

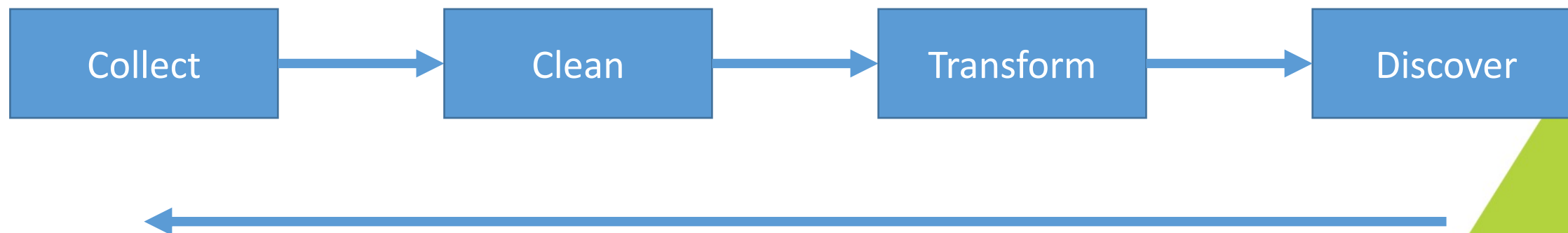
CS 295B/CS 395B
Systems for Knowledge
Discovery

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

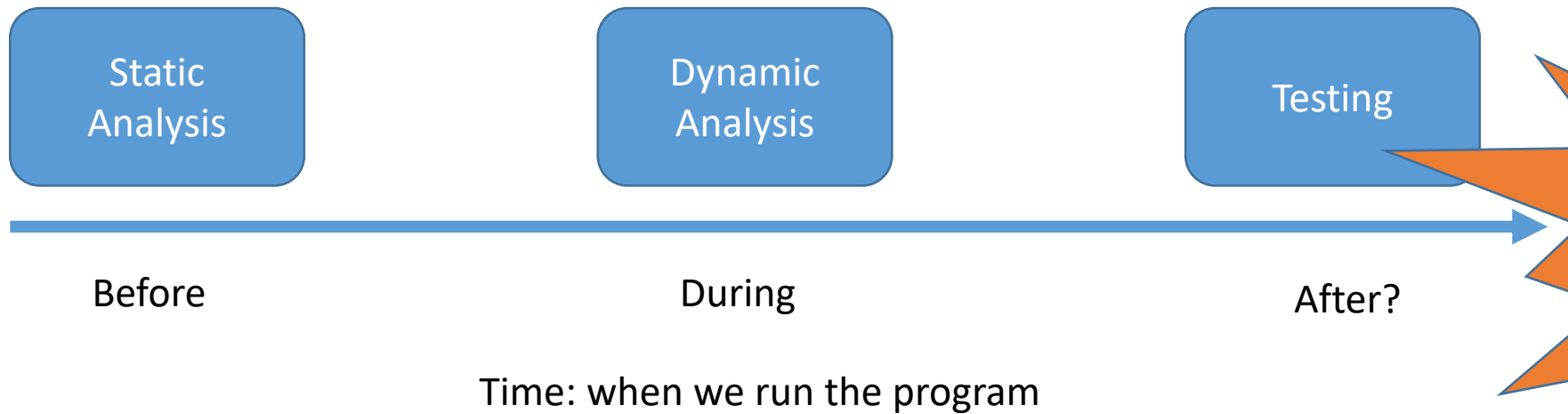


All these properties are
over static programs

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

Module – w.r.t. detailed design

Unit – w.r.t. implementation

Program

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

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 overall

Integration – w.r.t. subsystem design

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

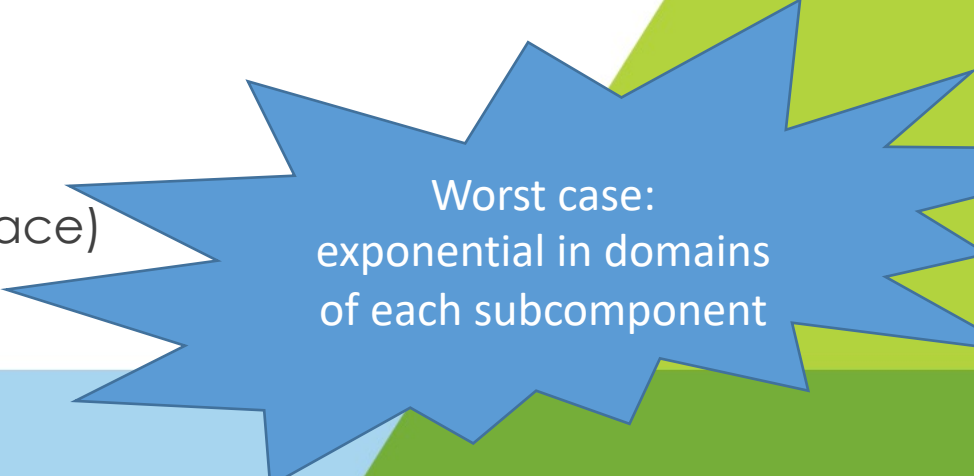
- Many images will be nonsense
- i.e., only a small subset of inputs is actually meaningful

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 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

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 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 – 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)

Submitted 02/13; published 06/13

The Arcade Learning Environment: An Evaluation Platform for General Agents

MG17@CS.UALBERTA.CA
EMPIRICALRESULTS.CA

Marc G. Bellemare
University of Alberta, Edmonton, Alberta, Canada
Yavar Naddaf
Empirical Results Inc., Vancouver,
British Columbia, Canada
Joel Veness
Michael Bowling
University of Alberta, Edmonton, Alberta, Canada

Abstract

In this article we introduce the Arcade Learning Environment (ALE), a domain-independent AI technology. ALE provides a platform and methodology for testing domain-independent AI technology. ALE provides a platform and methodology for testing domain-independent AI technology. ALE provides a platform and methodology for testing domain-independent AI technology.

1. Introduction

A longstanding goal of artificial intelligence is to develop a general competency in a variety of tasks. To this end, different types of artificial intelligence have been developed around the idea of "big" artificial intelligence competitions such as the General Game Playing competition, the International Joint Conference on Artificial Intelligence (IJCAI) competition, and the AAAI competition. The AAAI competition is a general game playing competition that has been running since 1997. The AAAI competition is a general game playing competition that has been running since 1997.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50

ToyBox: A Behavioral Testing Framework for Deep Reinforcement Learning Under Double-Blind Review at ICSE 2020. DO NOT CITE OR DISTRIBUTE.

Emma Tosch¹, John Foley², Kaleigh Clary¹, David Jensen¹
¹University of Massachusetts Amherst, ²Smith College

ABSTRACT

Autonomous deep reinforcement learning (RL) agents have achieved remarkable results on complex tasks that meet or exceed human competitive performance. High-profile examples include agents that beat humans at games such as Starcraft II, DOTA 2, and Go. Deep RL's successes have been accompanied by an acceleration in new RL research, seeking to develop agents that perform well in increasingly complex environments. Unfortunately, the pace of research has not kept up with our understanding of these agents. Deep RL is predicated on the notion that a deep network exhibits the same properties as the non-deep function it approximates. However, deep learning policies are often brittle and incomprehensible, because agents sequentially mutate their environment. We propose behavioral tests for RL agents. To this end, we have developed ToyBox, the first behavioral testing framework for deep RL. ToyBox provides efficient white-box environments that permit intervention, and a light-weight testing framework that permits case studies and falsify one prominent claim about deep agents' intelligence. ToyBox is open source; test configurations are isolated and easily distributed, facilitating reproducibility.

ACM Reference Format:
Tosch, E., Foley, J., Clary, K., Jensen, D. 2021. ToyBox: A Behavioral Testing Framework for Deep Reinforcement Learning. Under Double-Blind Review at ICSE 2020. DO NOT CITE OR DISTRIBUTE. Emma Tosch¹, John Foley², Kaleigh Clary¹, David Jensen¹. ¹University of Massachusetts Amherst, ²Smith College. In Proceedings of ACM Conference (Conference '21). ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/nmmmmmmmmmm>

1 INTRODUCTION

Reinforcement learning (RL) is a subfield of machine learning that develops methods for training agents to interact with an environment.

Deep RL rose to prominence in 2013 when DeepMind first presented work demonstrating a new deep learning training algorithm that beat the state-of-the-art (traditional, i.e. not deep) RL agents on seven Atari games from the Arcade Learning Environment (ALE) [6, 36, 38]. Their subsequent Nature paper established ALE Atari as the *de facto* deep RL benchmark for new training algorithms [39]. DeepMind then set the standard for human-competitive performance when they showed a newer algorithm surpassing human performance in 31 out of 49 ALE Atari games [60].

Atari has several appealing qualities: humans learn to play the "real world" environments that were not originally constructed to evaluate RL methods, and they have greater complexity than prior environments used in basic RL research. Furthermore, as a collection of games that require a variety of skills to master, ALE is highly appealing as a benchmark suite.

Unfortunately, ALE Atari has a major drawback: it is effectively a black-box. Therefore, assertions about intelligent agent behavior remain untestable. For example, ALE does not enable testing the conjecture that agents trained on Breakout learn to build tunnels [39] or that they enter a tunneling mode [20]. No system currently permits testing that could falsify such hypotheses about agent behavior.

This lack of testing is concerning in its own right; after all, RL agents are software, and "software without tests or verification has bugs!" Testing also enriches the goal of reproducibility in deep RL, which has received increased attention due to the number of deep RL algorithms and uncertainty about their relative merits [22, 25, 36]. Nearly all of this work on reproducibility focuses on the *how* (i.e. the specific hyperparameters, training procedures, and evaluation metrics) rather than the *what* (i.e. the specific tasks and environments).

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 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

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)

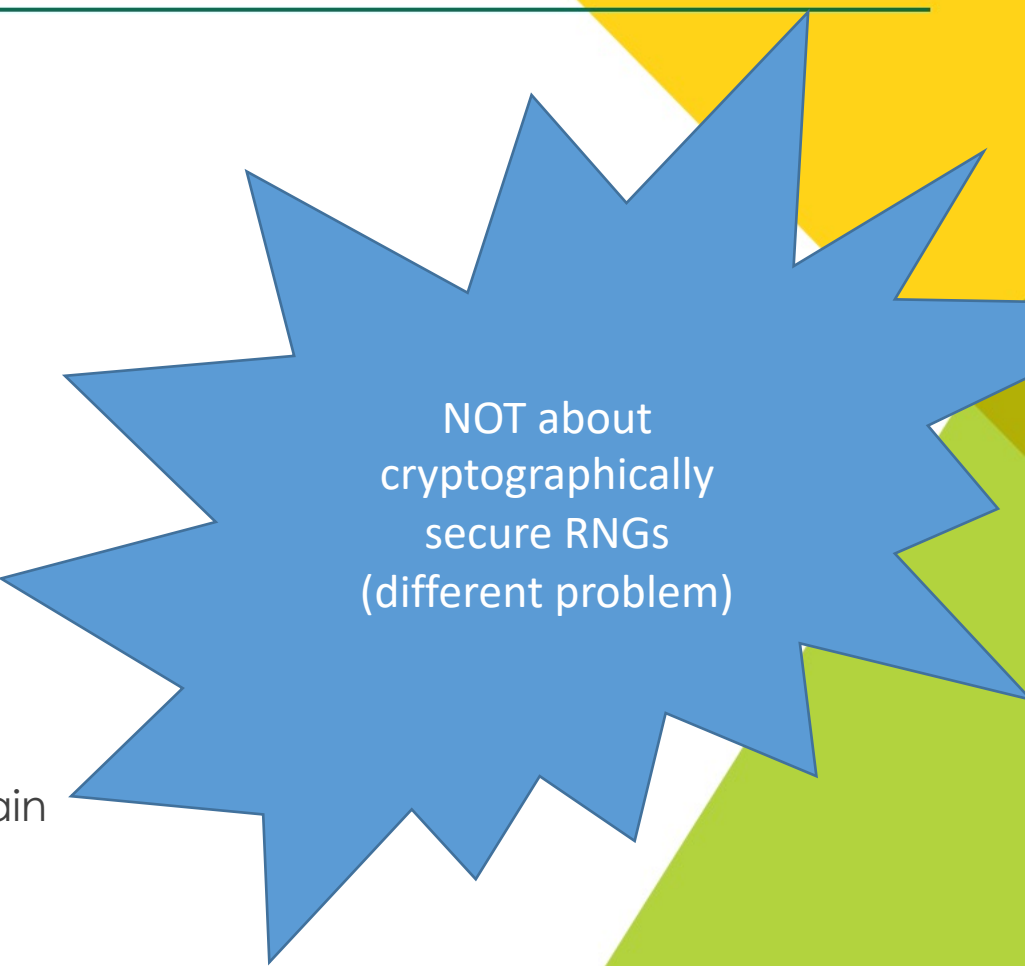
Global: generation

- 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)

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

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

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

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 use of user-coined names, and the conventions about choice of primitive operations. In this framework the design of a language is an independent task. It is the choice of the application areas 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) system for describing the data structures, and the benefit of a declarative instruction set. The system is designed to permit the programmer to write expressions easily, consistently and understandably.

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 bring these two aspects in some agreement.

This attempt has led to a number of 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 dependent on the intended application area. The choice of primitive operations 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 of primitives 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 obvious deficiencies of its deficiencies.

At first sight the facilities provided by Iswim will appear comparatively meager. This appearance will be greatly misleading. The manual, which is not appreciated in much of current manuals, are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialities. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the names are used and their relation to other contexts. The rules to produce, define, refer to, and constrain its use. The rules are considerably more restrictive 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 a real effect, it is likely to be non-trivial, by leading to problems such as ALGOL's lambda.)

So rules about user-coined names is available in which we might expect to find a variety of constraints. Such conditions give ground on 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 $f(b+2c)$ and $f(2b-c)$ where $f(x) = x(x+a)$. The latter is a 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

These theses are written in a language which 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) = ax2 + 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:

- FORTRAN

- 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

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 construction of expressions, and the organization of the program into 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 oriented towards "expression" rather than "statements." It includes a subsystem that aims to 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, that can be basic set of given things (Church's λ -notation, the 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 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. In fact, in many languages characterizing the former is more important than the latter. In fact, in many languages characterizing the latter is more important than the former.

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 choice of primitives is what is needed to specify the 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 relevant criticisms of its

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 programmer can coin names, and the contexts in which 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 are a very important part of the logic. When they are not, they are a source of confusion, 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 requires a functional relation. For example, the expression $f(b+2c)$ where $f(x) = x(x+a)$ 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."

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."

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

Languages differ in:

- "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 uses of user-coined names, and the constraints about the uses of user-coined names. (Thinking of this as a family of languages is a bit misleading, since the independent points are 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 aimed towards "expression" rather than "statements." It includes a subsystem that aims to be a subsystem 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, that can be basic set of given things (the "primitives"). 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 idiosyncracies 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 former, the latter is determined by many language characteristics, such as the degree of 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" are chosen so that what is needed to specify a particular problem is a small set of primitives.

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

design. The slight differences 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 specialities. For example, in almost every language a user can coin names

that introduce, define, declare, or otherwise constrain their 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 used are not necessarily the same as the restrictions on what they can be used for. The latter can be particularly confusing, 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 implies a functional relation. For example, the following are all 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 $g(f, p, q) = f(p+2q, 2p-q)$

Languages differ in:

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

Context: Next 700 Data Description Languages

- 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

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 constraints on their physical appearance. The framework is intended to be independent of the particular application area. 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 oriented towards “expressions” rather than “statements.” It includes a subsystem that aims to make 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, that can be basic set of given things. The (Wirth’s 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 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. In many languages, the physical appearance of language characters is determined by the 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 physical appearance of what is needed to specify a particular member of the family.

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

Some fielded some of the less relevant criticisms of its appearance. The slight differences 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 specialities. For example, in almost every language a programmer can coin names, and the contexts in which the names are used are 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 are also a source of confusion. When they are not stated, they are often assumed to be permissive, 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 implies a functional relation. For example, the expression $f(b+2c)$ where $f(x) = x(x+a)$ 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.”

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.”

of “where” require no explanation. Nor do the following:

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

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.

Languages differ in:

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

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 construction of statements, and the physical appearance of the language. 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 directed towards “expression” rather than “statements.” It includes a subsystem that aims to 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, that can be problem-oriented by appropriate choice of “primitives.”

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. In any language charged with the responsibility of being problem oriented.

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 author has found that a certain amount of syntactic variation is needed to specify the following uses of “where” require no explanation. Nor do the following:

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

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 programmer can coin names, and this is done in various contexts in which the programmer must attend 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 introduced by the programmer are not the same. While they are not always as obvious 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 implies a functional relation. For example, 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.”

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

$f(b+2c) + f(2b-c)$
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 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

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

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

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

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?

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 intended

subsystems can be merged important properties expressions easy to

1. Introduction

Most programming languages expressing things in terms of other basic set of given things. The ISWIM (Intermediate Symbolic Writing Mean) system is a byproduct of these two systems.

the language problem oriented

ISWIM is an attempt describing things in terms of problem-oriented by approach. So it is not a language of which each member has primitives. The system is needed to

ISWIM system

In practice, this is true 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.

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."

Next 700 Data Languages

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