

CS 295B/CS 395B

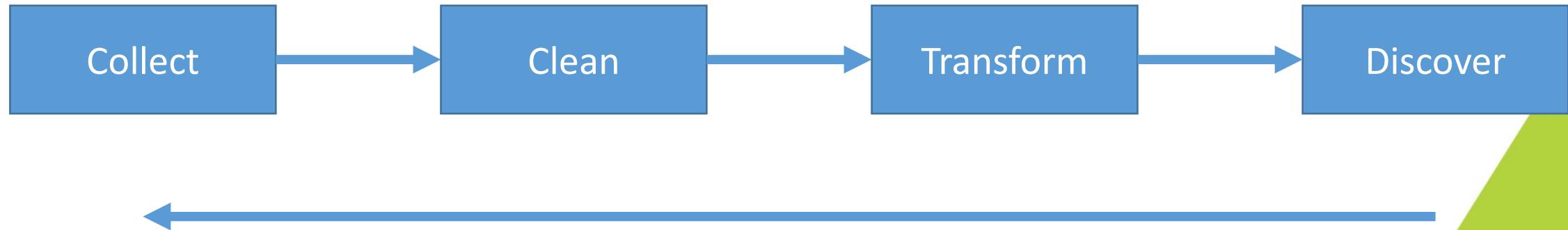
Systems for Knowledge Discovery

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

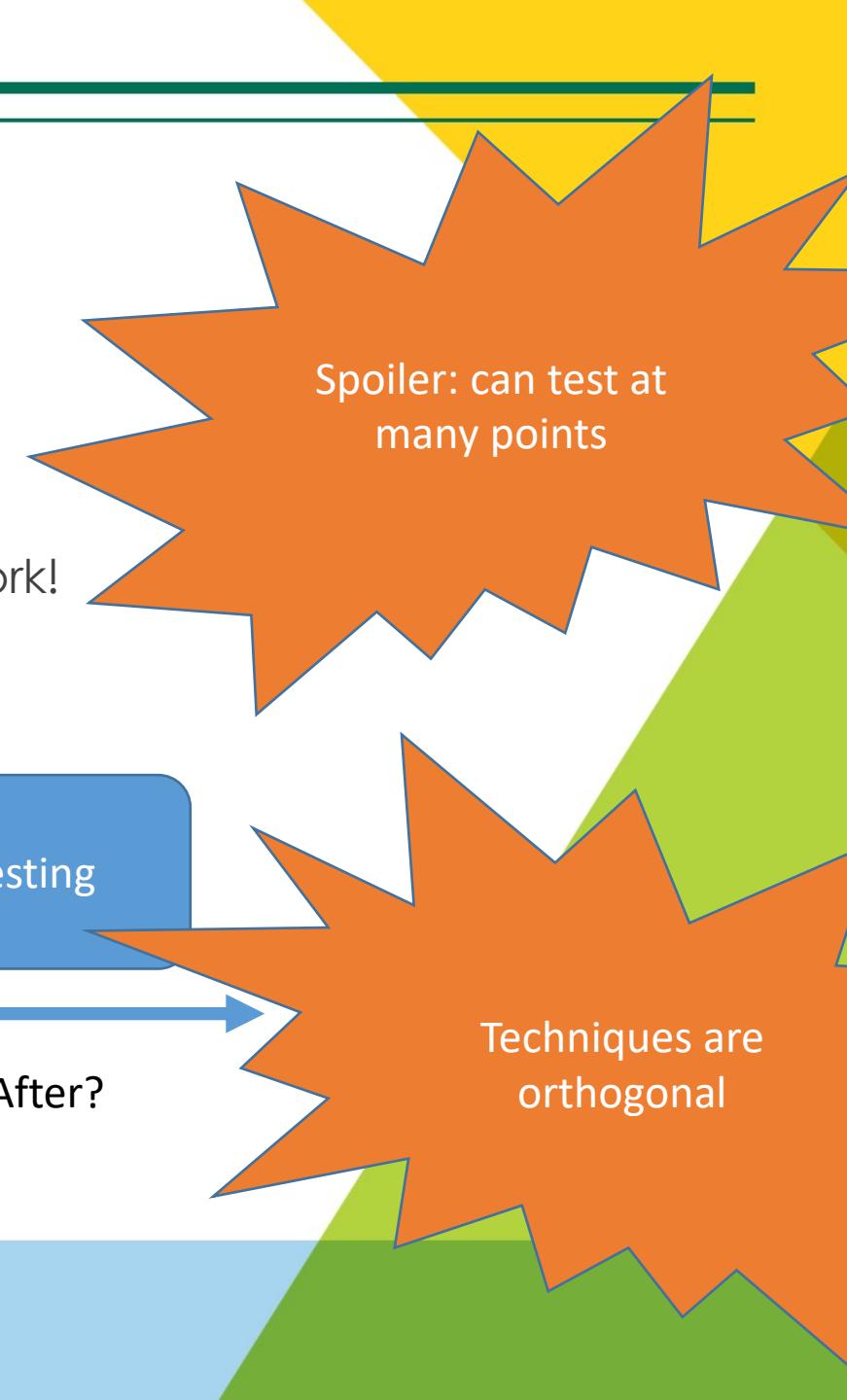
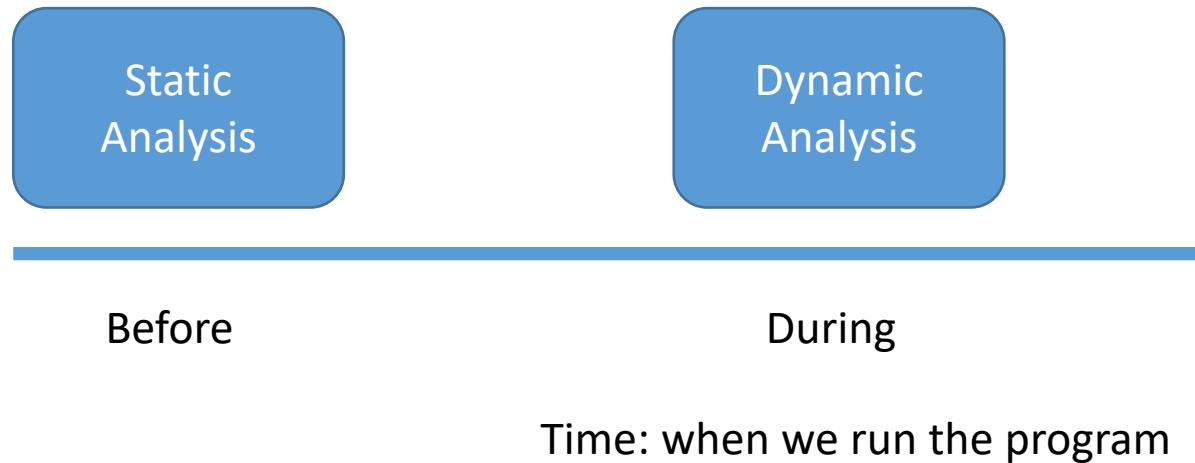


All these properties are over static programs

Verification

“Does the software do what we built it to do?”

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



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

Module – w.r.t. detailed design

Unit – w.r.t. implementation

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

Program

Class, file, module, etc.
(independent logical component)

Function or method

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

Framework
(temporal component)

Module – w.r.t. detailed design

Unit – w.r.t. implementation

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

Ammann & Offut's Testing Levels

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

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

Integration – w.r.t. subsystem design and interfaces

Module – w.r.t. detailed design

Unit – w.r.t. implementation

Many testing approaches

Today: focus on *fuzzing*

Applied to the familiar levels

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

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



Worst case: Cartesian product of domain

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 light-weight 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

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

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

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

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

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

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 light-weight 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

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

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

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

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 light-weight 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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see <https://creativecommons.org/licenses/by/4.0/deed.ast>.

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

EVOLUTION AND EVALUATION OF SPEC BENCHMARKS
Ivo Dujmovic
Performance and Tuning Group
Oracle Corporation¹
iivo@us.oracle.com

Jozo J. Dujmovic
Department of Computer Science
San Francisco State University
jozo@cs.sfsu.edu

Abstract. We present a method for quantitative evaluation of SPEC CPU benchmarks. The method is used for analysis of generations of SPEC CPU benchmarks: SPEC92, SPEC95, SPEC92, and SPEC95. Our approach is suitable for studying the relationship between individual benchmarks, (1) size, completeness and granularity of benchmarks notes, (2) the distribution of benchmark programs in a program space, (3) benchmark suite design and usage strategies, and (4) mark suits as the next generation of SPEC benchmarks.

The common need for standardization forced the computer industry to form a committee and maintain a benchmark suite, which was widely used for the evaluation of computer system performance. Examples include SPEC [12, 11], TPC [8] and others. SPEC is currently a major consortium for performance evaluation. Published in the System Performance Evaluation Cooperative, SPEC started the nonprofit Standard Performance Evaluation Corporation in Fall 1989. Currently, 10 computer suites started with 10 companies, partners

The SPEC benchmarks program (Release 1.0) includes 20 individual benchmark programs to measure performance in various application areas. The benchmarks are grouped into four categories: integer, floating-point, memory access, and disk I/O. The benchmarks are: SPFC95 (CINT95 and CFPP95), SPFC95 (CINT95 and CFPP95), and CINT95.

The workload difference matrix we can evaluate the size, redundancy, density, and granularity of benchmark

In the case of a "black-box" approach x_0 usually denotes the run time, the throughput ($1/t_0$) measured when executing P_0 on a given computer system S .

Individual benchmarks were practical in practice.

For SPEC benchmarks in memory system, the results based on commercially available computer systems are shown in Figure 1. In this figure, B is measured using the test bench. Measured value of B is converted to the reference value B_{ref} using the following equation:

$$B_{ref} = \frac{B}{\left(\frac{t_{ref}}{t}\right)^{\alpha}}$$

where t is the reference time. The reference time t_{ref} is 1000 ms for SPEC CPU2006 and 100 ms for SPEC CPU2006. The reference system is SPEC CPU2006. The reference array consists of perfect memory arrays.

Performance ratio is calculated as follows:

$$x_0 = \frac{B_{ref}}{B}$$

Image-Classification Datasets: A Need

Semantic Redundancies in Image Classification Datasets:
The 10% You Don't Need
Kush Baredar, Hossein Mobahi, Samy Bengio
<http://kushbaredar.com/research/MountainView.html>
bengio@google.com

vighneshb@google.com

have played a central role in fact, the period trained on ImageNet is considered as

Large datasets of raw learning data, such as AlexNet [Krizhevsky et al., 2012] or the setting presented in this paper, have had a crucial role in the evolution of learning methods like OpenImage [Kuznetsov et al., 2016].

KEYWORDS fuzzing, corpus mining, dataset, emerging, state-of-the-art, VGG, Simonyan and Huang et al., 2014; Dropout, GatedLSTM.

ties of
un-
P
the right to request or retransmit the item, though
ACM ISBN 978-1-4503-3903-9, 978-1-4503-3902-
<https://doi.org/10.1145/3390319.3394975>

*"What we observe is not *nature* itself, but *nature exposed to our method* of *questioning*"* – Werner Heisenberg

Progress in machine learning is measured by careful evaluation on problems of outstanding machine learning interest. However, complications due to different environments, adversarial attacks, and other challenges, diluted the basic evaluation model by overfitting well-known datasets.

Accidental cherry picking is increasingly likely, and this paper will take a step back to propose *Nash averaging*. The approach builds on a detailed analysis of agents' valuation suites. The key benefit of Nash averaging is that it automatically includes the structure of *redistribution*. The approach handles two basic scenarios: one where the incorporation of new agents is harm (computational cost aside) or where it is not.

The help models
2018 that
ruples

Julien Perolat*
Thore Graepel*

with e.g. ImageNet [1] and the Arcade game [2], and reinforcement learning [3].

Seed Selection for Success

and evaluate the gains in performance. This includes the selection of a target market, the choice of a marketing mix, and the analysis of the firm's strengths and weaknesses. The final step is to develop a marketing plan that outlines the specific actions to be taken to achieve the firm's goals.

the use of automated testing techniques for finding bugs. In this paper, we propose a technique for finding bugs in legacy systems. Triggering conditions in a target program are used to generate new test cases. Instead of randomizing the values of variables, our approach generates new test cases by randomizing the triggers. This allows us to find bugs in a large number of legacy systems. We have conducted experiments on a number of legacy systems, including Microsoft Office, Java, and Linux. The results show that our approach is effective in finding bugs in legacy systems. We also show that our approach is more efficient than randomization-based approaches. Our approach is based on the idea that bugs are often triggered by specific triggers. By randomizing the triggers, we can increase the chances of triggering a bug. This is particularly useful for legacy systems, which often have complex and non-intuitive trigger conditions. Our approach is also more efficient than randomization-based approaches because it only needs to generate a small number of test cases. This makes it suitable for use in real-world scenarios where resources are limited.

In part of this work for personal or professional use not made or distributed without the written consent of AGCO. All rights reserved.

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 idiosyncrasies 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

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."

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

$$\begin{array}{ll} x(x+a) & f(b+2c) \\ \text{where } x = b + 2c & \text{where } f(x) = x(x+a) \end{array}$$

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

Context: Next 700 Data Description Languages

Published in CACM in 1966

At the time:

- COBOL
- FORTRAN
- LISP
- ALGOL
- (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.

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 character-string representations; and in this framework the design of a dialect language is independent of its dependencies. One is the choice of system appendices 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) style of language to avoid the loss of expressiveness that would be met by a selection instruction, yet it allows the important phenomena that make it more convenient to read and write expressions easy to contact 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 at seeing these two aspects in some different light.

This attempt has led to a failure to think that many linguistic idiosyncrasies 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 linguistic peculiarities are irrelevant to the alleged problem orientation.

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

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."

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 severe deficiencies 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, observe certain rules about the contexts in which they are used and their relation to other user-defined names, and reduce, define, specialize, or otherwise constrain its use. These rules are generally common enough 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 an effect, however, it is likely to be pernicious, by leading to such as ALGOL interferences.)

So rules about user-coined names is an area in which we might expect to find the ability of common applications give ground to their logic. Another such area is 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 $f(b+2c)$ + $f(2b-c)$ where $f(x) = x(x+a)$ with $f(b+2c)$ + $f(2b-c)$ where $f(x) = x(x+a)$ and $b = u/(u+1)$ and $c = v/(v+1)$.

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

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

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

Context: Next 700 Data Description Languages

Published in CACM in 1966

At the time:

- COBOL
- FORTAN

ISWIM

- ALGOL
- (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.

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 for characterizing functional relationships. Within this framework the design of specific languages fall into two independent areas: the choice of written appearances of programs (or code generally, their physical presentation); and the choice of the objects of computation (sets, 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 begin to appear in more advanced systems, without sacrificing the important operators that make conventional right-hand-side evaluations easy, consistent and understandable.

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 parts. The ISWIM language is one that many linguists and logicians have come to terms with. In particular, the latter, we hope, will appreciate that 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, or more precisely, 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

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.”

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 curiously meager. The appearance may be especially misleading to someone who has not yet learned how much of current computer power is 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 elements of the program. To declare or otherwise constrain its use, however, varies considerably from one language to another. Equally curious within a single language there may be very different conventions for treating 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.)

ISWIM is thus part programming language and part problem-oriented research. It is possible to imagine the problem as 1700 dots in the grid of Correspondence between Church and Church's invention.¹

2. The where-Notation

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

$$\begin{aligned} & f(b+2c) \quad f(b+2c) \\ & \text{where } x = b + 2c \quad \text{where } f(x) = x(x+a) \\ & f(b+2c) + f(2b-c) \\ & \text{where } f(x) = x(x+a) \\ & \text{and } b = u/(u+1) \\ & \text{and } c = v/(v+1) \\ & g(f \text{ where } f(x) = ax^2 + bx + c, \\ & u/(u+1), \\ & v/(v+1)) \\ & \text{where } g(f, p, q) = f(p+2q, 2p-q) \end{aligned}$$

Context: Next 700 Data Description Languages

Published in CACM in 1966
At the time:

- COBOL
- FORTRAN
- ISWIM
- ALGOL

- (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.

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 use of user-defined names and the composition of programs, distinguishing "primitives" from "statements." Within this framework, two choices are possible into which a language can be cast, each having its own independent point. 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 designed to have "expressions" rather than "statements." It includes a functional calculus (a highly refined subsystem that aims to make the most general kinds of rules 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 what they can do, and partly a basic set of given primitives. The language W.M. (What Means) system is a by-product of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncrasies 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 paper below shows the many language characteristics required for a language designed for 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 possible combinations of what is what is needed to specify the language is shown 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

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."

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 fared spite of the less relevant criticisms of its idiosyncrasies.

Some insights by 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, is almost every language that can coin names, able to constrain the contexts in which the name is used? Is there a rule that 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 immediately used are not the same as the rules that they must be given a meaning.) There may also be penitential, by leading to puns such as ALGOT'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, as in

where $x = b + 2c$ 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

The possible combinations of what is what is needed to

specify the language is shown below:

$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$
 $f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$
and $b = u/(u+1)$
and $c = v/(v+1)$
 $g(f \text{ where } f(x) = ax^2 + bx + c,$
 $u/(u+1),$
 $v/(v+1))$
where $g(f, p, q) = f(p+2q, 2p-q)$

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

Context: Next 700 Data Description Languages

Published in CACM in 1966

At the time:

- COBOL
- FORTRAN
- LISP
- ALGOL
- (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.

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 use of user-defined names and the representation of data structures and relations. Within this framework, two choices can be made independently:

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 designed to support "expressions" rather than "statements." It includes a subsystem that aims to make the constraints 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 general, and partly a basic set of given primitives. The W-M (What-Mean) system is a by-product of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncrasies are concerned with the former rather than the latter, whereas aptitude for a particular class of tasks is essentially determined by the latter. (Other than the formalism of lambda calculus, no many-language characteristics require such a strong 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 possible primitives, and what is needed to specify them, are given 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

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."

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

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 fended off some of the less relevant criticisms of its primitives.

Use of 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, almost every language contains coin names and symbols in almost every context in which they are used and are elaborated in 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 immediately used are not the same as rules. When they are, they are still too often too 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.

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

The following examples illustrate the uses of ISWIM.

For example, in addition to computation and communication, these uses or 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 $c = v/(v+1)$

$$g(f \text{ where } f(x) = ax^2 + bx + c,$$

$u/(u+1),$

$v/(v+1))$

where $g(f, p, q) = f(p+2q, 2p-q)$

Context: Next 700 Data Description Languages

- business programming
 - then: COBOL
 - now: €€€€€ 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.

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 use of user-defined names and the composition of such names into meaningful units. Within this framework, two choices can be made independently 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 designed to have "expressions" rather than "statements." It includes a general-purpose (general) subsystem that aims to make the same general problem 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 particular applications, and partly a basic set of given things. The former is called "problem orientation." The latter is called "language characteristics." In many language characteristics require some kind of language-oriented problem orientation.

This attempt has led the author to think that many linguistic idiosyncrasies 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 author believes that many language characteristics require some kind of language-oriented 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 possible combinations of set and what is needed to specify them 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

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."

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

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 fared spite of the less relevant criticisms of its idiosyncrasies.

Some insights by 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, it is almost always the user who can coin names, and it is also certain that there are contexts in which the user would also be restricted by 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 how restrictions on what names can be coined sometimes affect the meaning of what they mean. (This is not necessarily a bad thing, but it is pernicious, by leading to puns such as ALGOT'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 between the name and the object it denotes.

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

The author would like to emphasize that, in addition to scientific and technical communication, these uses of x require no explanation. Nor do the following:

$$\begin{aligned} &f(b+2c) + f(2b-c) \\ &\text{where } f(x) = x(x+a) \\ &f(b+2c) + f(2b-c) \\ &\text{where } f(x) = x(x+a) \\ &\text{and } b = u/(u+1) \\ &\text{and } c = v/(v+1) \\ &g(f \text{ where } f(x) = ax^2 + bx + c, \\ &u/(u+1), \\ &v/(v+1)) \\ &\text{where } g(f, p, q) = f(p+2q, 2p-q) \end{aligned}$$

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!)

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 use of user-defined names and the composition of such a language, and it also provides within its framework the ability to decompose a language 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 organized into "expressions" rather than "statements." It includes a general-purpose (general) subsystem that aims to handle the same problem which 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 general, and partly a family of basic set of given things. The Wirth (i.e., Wirth Mean) system is a by-product of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncrasies 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. In this paper, we shall try to lay language characteristics responsible for this biased 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 primitives to be chosen are underlain by what is needed to specify them, but they are lower-level.

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

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."

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

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 fared spite of the less relevant criticisms of its idiosyncrasies.

Sights and 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 one can coin names, and in certain contexts in which the name could also be read in 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 immediately used are not the same as those where they are legal, since they may be pernicious, by leading to puns such as ALGOT'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 immediately implies a functional relation, and vice versa.

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

ISWIM is not designed for general 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 $c = v/(v+1)$
 $g(f \text{ where } f(x) = ax^2 + bx + c,$
 $u/(u+1),$
 $v/(v+1))$
where $g(f, p, q) = f(p+2q, 2p-q)$

Context: Next 700 Data Description Languages

Still true today!

Physical appearance: syntax

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

- (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.

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 idiosyncrasies 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

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."

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

$$\begin{array}{ll} x(x+a) & f(b+2c) \\ \text{where } x = b + 2c & \text{where } f(x) = x(x+a) \end{array}$$

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:

$$\begin{aligned} &f(b+2c) + f(2b-c) \\ &\text{where } f(x) = x(x+a) \\ &f(b+2c) + f(2b-c) \\ &\text{where } f(x) = x(x+a) \\ &\text{and } b = u/(u+1) \\ &\text{and } c = v/(v+1) \\ &g(f \text{ where } f(x) = ax^2 + bx + c, \\ &\quad u/(u+1), \\ &\quad v/(v+1)) \\ &\text{where } g(f, p, q) = f(p+2q, 2p-q) \end{aligned}$$

Context: Next 700 Data Description Languages

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

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 349 AI Memo No. 353 AI Memo 379

SCHEME
AN INTERPRETER FOR EXTENDED LAMBDA
by
Gerald Jay Sussman and Guy Lewis Steele Jr.

LAMBDA
THE ULTIMATE IMPERATIVE
by
Guy Lewis Steele Jr. and Gerald Jay Sussman

LAMBDA
THE ULTIMATE DECLARATIVE
by
Guy Lewis Steele Jr. *

Abstract:
Inspired by ACTORS [Greif and Hewitt] [Smith] implemented an interpreter for a LISP-like language lambda calculus [Church], but extended for side effects process synchronization. The purpose of this implementation is to:
(1) alleviate the confusion caused by Micro-PLANNER clarifying the embedding of non-recursive control host language like LISP.
(2) explain how to use these control structures, pattern matching and data base manipulation.
(3) have a simple concrete experimental domain for programming semantics and style.

This paper is organized into sections. The first "reference manual" containing specifications for SCHEME. Next, we present a sequence of programs in various programming styles, and how to use them. Various issues of semantics which we will try to clarify in the third section. In the fourth section we will discuss issues facing an implementor of an interpreter for lambda calculus. Finally, we will present a complete SCHEME, written in MacLISP [Moon], to acquaint the reader of implementing non-recursive control language like LISP.

This report describes research done at the Massachusetts Institute of Technology's Artificial Intelligence Research Laboratory. The work was funded by the Defense Advanced Research Projects Agency of the Department of Defense under contract N00014-75-C-0643.

Abstract:
We demonstrate how to model the following common patterns of an applicative order language similar to LISP:
Simple Recursion
Iteration
Compound Statements and Expressions
GO TO and Assignment
Continuation-Passing
Escape Expressions
Fluid Variables
Call by Name, Call by Need, and Call by Reference. The models require only (possibly self-referent) conditionals, and (rarely) assignment. No compilation stacks are used. The models are transparent, i.e., they transform programs without changing their meaning.

Some of these models, such as those for GO TO known, and appear in the work of Landin, Reynolds, and Moon. This paper is partly tutorial in intent escape expressions, fluid variables, and call new. The paper is designed to be used as a guide to the reader of implementing non-recursive control language like LISP.

This report describes research done at the Massachusetts Institute of Technology's Artificial Intelligence Research Laboratory. The work was funded by the Defense Advanced Research Projects Agency of the Department of Defense under contract N00014-75-C-0643.

Structure and Interpretation of Computer Programs

Second Edition



Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

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 by a unified framework. This framework dictates the 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 designed to be modular, so that subsystems can be merged or separated. The important properties of expressions easy to

1. Introduction

Most programming languages express things in terms of other basic set of given things. The ISWM (Mean) system is a byproduct of these two sets:

the language problem oriented

ISWM is an attempt at describing things in terms of problem-oriented by approach. So it is not a language of which each member is a primitive. That is needed to

ISW

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

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 would have been "Church without lambda."

$$\begin{aligned} & \text{where } f(x) = ax^2 + bx + c, \\ & u/(u+1), \\ & v/(v+1) \\ & \text{where } g(f, p, q) = f(p+2q, 2p-q) \end{aligned}$$

Notable:
~25 years before calcification of
PL families

Another 10 years for data
description languages?

(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