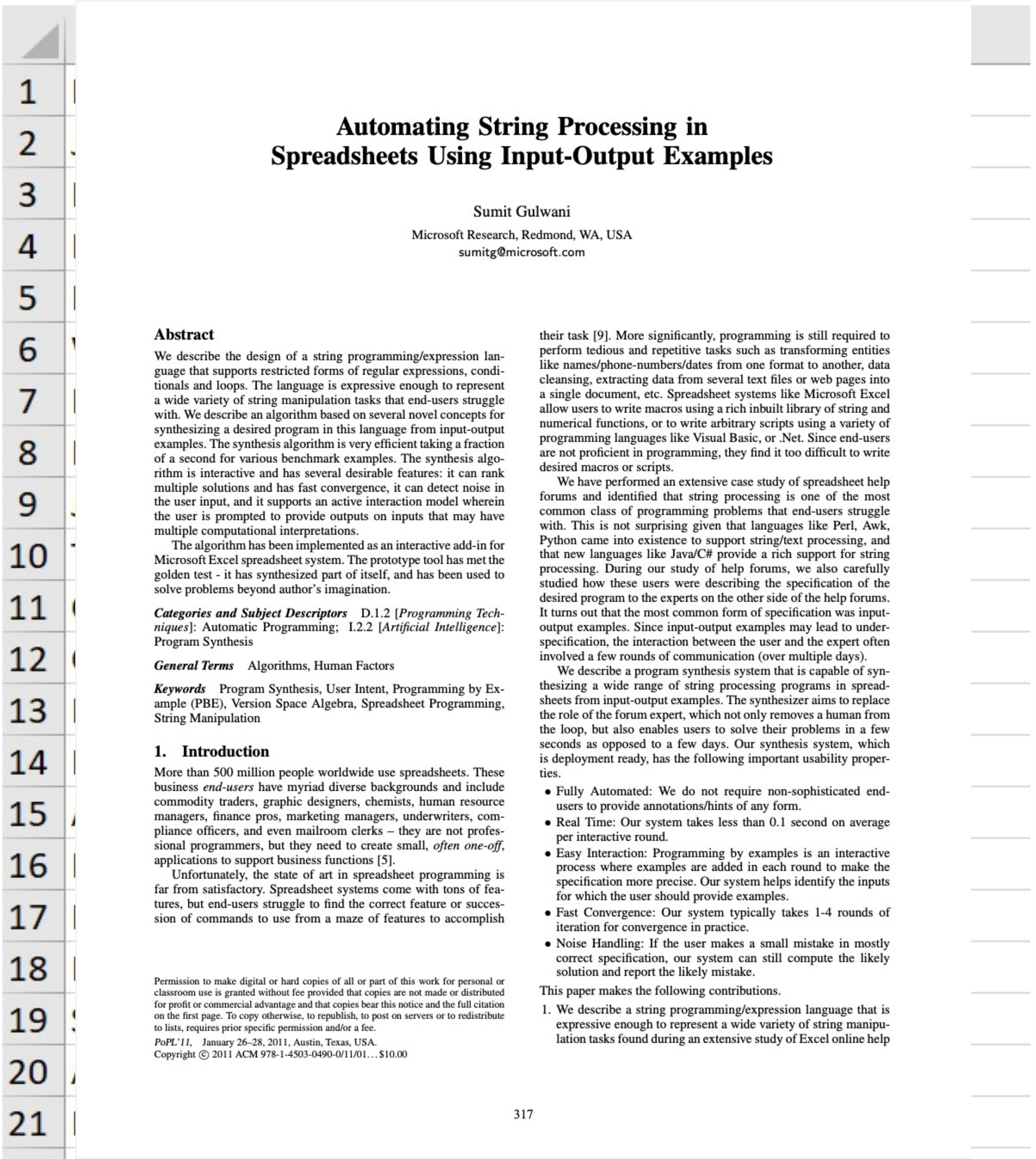
Context: PADS project

Project active years: 2001—2010ish

- First paper: *PADS: Processing Arbitrary Data Streams*
- Retrospective in 2011
- Haskell repo last updated in 2019
- C repo last updated in 2015

XML, JSON, YAML, TOML

- These are all structured data
- Easier to just write something that outputs one of these formats
- History
    - XML – bloat
    - JSON – inefficient, hard to read
    - YAML – user-friendly, ambiguous grammar
    - TOML – current config fave



Automating String Processing in Spreadsheets Using Input-Output Examples

Sumit Gulwani

Microsoft Research, Redmond, WA, USA

sumitg@microsoft.com

Abstract

We describe the design of a string programming/expression language that supports restricted forms of regular expressions, conditionals and loops. The language is expressive enough to represent a wide variety of string manipulation tasks that end-users struggle with. We describe an algorithm based on several novel concepts for synthesizing a desired program in this language from input-output examples. The synthesis algorithm is very efficient taking a fraction of a second for various benchmark examples. The synthesis algorithm is interactive and has several desirable features: it can rank multiple solutions and has fast convergence, it can detect noise in the user input, and it supports an active interaction model wherein the user is prompted to provide outputs on inputs that may have multiple computational interpretations.

The algorithm has been implemented as an interactive add-in for Microsoft Excel spreadsheet system. The prototype tool has met the golden test - it has synthesized part of itself, and has been used to solve problems beyond author's imagination.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Program Synthesis

General Terms Algorithms, Human Factors

Keywords Program Synthesis, User Intent, Programming by Example (PBE), Version Space Algebra, Spreadsheet Programming, String Manipulation

1. Introduction

More than 500 million people worldwide use spreadsheets. These business *end-users* have myriad diverse backgrounds and include commodity traders, graphic designers, chemists, human resource managers, finance pros, marketing managers, underwriters, compliance officers, and even mailroom clerks – they are not professional programmers, but they need to create small, *often one-off*, applications to support business functions [5].

Unfortunately, the state of art in spreadsheet programming is far from satisfactory. Spreadsheet systems come with tons of features, but end-users struggle to find the correct feature or succession of commands to use from a maze of features to accomplish their task [9]. More significantly, programming is still required to perform tedious and repetitive tasks such as transforming entities like names/phone-numbers/dates from one format to another, data cleansing, extracting data from several text files or web pages into a single document, etc. Spreadsheet systems like Microsoft Excel allow users to write macros using a rich inbuilt library of string and numerical functions, or to write arbitrary scripts using a variety of programming languages like Visual Basic, or .Net. Since end-users are not proficient in programming, they find it too difficult to write desired macros or scripts.

We have performed an extensive case study of spreadsheet help forums and identified that string processing is one of the most common class of programming problems that end-users struggle with. This is not surprising given that languages like Perl, Awk, Python came into existence to support string/text processing, and that new languages like Java/C# provide a rich support for string processing. During our study of help forums, we also carefully studied how these users were describing the specification of the desired program to the experts on the other side of the help forums. It turns out that the most common form of specification was input-output examples. Since input-output examples may lead to under-specification, the interaction between the user and the expert often involved a few rounds of communication (over multiple days).

We describe a program synthesis system that is capable of synthesizing a wide range of string processing programs in spreadsheets from input-output examples. The synthesizer aims to replace the role of the forum expert, which not only removes a human from the loop, but also enables users to solve their problems in a few seconds as opposed to a few days. Our synthesis system, which is deployment ready, has the following important usability properties.

- Fully Automated: We do not require non-sophisticated end-users to provide annotations/hints of any form.
- Real Time: Our system takes less than 0.1 second on average per interactive round.
- Easy Interaction: Programming by examples is an interactive process where examples are added in each round to make the specification more precise. Our system helps identify the inputs for which the user should provide examples.
- Fast Convergence: Our system typically takes 1-4 rounds of iteration for convergence in practice.
- Noise Handling: If the user makes a small mistake in mostly correct specification, our system can still compute the likely solution and report the likely mistake.

This paper makes the following contributions.

1. We describe a string programming/expression language that is expressive enough to represent a wide variety of string manipulation tasks found during an extensive study of Excel online help

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PoPL'11, January 26–28, 2011, Austin, Texas, USA.

Copyright © 2011 ACM 978-1-4503-0490-0/11/01…$10.00

317

Context: FlashFill

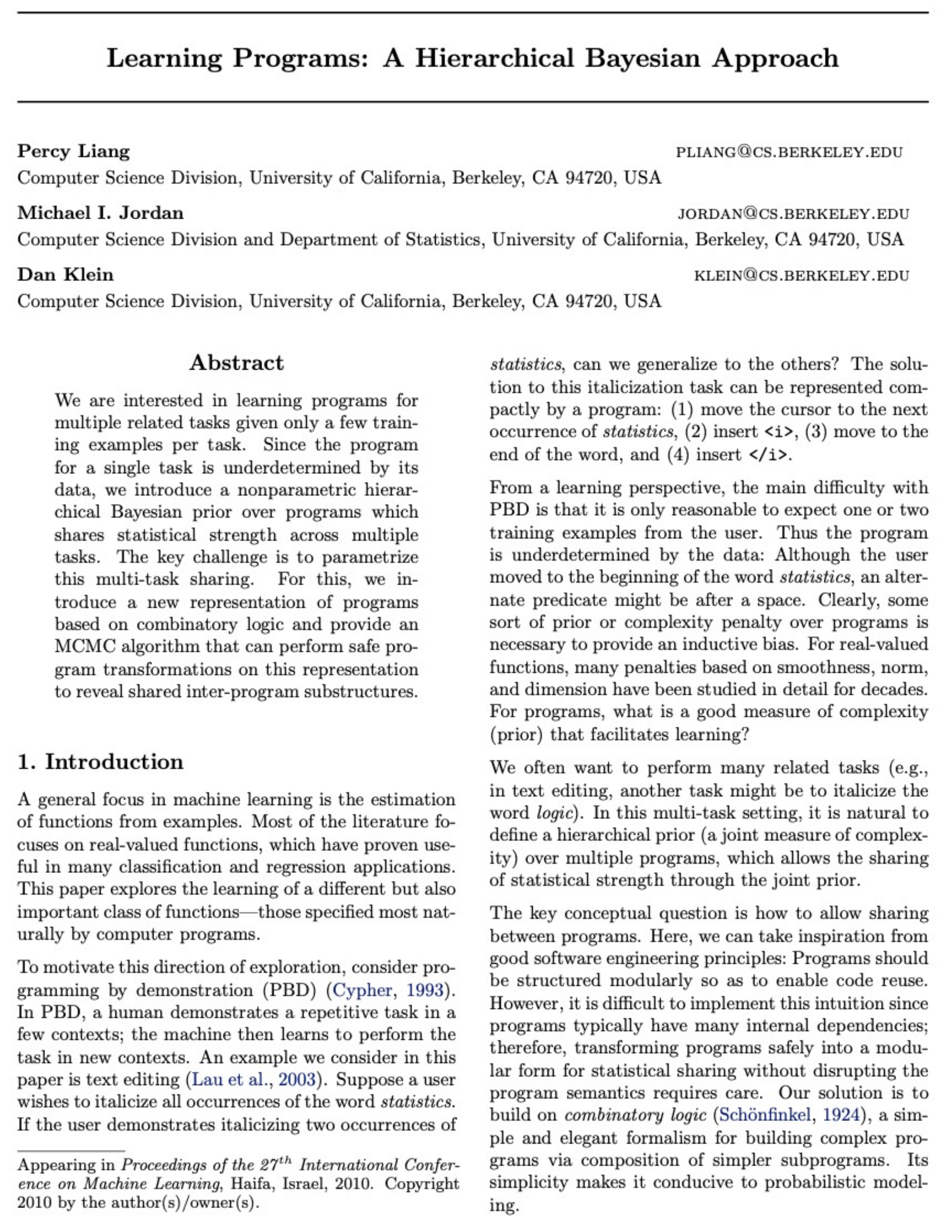
- Appeared: POPL 2011
- Test of Time Award: 2021

Why?

- Spawned several other tools: FlashRelate, FlashExtract, other programming by example tools
- Ushered in practical program synthesis
- Actually implemented in end-user software

Where are they now?

- Why did FlashFill take off?
- Why don't we use PADS?



Learning Programs: A Hierarchical Bayesian Approach

Percy Liang PLIANG@CS.BERKELEY.EDU
Computer Science Division, University of California, Berkeley, CA 94720, USA

Michael I. Jordan JORDAN@CS.BERKELEY.EDU
Computer Science Division and Department of Statistics, University of California, Berkeley, CA 94720, USA

Dan Klein KLEIN@CS.BERKELEY.EDU
Computer Science Division, University of California, Berkeley, CA 94720, USA

Abstract

We are interested in learning programs for multiple related tasks given only a few training examples per task. Since the program for a single task is underdetermined by its data, we introduce a nonparametric hierarchical Bayesian prior over programs which shares statistical strength across multiple tasks. The key challenge is to parametrize this multi-task sharing. For this, we introduce a new representation of programs based on combinatory logic and provide an MCMC algorithm that can perform safe program transformations on this representation to reveal shared inter-program substructures.

1. Introduction

A general focus in machine learning is the estimation of functions from examples. Most of the literature focuses on real-valued functions, which have proven useful in many classification and regression applications. This paper explores the learning of a different but also important class of functions—those specified most naturally by computer programs.

To motivate this direction of exploration, consider programming by demonstration (PBD) (Cypher, 1993). In PBD, a human demonstrates a repetitive task in a few contexts; the machine then learns to perform the task in new contexts. An example we consider in this paper is text editing (Lau et al., 2003). Suppose a user wishes to italicize all occurrences of the word *statistics*. If the user demonstrates italicizing two occurrences of *statistics*, can we generalize to the others? The solution to this italicization task can be represented compactly by a program: (1) move the cursor to the next occurrence of *statistics*, (2) insert `<i>`, (3) move to the end of the word, and (4) insert `</i>`.

From a learning perspective, the main difficulty with PBD is that it is only reasonable to expect one or two training examples from the user. Thus the program is underdetermined by the data: Although the user moved to the beginning of the word *statistics*, an alternate predicate might be after a space. Clearly, some sort of prior or complexity penalty over programs is necessary to provide an inductive bias. For real-valued functions, many penalties based on smoothness, norm, and dimension have been studied in detail for decades. For programs, what is a good measure of complexity (prior) that facilitates learning?

We often want to perform many related tasks (e.g., in text editing, another task might be to italicize the word *logic*). In this multi-task setting, it is natural to define a hierarchical prior (a joint measure of complexity) over multiple programs, which allows the sharing of statistical strength through the joint prior.

The key conceptual question is how to allow sharing between programs. Here, we can take inspiration from good software engineering principles: Programs should be structured modularly so as to enable code reuse. However, it is difficult to implement this intuition since programs typically have many internal dependencies; therefore, transforming programs safely into a modular form for statistical sharing without disrupting the program semantics requires care. Our solution is to build on *combinatory logic* (Schönfinkel, 1924), a simple and elegant formalism for building complex programs via composition of simpler subprograms. Its simplicity makes it conducive to probabilistic modeling.

Appearing in *Proceedings of the 27th International Conference on Machine Learning*, Haifa, Israel, 2010. Copyright 2010 by the author(s)/owner(s).