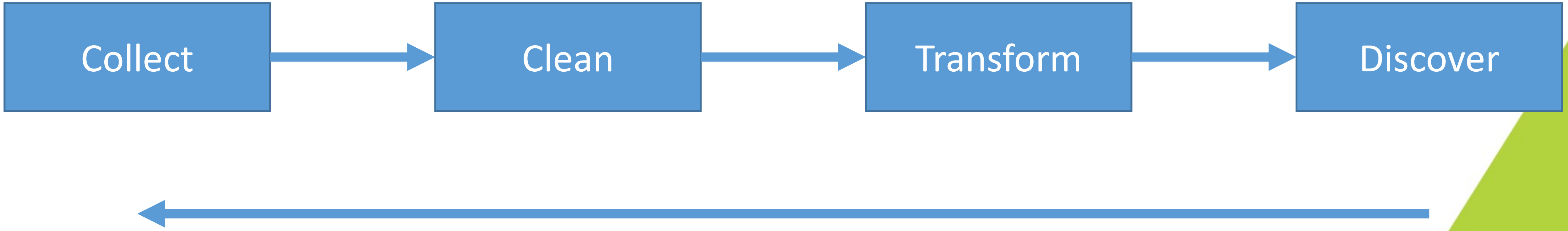
CS 295B/CS 395B Systems for Knowledge Discovery

Potpourri

The University of Vermont

Done with this part of the course



This week:

- Return to PADS in the context of KDD
- Cross current: randomness + fuzz testing

Today

- Start with fuzz testing paper
  - Testing: not previously discussed in class
- PADS redux (bigger context): may not finish discussion today
  - Some new notation
  - Try to read at the high level, understand why things are formalized
    - What the formalization is doing/why it matters
    - Don't worry about understanding how it works
  - Consider: how is the paper different from the previous PADS paper?

Context: Seed Selection for Successful Fuzzing

Theme thus far:

- Using systems for knowledge discovery
  - Many systems have nice properties, e.g.,
    - formal languages that are "correct by construction"
    - tools that automate manual processes
    - tools with statistical guarantees

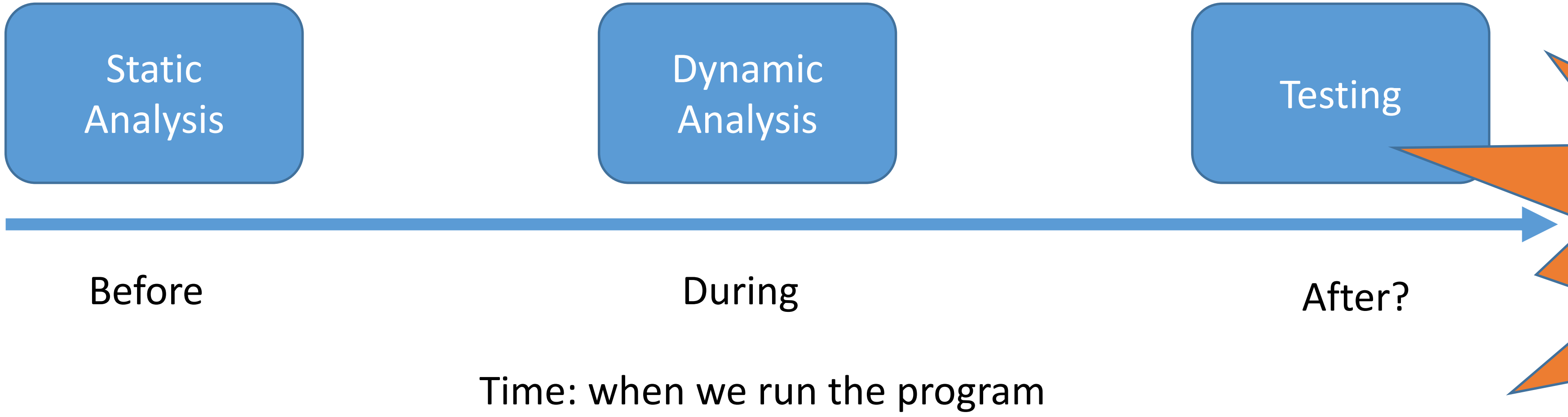
All these properties are over static programs

Verification

"Does the software do what we built it to do?"

Note: "software," not "program" – could be over a whole framework!

Spoiler: can test at many points



Techniques are orthogonal

Draw diagram on board

Beizer's Levels of Software Testing

Level 0 – No difference between testing and debugging

Level 1 – The purpose of testing is to show correctness

Level 2 – The purpose of testing is to show that the software does not work

Level 3 – The purpose of testing is not to prove anything specific, but to reduce the risk of using the software

Level 4 – Testing is a mental discipline that helps all IT professionals develop higher-quality software

(from Ammann & Offutt's *Introduction to Software Testing*, 2nd edition)

Ammann & Offut's Testing Levels

Integration – w.r.t. subsystem design — Program

Module – w.r.t. detailed design — Class, file, module, etc. (independent logical component)

Unit – w.r.t. implementation — Function or method

(from Ammann & Offutt's *Introduction to Software Testing,* 2nd edition)

Ammann & Offut's Testing Levels

Acceptance – w.r.t. requirements or users' needs

Human component
(requires requirements, notion of client)

System – w.r.t. architectural design and overall *behavior*

Integration – w.r.t. subsystem design

Module – w.r.t. detailed design

Unit – w.r.t. implementation

Framework
(temporal component)

(from Ammann & Offutt's *Introduction to Software Testing*, 2nd edition)

Ammann & Offut's Testing Levels

Acceptance – w.r.t. requirements or user' needs

System – w.r.t. architectural design and o

Integration – w.r.t. subsystem desig

Module – w.r.t. detailed design

Unit – w.r.t. implementation

(from Ammann & Offutt's *Introduction to Software Testing*, 2nd

Many testing approaches

Today: focus on *fuzzing*

Applied to the familiar levels

Background: Test suite generation

Worst case: Cartesian product of domain

"Inputs" can mean many things:

- **Inputs to** or **parameters of** to a function
  - Numbers, strings, structs, instances of other abstract data types, etc.
  - Not very complex, can easily reason over whole domain or equivalence classes

Aside: the simplicity is a lie

How you generate inputs matters

Think: testing image processing over 512x512 pixel RGBA images

- Many images will be nonsense
- i.e., only a small subset of inputs is actually meaningful

This is a huge problem in machine learning

Background: Test suite generation

"Inputs" can mean many things:

- **Inputs to** or **parameters of** to a function
    - Numbers, strings, structs, instances of other abstract data types, etc.
    - Not very complex, can easily reason over whole domain or equivalence classes
- Also: **whole programs**
    - Think: testing a compiler
- Higher testing levels, more complex inputs (+ larger input space)

Worst case:
exponential in domains
of each subcomponent

Fuzzing: Challenges and Reflections

Marcel Böhme, Monash University

Cristian Cadar, Imperial College London

Abhik Roychoudhury, National University of Singapore

// We summarize the open challenges and opportunities for fuzzing and symbolic execution as they emerged in discussions among researchers and practitioners in a Shonan Meeting and that were validated in a subsequent survey. //

THE INTERNET AND the world's Digital Economy run on a shared, critical open source software infrastructure. A security flaw in a single library can have severe consequences. For instance, OpenSSL implements protocols for secure communication and is widely used by Internet servers, including the majority of HTTPS websites. The Heartbleed vulnerability in an earlier version of OpenSSL would leak secret data and caused huge financial losses. It is important for us to develop practical and effective techniques to discover vulnerabilities automatically and at scale. Today, *fuzzing* is one of the most promising techniques in this regard. Fuzzing is an automatic bug and vulnerability discovery technique that continuously generates inputs and reports those that crash the program. There are three main categories of fuzzing tools and techniques: black-, gray-, and white-box fuzzing.

Black-box fuzzing generates inputs without any knowledge of the program. There are two main variants of black-box fuzzing: mutational and generational. In mutational black-box fuzzing, the fuzz campaign starts with one or more seed inputs. These seeds are modified to generate new inputs. Random mutations are applied to random locations in the input. For instance, a file fuzzer may flip random bits in a seed file. The process continues until a time budget is exhausted. In generational black-box fuzzing, inputs are generated from scratch. If a structural specification of the input format is provided, new inputs are generated that meet the grammar. Peach (http://community.peachfuzzer.com) is one popular black-box fuzzer.

Gray-box fuzzing leverages program instrumentation to get lightweight feedback, which is used to steer the fuzzer. Typically, a few control locations in the program are instrumented at the compile time and an initial seed corpus is provided. Seed inputs are mutated to generate new inputs. Generated inputs that cover new control locations and, thus, increase code coverage are added to the seed corpus. The coverage feedback allows a gray-box fuzzer to gradually reach deeper into the code. To identify bugs and vulnerabilities, *sanitizers* inject assertions into the program. Existing gray-box fuzzing tools include American fuzzy lop (AFL) (https://lcamtuf.coredump.cx/afl/), LibFuzzer (https://llvm.org/docs/LibFuzzer.html), and Honggfuzz (https://github.com/google/honggfuzz).

White-box fuzzing is based on a technique called *symbolic execution*,[1] which uses program analysis and constraint solvers to systematically enumerate interesting program paths. The constraint solvers used as the back end in white-box fuzzing are Satisfiability Modulo

Digital Object Identifier 10.1109/MS.2020.3016773
Date of current version: 13 August 2020

Background: fuzzing

Basic concept: generate random inputs *to some part of the program*

Can partition research depending on *where:*

Black-box – one point of eligible input

White-box – path-based, given knowledge of program structure

Grey-box – instrumentation-based, given access to program points

FOCUS: ON SOFTWARE QUALITY

Fuzzing: Challenges and Reflections

Marcel Böhme, Monash University

Cristian Cadar, Imperial College London

Abhik Roychoudhury, National University of Singapore

// We summarize the open challenges and opportunities for fuzzing and symbolic execution as they emerged in discussions among researchers and practitioners in a Shonan Meeting and that were validated in a subsequent survey. //

THE INTERNET AND the world's Digital Economy run on a shared, critical open source software infrastructure. A security flaw in a single library can have severe consequences. For instance, OpenSSL implements protocols for secure communication and is widely used by Internet servers, including the majority of HTTPS websites. The Heartbleed vulnerability in an earlier version of OpenSSL would leak secret data and caused huge financial losses. It is important for us to develop practical and effective techniques to discover vulnerabilities automatically and at scale. Today, *fuzzing* is one of the most promising techniques in this regard. Fuzzing is an automatic bug and vulnerability discovery technique that continuously generates inputs and reports those that crash the program. There are three main categories of fuzzing tools and techniques: black-, gray-, and white-box fuzzing.

Black-box fuzzing generates inputs without any knowledge of the program. There are two main variants of black-box fuzzing: mutational and generational. In mutational black-box fuzzing, the fuzz campaign starts with one or more seed inputs. These seeds are modified to generate new inputs. Random mutations are applied to random locations in the input. For instance, a file fuzzer may flip random bits in a seed file. The process continues until a time budget is exhausted. In generational black-box fuzzing, inputs are generated from scratch. If a structural specification of the input format is provided, new inputs are generated that meet the grammar. Peach (http://community.peachfuzzer.com) is one popular black-box fuzzer.

Gray-box fuzzing leverages program instrumentation to get lightweight feedback, which is used to steer the fuzzer. Typically, a few control locations in the program are instrumented at the compile time and an initial seed corpus is provided. Seed inputs are mutated to generate new inputs. Generated inputs that cover new control locations and, thus, increase code coverage are added to the seed corpus. The coverage feedback allows a gray-box fuzzer to gradually reach deeper into the code. To identify bugs and vulnerabilities, *sanitizers* inject assertions into the program. Existing gray-box fuzzing tools include American fuzzy lop (AFL) (https://lcamtuf.coredump.cx/afl/), LibFuzzer (https://llvm.org/docs/LibFuzzer.html), and Honggfuzz (https://github.com/google/honggfuzz).

White-box fuzzing is based on a technique called *symbolic execution*,[1] which uses program analysis and constraint solvers to systematically enumerate interesting program paths. The constraint solvers used as the back end in white-box fuzzing are Satisfiability Modulo

Digital Object Identifier 10.1109/MS.2020.3016773
Date of current version: 13 August 2020

Background: fuzzing

Basic concept: generate random inputs *to some part of the program*

(Classically, based on what you *have*):

Black-box – one point of eligible input

White-box – path-based, given knowledge of program structure

Grey-box – instrumentation-based, given access to program points


FOCUS: ON SOFTWARE QUALITY

Fuzzing: Challenges and Reflections

Marcel Böhme, Monash University

Cristian Cadar, Imperial College London

Abhik Roychoudhury, National University of Singapore

// We summarize the open challenges and opportunities for fuzzing and symbolic execution as they emerged in discussions among researchers and practitioners in a Shonan Meeting and that were validated in a subsequent survey. //

THE INTERNET AND the world's Digital Economy run on a shared, critical open source software infrastructure. A security flaw in a single library can have severe consequences. For instance, OpenSSL implements protocols for secure communication and is widely used by Internet servers, including the majority of HTTPS websites. The Heartbleed vulnerability in an earlier version of OpenSSL would leak secret data and caused huge financial losses. It is important for us to develop practical and effective techniques to discover vulnerabilities automatically and at scale. Today, *fuzzing* is one of the most promising techniques in this regard. Fuzzing is an automatic bug and vulnerability discovery technique that continuously generates inputs and reports those that crash the program. There are three main categories of fuzzing tools and techniques: black-, gray-, and white-box fuzzing.

Black-box fuzzing generates inputs without any knowledge of the program. There are two main variants of black-box fuzzing: mutational and generational. In mutational black-box fuzzing, the fuzz campaign starts with one or more seed inputs. These seeds are modified to generate new inputs. Random mutations are applied to random locations in the input. For instance, a file fuzzer may flip random bits in a seed file. The process continues until a time budget is exhausted. In generational black-box fuzzing, inputs are generated from scratch. If a structural specification of the input format is provided, new inputs are generated that meet the grammar. Peach (http://community.peachfuzzer.com) is one popular black-box fuzzer.

Gray-box fuzzing leverages program instrumentation to get lightweight feedback, which is used to steer the fuzzer. Typically, a few control locations in the program are instrumented at the compile time and an initial seed corpus is provided. Seed inputs are mutated to generate new inputs. Generated inputs that cover new control locations and, thus, increase code coverage are added to the seed corpus. The coverage feedback allows a gray-box fuzzer to gradually reach deeper into the code. To identify bugs and vulnerabilities, *sanitizers* inject assertions into the program. Existing gray-box fuzzing tools include American fuzzy lop (AFL) (https://lcamtuf.coredump.cx/afl/), LibFuzzer (https://llvm.org/docs/LibFuzzer.html), and Honggfuzz (https://github.com/google/honggfuzz).

White-box fuzzing is based on a technique called *symbolic execution*,[1] which uses program analysis and constraint solvers to systematically enumerate interesting program paths. The constraint solvers used as the back end in white-box fuzzing are Satisfiability Modulo

Digital Object Identifier 10.1109/MS.2020.3016773
Date of current version: 13 August 2020

Background: fuzzing

Basic concept: generate random inputs *to some part of the program*

(Classically, based on what you *have*):

Black-box – no code access, execute only

White-box – path-based, given knowledge of program structure

Grey-box – instrumentation-based, given access to program points

FOCUS: ON SOFTWARE QUALITY

Fuzzing: Challenges and Reflections

Marcel Böhme, Monash University

Cristian Cadar, Imperial College London

Abhik Roychoudhury, National University of Singapore

// *We summarize the open challenges and opportunities for fuzzing and symbolic execution as they emerged in discussions among researchers and practitioners in a Shonan Meeting and that were validated in a subsequent survey.* //

THE INTERNET AND the world's Digital Economy run on a shared, critical open source software infrastructure. A security flaw in a single library can have severe consequences. For instance, OpenSSL implements protocols for secure communication and is widely used by Internet servers, including the majority of HTTPS websites. The Heartbleed vulnerability in an earlier version of OpenSSL would leak secret data and caused huge financial losses. It is important for us to develop practical and effective techniques to discover vulnerabilities automatically and at scale. Today, *fuzzing* is one of the most promising techniques in this regard. Fuzzing is an automatic bug and vulnerability discovery technique that continuously generates inputs and reports those that crash the program. There are three main categories of fuzzing tools and techniques: black-, gray-, and white-box fuzzing.

Black-box fuzzing generates inputs without any knowledge of the program. There are two main variants of black-box fuzzing: mutational and generational. In mutational black-box fuzzing, the fuzz campaign starts with one or more seed inputs. These seeds are modified to generate new inputs. Random mutations are applied to random locations in the input. For instance, a file fuzzer may flip random bits in a seed file. The process continues until a time budget is exhausted. In generational black-box fuzzing, inputs are generated from scratch. If a structural specification of the input format is provided, new inputs are generated that meet the grammar. Peach (http://community.peachfuzzer.com) is one popular black-box fuzzer.

Gray-box fuzzing leverages program instrumentation to get lightweight feedback, which is used to steer the fuzzer. Typically, a few control locations in the program are instrumented at the compile time and an initial seed corpus is provided. Seed inputs are mutated to generate new inputs. Generated inputs that cover new control locations and, thus, increase code coverage are added to the seed corpus. The coverage feedback allows a gray-box fuzzer to gradually reach deeper into the code. To identify bugs and vulnerabilities, *sanitizers* inject assertions into the program. Existing gray-box fuzzing tools include American fuzzy lop (AFL) (https://lcamtuf.coredump.cx/afl/), LibFuzzer (https://llvm.org/docs/LibFuzzer.html), and Honggfuzz (https://github.com/google/honggfuzz).

White-box fuzzing is based on a technique called *symbolic execution*,[1] which uses program analysis and constraint solvers to systematically enumerate interesting program paths. The constraint solvers used as the back end in white-box fuzzing are Satisfiability Modulo

Digital Object Identifier 10.1109/MS.2020.3016773
Date of current version: 13 August 2020

Background: fuzzing

Basic concept: generate random inputs *to some part of the program*

(Classically, based on what you *have*):

Black-box – no code access, execute only

White-box – access to complete source code

Grey-box – instrumentation-based, given access to program points

FOCUS: ON SOFTWARE QUALITY

Fuzzing: Challenges and Reflections

Marcel Böhme, Monash University

Cristian Cadar, Imperial College London

Abhik Roychoudhury, National University of Singapore

// *We summarize the open challenges and opportunities for fuzzing and symbolic execution as they emerged in discussions among researchers and practitioners in a Shonan Meeting and that were validated in a subsequent survey.* //

THE INTERNET AND the world's Digital Economy run on a shared, critical open source software infrastructure. A security flaw in a single library can have severe consequences. For instance, OpenSSL implements protocols for secure communication and is widely used by Internet servers, including the majority of HTTPS websites. The Heartbleed vulnerability in an earlier version of OpenSSL would leak secret data and caused huge financial losses. It is important for us to develop practical and effective techniques to discover vulnerabilities automatically and at scale. Today, *fuzzing* is one of the most promising techniques in this regard. Fuzzing is an automatic bug and vulnerability discovery technique that continuously generates inputs and reports those that crash the program. There are three main categories of fuzzing tools and techniques: black-, gray-, and white-box fuzzing.

Black-box fuzzing generates inputs without any knowledge of the program. There are two main variants of black-box fuzzing: mutational and generational. In mutational black-box fuzzing, the fuzz campaign starts with one or more seed inputs. These seeds are modified to generate new inputs. Random mutations are applied to random locations in the input. For instance, a file fuzzer may flip random bits in a seed file. The process continues until a time budget is exhausted. In generational black-box fuzzing, inputs are generated from scratch. If a structural specification of the input format is provided, new inputs are generated that meet the grammar. Peach (http://community.peachfuzzer.com) is one popular black-box fuzzer.

Gray-box fuzzing leverages program instrumentation to get lightweight feedback, which is used to steer the fuzzer. Typically, a few control locations in the program are instrumented at the compile time and an initial seed corpus is provided. Seed inputs are mutated to generate new inputs. Generated inputs that cover new control locations and, thus, increase code coverage are added to the seed corpus. The coverage feedback allows a gray-box fuzzer to gradually reach deeper into the code. To identify bugs and vulnerabilities, *sanitizers* inject assertions into the program. Existing gray-box fuzzing tools include American fuzzy lop (AFL) (https://lcamtuf.coredump.cx/afl/), LibFuzzer (https://llvm.org/docs/LibFuzzer.html), and Honggfuzz (https://github.com/google/honggfuzz).

White-box fuzzing is based on a technique called *symbolic execution*,[1] which uses program analysis and constraint solvers to systematically enumerate interesting program paths. The constraint solvers used as the back end in white-box fuzzing are Satisfiability Modulo

Digital Object Identifier 10.1109/MS.2020.3016773
Date of current version: 13 August 2020

Background: fuzzing

Basic concept: generate random inputs *to some part of the program*

(Classically, based on what you *have*):

Black-box – no code access, execute only

White-box – access to complete source code

Grey-box – partial access to code (e.g., compiled code or binaries)

Example: Reinforcement Learning Environments

Task: test whether an agent has learned to play PacMan

- **Black box** – just interact via controller
- **Grey box** – Arcade Learning Environment (ALE)
- **White box** -- Toybox


FOCUS: ON SOFTWARE QUALITY

Fuzzing: Challenges and Reflections

Marcel Böhme, Monash University

Cristian Cadar, Imperial College London

Abhik Roychoudhury, National University of Singapore

// We summarize the open challenges and opportunities for fuzzing and symbolic execution as they emerged in discussions among researchers and practitioners in a Shonan Meeting and that were validated in a subsequent survey. //

THE INTERNET AND the world's Digital Economy run on a shared, critical open source software infrastructure. A security flaw in a single library can have severe consequences. For instance, OpenSSL implements protocols for secure communication and is widely used by Internet servers, including the majority of HTTPS websites. The Heartbleed vulnerability in an earlier version of OpenSSL would leak secret data and caused huge financial losses. It is important for us to develop practical and effective techniques to discover vulnerabilities automatically and at scale. Today, *fuzzing* is one of the most promising techniques in this regard. Fuzzing is an automatic bug and vulnerability discovery technique that continuously generates inputs and reports those that crash the program. There are three main categories of fuzzing tools and techniques: black-, gray-, and white-box fuzzing.

Black-box fuzzing generates inputs without any knowledge of the program. There are two main variants of black-box fuzzing: mutational and generational. In mutational black-box fuzzing, the fuzz campaign starts with one or more seed inputs. These seeds are modified to generate new inputs. Random mutations are applied to random locations in the input. For instance, a file fuzzer may flip random bits in a seed file. The process continues until a time budget is exhausted. In generational black-box fuzzing, inputs are generated from scratch. If a structural specification of the input format is provided, new inputs are generated that meet the grammar. Peach (http://community.peachfuzzer.com) is one popular black-box fuzzer.

Gray-box fuzzing leverages program instrumentation to get lightweight feedback, which is used to steer the fuzzer. Typically, a few control locations in the program are instrumented at the compile time and an initial seed corpus is provided. Seed inputs are mutated to generate new inputs. Generated inputs that cover new control locations and, thus, increase code coverage are added to the seed corpus. The coverage feedback allows a gray-box fuzzer to gradually reach deeper into the code. To identify bugs and vulnerabilities, *sanitizers* inject assertions into the program. Existing gray-box fuzzing tools include American fuzzy lop (AFL) (https://lcamtuf.coredump.cx/afl/), LibFuzzer (https://llvm.org/docs/LibFuzzer.html), and Honggfuzz (https://github.com/google/honggfuzz).

White-box fuzzing is based on a technique called *symbolic execution*,[1] which uses program analysis and constraint solvers to systematically enumerate interesting program paths. The constraint solvers used as the back end in white-box fuzzing are Satisfiability Modulo

Digital Object Identifier 10.1109/MS.2020.3016773
Date of current version: 13 August 2020

Mutation vs. Generation

Two major ways to generate inputs:

Local: mutation

- Start with a representative program
- Make a random change in a systematic way
- (seen recently in PlanAlyzer paper)

Global: generation

- Use specification (e.g., BNF, protocol, etc.) to randomly generate
- Often uses model-based approaches

So Random

Cannot take randomness for granted

- Recurring problem in computing
- About uniformity, not true randomness:
    - Want: uniform random selection over some domain
    - Have: the ability to draw from some function of that domain
    - Need: a better understanding of the mapping

NOT about cryptographically secure RNGs (different problem)

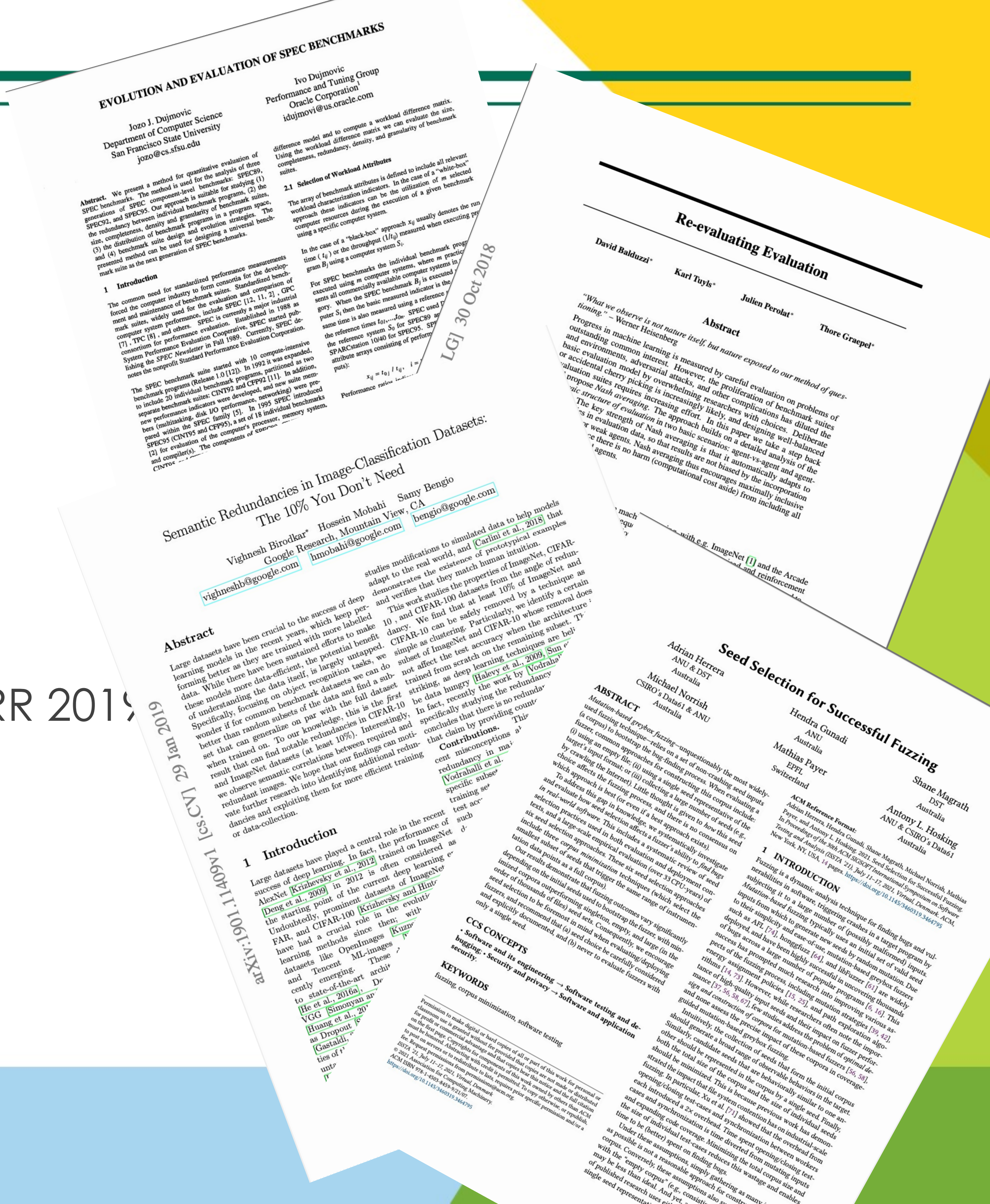
Examples

Mostly see this in criticism of **benchmarks**:

- HPC – SPEC benchmark (SIGMETRICS1998)
- Reinforcement learning – ALE (NeurIPS 2018)
- Computer Vision – CIFAR-10 & ImageNet (CoRR 2019)
- Software Testing – seed selection (ISSTA 2021)

Q: Why does this keep happening?

A: *Lifecycle of research*



Food for thought

- Connections between software testing and experimentation
- Software testing as knowledge discovery for software?
- Is this test case an edge case or a representative of a larger class?
    - Connections to models (simplified views of the world)
    - Connections to machine learning
    - Connections to causality
- **Methodology of testing** vs. **testing of methodology** vs. **testing as methodology**
- Role of randomness


The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework. This framework dictates the rules about the uses of user-coined names, and the conventions about characterizing functional relationships. Within this framework the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a nonprocedural (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side expressions easy to construct and understand.

1. Introduction

Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things. The ISWIM (If you See What I Mean) system is a byproduct of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncracies are concerned with the former rather than the latter, whereas aptitude for a particular class of tasks is essentially determined by the latter rather than the former. The conclusion follows that many language characteristics are irrelevant to the alleged problem orientation.

ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. The possibilities concerning this set and what is needed to specify such a set are discussed below.

ISWIM is not alone in being a family, even after mere syntactic variations have been discounted (see Section 4). In practice, this is true of most languages that achieve more than one implementation, and if the dialects are well disciplined, they might with luck be characterized as differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have fielded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. (Note that restrictions on what names can be coined also vary, but these are trivial differences. When they have any logical significance it is likely to be pernicious, by leading to puns such as ALGOL's integer labels.)

So rules about user-coined names is an area in which we might expect to see the history of computer applications give ground to their logic. Another such area is in specifying functional relations. In fact these two areas are closely related since any use of a user-coined name implicitly involves a functional relation; e.g., compare

$x(x+a)$
where $x = b + 2c$

$f(b+2c)$
where $f(x) = x(x+a)$

ISWIM is thus part programming language and part program for research. A possible first step in the research program is 1700 doctoral theses called "A Correspondence between x and Church's λ-notation."[1]

2. The where-Notation

In ordinary mathematical communication, these uses of '**where**' require no explanation. Nor do the following:

$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$

$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$
and $b = u/(u+1)$
and $c = v/(v+1)$

$g(f$ **where** $f(x) = ax^2 + bx + c,$
$u/(u+1),$
$v/(v+1))$
where $g(f, p, q) = f(p+2q, 2p-q)$

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965.

[1] There is no more use or mention of λ in this paper—cognoscenti will nevertheless sense an undercurrent. A not inappropriate title would have been "Church without lambda."

Context: Next 700 Data Description Languages

Published in CACM in 1966

At the time:

- COBOL
- FORTRAN
- LISP
- ALGOL
- (notable: C not yet invented!)

"…today …1,700 special programming languages used to 'communicate' in over 700 application areas."

Computer Software Issues, an American Mathematical Association Prospectus, July 1965

"Most programming languages are partly a way of **expressing things in terms of other things** and partly a **basic set of given things**."

Context: Next 700 Data Description Languages

Published in CACM in 1966

At the time:

- COBOL
- FORTRAN
- LISP
- ALGOL
- (notable: C not yet invented!)

Languages differ in:

- “application area”
- “phase of computer use”
- physical appearance
- logical structure

Context: Next 700 Data Description Languages

Languages differ in:

- **“application area”**
- “phase of computer use”
- physical appearance
- logical structure

- business programming
  - then: COBOL
  - now: ~~COBOL~~ spreadsheets
- mathematical computing
  - then: FORTRAN
  - now: ~~FORTRAN via~~ Numpy
- algorithmic computing
  - then: ALGOL
  - now: general purpose lang. of your choice

- (notable: C not yet invented!)

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software Issues*, an American Mathematical Association Prospectus, July 1965.

Languages differ in:

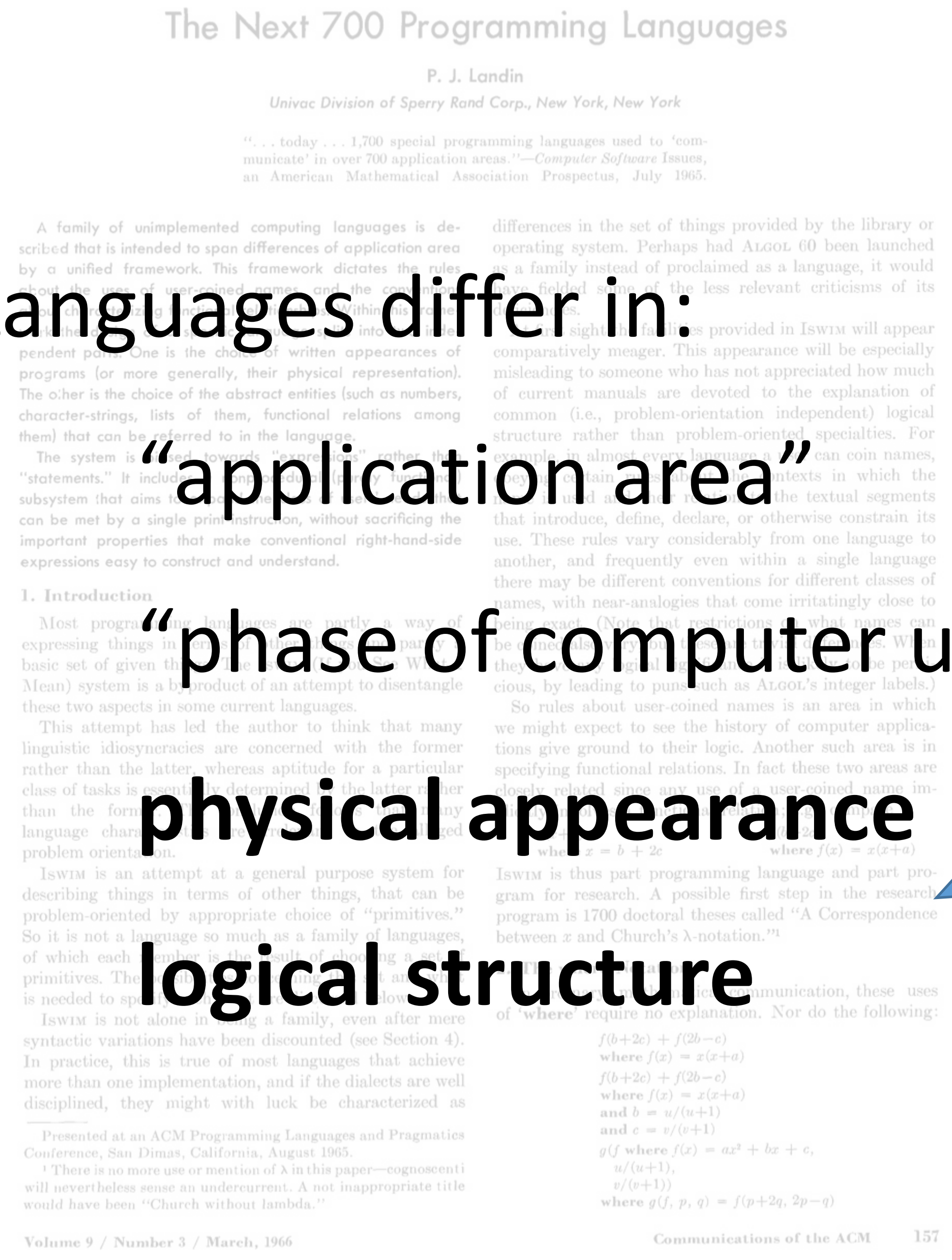
- "application area"
- **"phase of computer use"**
- physical appearance
- logical structure

Context: Next 700 Data Description Languages

"high-level programming, program assembly, job scheduling, etc."

Today: would separate into **end-user programming** vs. **intermediate or internal representation**

- (notable: C not yet invented!)



Languages differ in:

- "application area"
- "phase of computer use"
- **physical appearance**
- **logical structure**

Still true today!

Physical appearance: syntax

Logical structure: evaluation order (arguments, compiler passes, etc.)

Context: Next 700 Data Description Languages

- (notable: C not yet invented!)

Context: Next 700 Data Description Languages

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework. This framework dictates the rules about the uses of user-coined names, and the conventions about characterizing functional relationships. Within this framework the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a nonprocedural (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side expressions easy to construct and understand.

1. Introduction

Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things. The ISWIM (If you See What I Mean) system is a byproduct of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncracies are concerned with the former rather than the latter, whereas aptitude for a particular class of tasks is essentially determined by the latter rather than the former. The conclusion follows that many language characteristics are irrelevant to the alleged problem orientation.

ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. The possibilities concerning this set and what is needed to specify such a set are discussed below.

ISWIM is not alone in being a family, even after mere syntactic variations have been discounted (see Section 4). In practice, this is true of most languages that achieve more than one implementation, and if the dialects are well disciplined, they might with luck be characterized as differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have fielded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. (Note that restrictions on what names can be coined also vary, but these are trivial differences. When they have any logical significance it is likely to be pernicious, by leading to puns such as ALGOL's integer labels.)

So rules about user-coined names is an area in which we might expect to see the history of computer applications give ground to their logic. Another such area is in specifying functional relations. In fact these two areas are closely related since any use of a user-coined name implicitly involves a functional relation; e.g., compare

$x(x+a)$
where $x = b + 2c$

$f(b+2c)$
where $f(x) = x(x+a)$

ISWIM is thus part programming language and part program for research. A possible first step in the research program is 1700 doctoral theses called "A Correspondence between x and Church's λ-notation."[1]

2. The where-Notation

In ordinary mathematical communication, these uses of '**where**' require no explanation. Nor do the following:

$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$

$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$
and $b = u/(u+1)$
and $c = v/(v+1)$

$g(f$ **where** $f(x) = ax^2 + bx + c,$
$u/(u+1),$
$v/(v+1))$
where $g(f, p, q) = f(p+2q, 2p-q)$

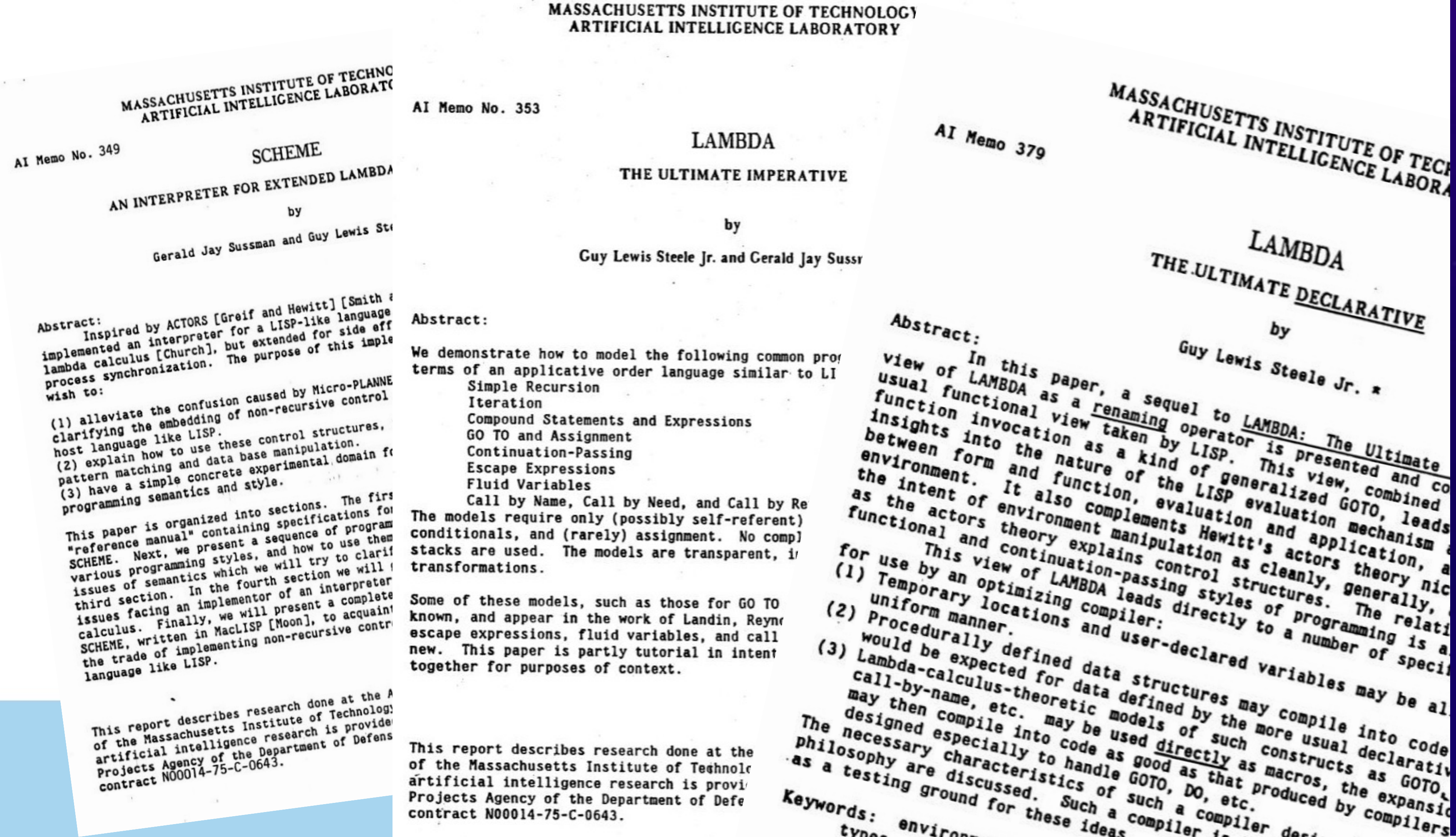
Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965.

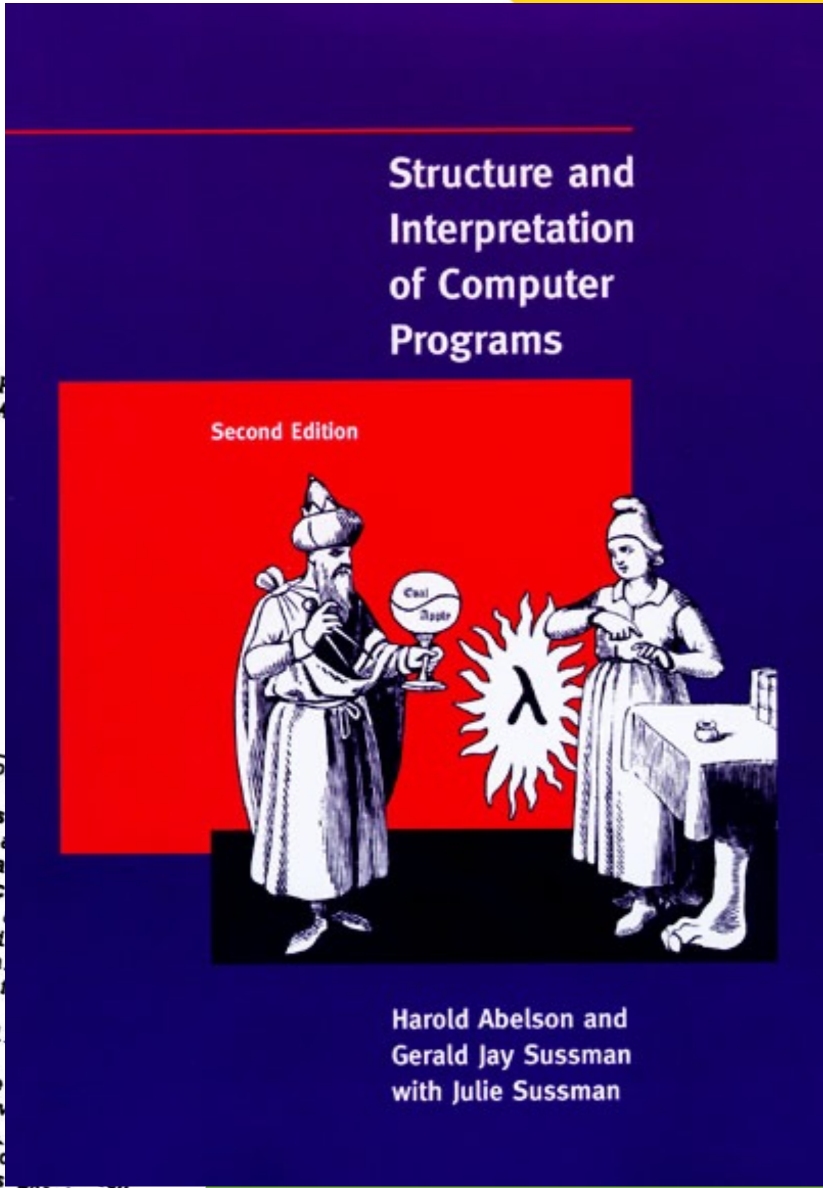
[1] There is no more use or mention of λ in this paper—cognoscenti will nevertheless sense an undercurrent. A not inappropriate title would have been "Church without lambda."

Broader context:

- Already have robust theory of computability
  - lambda calculus
  - Turing machines
  - von Neumann machines
- Attempt to refine understanding
  - Which things should be primitives?
  - What makes a language usable?
  - What constructs are most efficient?

Interlude: Lambda Papers





The Next 700 Programming Languages

P. J. Landin
Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages ... scribed that is intended to span differences of application ... by a unified framework. This framework dictates the ... about the uses of user-coined names, and the convent... about characterizing functional relationships. Within this fram... work the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system ... subsys... can be me... important prop... expressions easy to ...

...rences in the set of things provided by the library ... ystem. Perhaps had ALGOL 60 been launch... of proclaimed as a language, it w... less relevant criticisms ... SWIM wil...

1. Introduction

Most programming languag... expressing things in terms of othe... basic set of given things. The ISWIM (... Mean) system is a byprod... these two ...

... language ... problem orienta...

ISWIM is an attem... describing things in terms o... problem-oriented by appro... So it is not a language ... of which each mem... primitives. Th... is needed ...

Isw... su...

In practice, this is true ... more than one implementation, and if the dialects are disciplined, they might with luck be characterize...

Presented at an ACM Programming Languages and Pragm... Conference, San Dimas, California, August 1965.

[1] There is no more use or mention of λ in this paper—cognos... will nevertheless sense an undercurrent. A not inappropriate ... would have been "Church without lambda."

...$x+a$)
...$u+1$)
$v/(v+1)$
where $f(x) = ax^2 + bx + c$,
$u/(u+1)$,
$v/(v+1)$)
where $g(f, p, q) = f(p+2q, 2p-q)$

Notable:
~25 years before calcification of PL families

Another 10 years for data description languages?

Co... Next 700 Data ...n L... guages

...GOL

(notable: C not yet invented!)

Why return to PADS?

This paper: high-level calculus (previous: specific tool in the pipeline)

Data processing in KDD pipeline is still manual – still an important problem!

- Relation to course projects

Challenge: PADS tools hard for non-experts to use

- Papers are *not* for a data science audience
- …"data science" not coined for another two years