OPLSS 2023 - Program Analysis

Speaker: Sukyoung Ryu — KAIST

July 3rd - 5th 2023

Abstract

This document is the companion notes for the program analysis lecture series at OPLSS 2023 by Professor Sukyoung Ryu. Program analysis is the study of program behavior. This lecture series focuses on static analysis in particular.

Part I
Lecture 1

1 Software Bugs

1. Bugs can get people killed or destroy millions of dollars in property.
2. Bugs are still very much still around today as in the 90s.
3. They're even more expensive in production than in development. Better to find (and fix) them early.
4. Many ways and ideas for how to do that (tradeoffs).
5. The focus for this lecture series is on static analysis (SA), which can be used for many applications. Her focuses mostly on using it to find security vulnerabilities.
6. Typically SA focuses on analyzing for bad behaviors rather than good.

2 Properties of interest

Program analysis can be used for various program interests like verification, bug detection, optimization(e.g., compiler optimiztaion), Understanding (e.g program slicing and reducing) etc.

2.1 Safety Properties

A safety property states that a program will never exhibit a behavior observable with finite time.

1. "Well Typed program never go wrong" translates to well-typed program never run to error states
2. For e.g program never runs into states such as integer overflow, buffer overflow, deadlock, etc.

Invariant In order to prove that the program never runs into error states, we insert a global assertion that is always true for a set of states. In any program execution starting from a state with an invariant, the computation step taken will lead to another state in the invariant.

Loop Invariant: Assertion is true at the beginning of every loop iteration.

Example1 : In this example, we have two loop invariants, where *x is an integer* and other $0 \leq x < 10$

```
x=0; while (x<10){x = x+1;}
```

Example2 : Division by Zero

```
int main(){
int x = input();
x= 2* x-1;
while(x>0){ x=x-2;}
assert (x!=0);
return 10/xl;}
```

2.2 Liveness Properties

The liveness property states that the program will exhibit a behavior observed only after an infinite time. Good things such as termination, fairness(means no starvation: the requester can always get the resource it requests), etc do eventually occur. Upon failure, there exists an infinite counterexample.

Variant In order to prove that all states eventually reach the target states, we have the variant, where the quantity evolves towards the set of target states. The value for some well-founded order relations is strictly decreasing.
Example: An expression of integer type that always takes a positive value and strictly decreases.

```
x = pos_int();
while (x>0) { x= x+1; }
```

2.3 Trace Properties

Trace property is a semantic property $\mathcal{P}$ that can be defined as a set of execution traces that satisfies $\mathcal{P}$.
Safety and liveness properties are trace properties

2.4 Information flow (IF) properties

1. Not a trace property because it needs more than one trace.
2. Hyper-properties are properties across sets of traces. Information flow is one such property.
3. The related hyper-property of Secure Information Flow (SIF) was discussed in the first logical relations lecture, specifically the definition of noninterference from the Denning & Denning paper.
4. An informal example of a SIF property: all user input (public, or low security) should not cause the password (private, or high security) to be leaked. You want that to be true of all possible traces, so you can be sure the password is never leaked.

2.5 A Hard Limit : Undecidability

Rice's theorem states that any non-trivial semantic property of a program is undecidable. An intuitive proof of the theorem can be found on `https://en.wikipedia.org/wiki/Rice%27s_theorem#Formal_statement`.

What this means is that an analysis that is **automatic**, **terminating** and has **exact** reasoning is impossible. We can choose any two (of the three) properties.

2.6 Soundness and Completeness

Take any semantic property $\mathscr{P}$, and an analysis tool $\mathbf{A}$:

A **sound** analysis means $\forall P \in$ programs $\mathbf{A}(\mathbf{p}) = \mathbf{T} \rightarrow P$

A **complete** analysis means $\forall P \in$ programs $\mathbf{A}(\mathbf{p}) = \mathbf{T} \leftarrow P$ satisfies $\mathscr{P}$

3 Toward Computability

3.1 Testing

1. Testing is fundamentally dynamic (vs static) and has finite run-time.
2. Avoids the Undecidability trap by relaxing the termination requirement.
3. Testing as a strategy accepts nonteriminating code, often by incorporating timeouts.
4. Pros: The positives are usually real, not false (spurious). If the test fails there should be a true bug there and you often have the concrete counterexample as well.
5. Cons: Doesn't prove or guarantee anything when bugs are not found.
6. "Absence of evidence is not evidence of absence", Dr Carl Sagan
7. symbolic or concolic execution is placed under testing in this lecture, a controversial opinion in certain circles.

3.2 (Human) Assisted Proving

1. Avoids the Undecidability trap by relaxing the automation requirement.
2. Require nontrivial human labor and skill
3. Up to the limits of the prover, can get both soundness and completeness
4. Powerful features users want, like generics, can vastly complicate the proof and the effort needed.
5. Not always possible, but great when you can get it!

3.3 Model Checking

Automatic, sound and complete (with respect to the model).

1. Model: finite automata generated from the target program.
2. Specification: written by human, usually in logical formula.
3. Verification: exhaustive search of the state space (graph .reachability)

Example: SLAM (MS Windows device driver verifier)

3.3.1 Model

A model is a finite state machine, manually or automatically constructed from program. Either unsound or incomplete, with respect to the target program. The very first construction of model can be low efficient, so **refinement** is required. But refinement may not terminating, so stopping mechanisms are required!

3.3.2 Specification

Written in *modal logic* Modal Logic = propositional logic + necessarily, possibly E.g. truth values of assertions vary with time (temporal logic). like Linear Temporal Logic, Computational Tree Logic. "x is always positive" "x remains positive until y is negative"...

3.3.3 Reachability Check

Sometimes you search some certain error states in your model (FSM). but you are sure they are unreachable, aka spurious reachability. in this case, you need to refine your model to rule them out.

3.3.4 Abstract Refinement

Automatically refine the model when a spurious counterexample is found input: old model (with spurious states) output: new model, concluding that the spurious error states are infeasible keeps running until a "real" (not spurious) counterexample is found or one proof is completed which means, it may not terminate example: CEGAR: CoutnerExample-Guided Abstraction Refinement

4 Static Analysis

5 Summary

The **Dafny** tool would come in the Assisted Proving row. Note that what is a *Model* vs. *Program* can be fuzzy. For example, when Model Checking at program level one can argue that we are checking the Model and not the actual Program that runs on hardware.

	Automatic	Sound	Complete	Object	When
Testing	Yes	No	Yes	Program	Dynamic
Assisted Proving	No	Yes	Yes/No	Model	Static
Model Checking of finite-state model	Yes	Yes	Yes	Finite Model	Static
Model Checking at program level	Yes	Yes	No	Program	Static
Conservative Static Analysis	Yes	Yes	No	Program	Static
Bug Finding	Yes	No	No	Program	Static

6 Resources Mentioned in Slack or Class

1. Soundiness Manifesto
2. Astree, static analysis on a very restricted version of C (no general recursion) with a trophy case `https://www.absint.com/astree/index.htm`
3. "SmallCheck and Lazy SmallCheck automatic exhaustive testing for small values" (in Haskell) `https://www.cs.york.ac.uk/fp/smallcheck/smallcheck.pdf`
4. Principles of Abstract Interpretation `https://mitpress.mit.edu/9780262044905/principles-of-abstract-interpretation/`
5. "A few Billion Lines of code Later using static Analysis to find Bugs in the Real World" `https://web.stanford.edu/~engler/BLOC-coverity.pdf`
6. "The Relevance of Classic Fuzz Testing: Have We Solved This One?" `https://dl.acm.org/doi/abs/10.1109/TSE.2020.3047766`

Part II

Lecture 2

7 A sample Imperative language Synatx

8 Different Styles of Semantics

There are many flavors of semantics that are suited to different styles/ preferences. The original plan for the lecture intended to cover both operational and denotational semantics. However, to allow for more time and other topics, it will only focus on denotational semantics.

8.1 Operational Semantics

(a) Widely used style of semantics briefly mentioned on slide 4 of Lecture 2. Not used in the rest of the series.

(b) Brass tacks semantics. Focus on the how or computation of the result.

(c) Key component of this style is state, which will be ignored in this lecture series.

(d) Transitions are stateful.

8.2 Denotational Semantics

(a) Discussion begins on slide 5 of the Lecture 2 slides.

(b) 'Mathy' semantics. Focuses on the result

(c) No state, analogous to pure functional semantics.

(d) Transitions are pure (w.r.t state) like pure functional programming, made up of composing smaller parts into bigger ones.

(e) Many things don't change memory; they merely compute.

(f) Key components of this style are domains and functions. Will need domains and functions with the right properties to have nice correct analyzers.

(g) Intuitive, familiar recursion is a problem, that most of this lecture will focus on how we solve it by getting to fixed point forms.

9 Semantic Domains

Semantic domains include a set of semantic objects. Where the memory state boils down to a function $\mathbb{M}$ from the (fixed) finite set of variables $\mathbb{X}$ into a set of values $\mathbb{V}$. Thus the set of memory states $\mathbb{M}$ is defined by

$$\mathbb{M} = \mathbb{X} \mapsto \mathbb{V}$$

9.1 Semantics of Expressions

The semantics of scalar expression produces a scalar value. In the case of constant n, the result is simply n. In the case of variable x, the result is obtained by reading the content of variable x in the current memory state. The result of the evaluation of an expression composed of an operator applied to two sub-expressions is obtained by evaluating the sub-expressions and applying to their results the mathematical functions described by the operator. Last, the case of boolean expressions is very similar to the semantics of scalar expressions, with the only difference being that it returns a boolean value. Refer to slide semantics of expression from lecture 2 for the complete formalization of this semantics.

9.2 Semantics of Commands

The semantics of command C is a function noted $[\![C]\!]$ that maps a set of input states to a set of output states. Refer to slide *Semantics of commands* form *lecture2* for formal semantics. The semantics of the skip command is the identity function. The semantics of the sequence of commands is the composition of the semantics of each command. The evaluation of expression $x := E$ updates the value of x in the memory states with the result produced by the evaluation of E. The evaluation of $input(x)$ replaces the value of variable x with any possible scalar value. The evaluation of conditional command is determined by the result of the condition.

9.3 Semantics of Fixed Point

The case of loops is more complex and interesting due to unbounded executions. The validity of the semantics of a while statement is done mathematically. We need to solve the recursive equation, this leads us to find the least fixed point of the equation using Kleene's fixpoint theorem, and application of the theorem requires verifying the continuity of F. We want that the solution to be continuous. This will ensure our ability to prove the correctness of composed programs using structural induction.

10 Domain Theory

To define the meaning of programs in a given language, we must first, define the building blocks—the primitive data values and operations—from which computations in the language are constructed. Domain theory is a comprehensive mathematical framework for defining the data values and primitive operations of a programming language. A a critical feature of domain theory is the fact that program operations are also data values; in domain theory both operations and data values that the operations operate on are elements of computational domains. To support the idea of describing data values by generating "better and better" approximations, we need to specify an ordering relation among the finite approximations to data values.

10.1 Partial Order

A relation $\sqsubseteq$ is a partial order if it is a reflexive, transitive, and antisymmetric relation.

- reflexive $\forall x \in D \quad x \sqsubseteq D$
- antisymmetric $\forall xy \in D\ x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$
- transitive $\forall x,\ y,\ z \in D\ x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$

Poset: A set equipped with a partial order is called poset

10.2 Least Upper Bound

Let X be a subset of a partial order D and the element $d \in D$ is an upper bound of X iff $\forall x \in X$ and $s \sqsubseteq d$. An upper bound d of X is the least upper bound of X iff for all upper bounds y of $X \quad d \sqsubseteq y$. The least upper bound of X is denoted by $\bigsqcup X$

10.3 Chain

Let $(D, \sqsubseteq)$ be a partial ordered set. A subset $X \subseteq D$ is called *chain* if X is totally ordered.

$$\forall x_1, x_2 \in X.\ x_1 \sqsubseteq x_2\ or x_2 \sqsubseteq x_1$$

An upper bound in a chain is an element which is "bigger" than all of the other elements in the chain and for any upper bounded chain, there exists a unique least upper bound.

10.4 Complete Partial Order (CPO)

Definition (CPO). A poset $(D, \sqsubseteq)$ is a **C**omplete **P**artial **O**rder if every chain X of D has $\bigsqcup X \in D$.

In other words, for any given sequence in the set, you can always find an element which is greater than or equal to everything in that sequence but is also the smallest such element.

10.5 Continuous Function

Definition (Continuous Function). Given two partially ordered sets D_1 and D_2, a function $f : D_1 \to D_2$ is **continuous** if it preserves least upper bounds of chains:

$$\forall chain X \subseteq D_1. \bigsqcup_{x \in X} f(x) = f(\bigsqcup X).$$

This means that if you have a chain of elements (a sequence of elements where each is less than or equal to the next), and you apply the function to each element in the chain, the resulting sequence also forms a chain. If you take the least upper bound (lub) of the original chain and apply the function to it, you get the same result as if you took the lub of the chain resulting from applying the function to each element. The two examples on slides are (1): not monotone and (2): not preserving LUB.

11 Fixed Point

Finally we can have a formal definition of fixed point. Definition (Fixed Point). Let $(D, \sqsubseteq)$ be a partially ordered set. A **fixed point** of a function $f : D \to D$ is an element x such that $f(x) = x$. We write $\mathbf{lfp}f$ for the **least fixed point** of f such that

$$f(\mathbf{lfp}f) = \mathbf{lfp}f$$

and

$$\forall d \in D. f(d) = d \Rightarrow \mathbf{lfp}f \sqsubseteq d$$

So, why do we need a fixed point? The simple answer is that a fixed point of a function is an element that is left unchanged by the function. This directly corresponds to the nature of a while loop: when a while loop finishes executing, the state of the program is in a "fixed" state where executing the while loop again wouldn't change the state. This is exactly what we need to capture the semantics of a while loop. (considering a while loop when it has been "finished": the condition has been turned to false so run it again won't execute anything, well, except some side effects, this corresponds to a "fixed point" in math, so we use it to represent the while block.)

12 Resources Mentioned in Slack or Class

(a) A different lecture that used the same notations and definitions. Well reviewed by some participants https://www.cs.cornell.edu/courses/cs6110/2014sp/Lectures/lec21.pdf

(b) article mentioned on how to construct a fix point https://medium.com/@dkeout/why-you-must-actually-understand-the-%CF%89-and-y-combinators-c9204241da7a

Part III
Lecture 3

13 Abstract Interpretation

First coined by Patrick and Radhia Cousot, Abstraction in general can be done by retaining only rather coarse information about program states, and considering how the program runs, we can still infer interesting information about the set of all program executions. Such information is captured by a set of logical properties that the analysis may manipulate. Abstract interpretation hence provides a strong mathematical foundation for reasoning about static program analysis.

-Is my analysis sound?-(Does it safely approximate the actual program behavior?) -Is it as precise as possible for the currently used analysis? -If not, where can precision losses arise? Which precision losses can be avoided (without sacrificing soundness)? Answering such questions requires a precise definition of the semantics of the programming language, and precise definitions of the analysis abstractions in terms of the semantics. Many forms of semantics have been proposed, and some are more expressive than others. For instance, trace semantics describes program executions as sequences of program states, whereas denotational semantics describes only input-output relations, and ignores the intermediate steps. Some other forms of semantics only describe the sets of reachable states. In this lecture series, denotational semantics was used.

In this context, we can use *concrete* qualifier to denote actual program behaviors, whereas the "abstract" qualifier applies to the properties used in the (automatic) proofs. As an example, concrete semantics is the actual semantics of programs. By contrast, abstract semantics shall define a computable over-approximation of the concrete semantics expressed in terms of abstract states. Hence, the goal of static analysis is precisely to compute such sound abstract semantics.

Concrete Semantics Concrete semantics are usually a simple extension of standard semantics, formalizing all possible program executions. It can be referred to as collecting semantics. We use the same standard semantics for commands, expressions, and fixed points from the previous lecture, except we take a power set of memory states. We could use traces(sequence of states) or reachable states(set of states) for concrete semantics.

Abstract Semantics Collecting the exact set of all the states that can occur during program executions (transition sequences) is in general either too costly or impossible in finite time. Indeed, due to loops, program executions may be arbitrarily long. It is important to note that inaccurate does not mean wrong. Indeed, if the kind of inaccuracy is known, the user may still draw (possibly partly) conclusive results from the analysis

output. For example, suppose we are interested in program termination. Given an input program to verify, the program analysis may answer *yes* or *no* only when it is fully sure about the answer. When the analysis is not sure, it will just return an undetermined result: don't know. Such an analysis would be still useful if the cases where it answers "don't know" are not too frequent. Note, without abstract semantics, it is undecidable to subsume all possible behaviors of the software.

To be able to make approximations sound(or correct), we should formalize these approximations. We formalize the meaning of the program (semantics) and then the approximations.

$$Idea : Concrete\ Semantics \cong Abstract\ Semantics$$

13.1 Abstract Domain

The point of abstract interpretation is computing for a "sound approximation" of the semantics of programs. How do we define the approximation of the programs in an exact form? The answer is abstract domain: a CPO representing a range of possible concrete states.

(a) Define an abstract domain $\mathbb{D}^\sharp$ (CPO)

(b) Define an abstract semantic function $F^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ (this function must be monotone or extensive): the intuition of these two properties are that, the abstract semantic function is a "description" or "interpretation" of the source program, but this "description" is lossy: some information about the source program is lost during this mapping. That's the motivation of the usage of poset: a poset can be used to represent the lost of information during the mapping. Similarly, we must control the lost in a certain degree (otherwise we can't perform analysis!), so we require that this abstract semantic function must be monotone or extensive.

Monotone: $\forall x^\sharp, y^\sharp \in \mathbb{D}^\sharp . x^\sharp \sqsubseteq y^\sharp \Rightarrow F^\sharp(x^\sharp) \sqsubseteq F^\sharp(y^\sharp)$

Extensive: $\forall x^\sharp \in \mathbb{D} . x^\sharp \sqsubseteq F^\sharp(x^\sharp)$

(c) Then we can conclude that, static analysis is to compute an upper bound of the chain:

$$\bigsqcup_{i \geq 0} F^{\sharp i}(\bot^\sharp)$$

14 galois connections

Now let's elaborate the concept of abstract semantic function. It can be represented by a more generalized concept: galois connection. The

Galois Connection is a pair of adjoint functions (α, γ): an abstraction function, which maps from the concrete to the abstract domain, and a concretization function, which maps in the opposite direction. These two functions must satisfy certain properties. For all elements c in the concrete domain and a in the abstract domain, it should be the case that: if you abstract c and then concretize, you get something less than or equal to c (concretize(abstract(c)) <= c), and if you concretize a and then abstract, you get something greater than or equal to a (abstract(concretize(a)) >= a). The abstraction function should over-approximate the concrete values, and the concretization function should under-approximate the abstract values. Formally, a galois connection consists of these two functions:

(a) Abstract function: $\alpha \in \mathbb{D} \to \mathbb{D}^\sharp$

(b) Concretization function: $\gamma \in \mathbb{D}^\sharp \to \mathbb{D}$

$\mathbb{D}$ is concrete domain, and $\mathbb{D}^\sharp$ is abstract domain.

15 Properties of galois connection

The galois connection must satisfy these properties:

$$id \sqsubseteq \gamma \circ \alpha$$

$$\alpha \circ \gamma \sqsubseteq id$$

monotone:

$$\alpha(x) \sqsubseteq \alpha(y)$$
$$\gamma(x^\sharp) \sqsubseteq \gamma(y^\sharp)$$

And two deriving properties: pointwise lifting and composition. Their definitions are on page 37 of lecture 3 slides. The slides contain more examples of analysis represented by galois connection, such as memory abstraction and operational semantics abstraction.

16 Fixed point transfer theorems

(a) Recall the two options for soundness between F and $F^\sharp$, and we're trying to get a fixpoint function instead of the intuitive recursive one.

(b) slide 43 has the formal definitions, one for each on slide 39.

(c) We roll up all the galois connections and chains into this

(d) If we meet its conditions, we get our soundness result for F and $F^\sharp$

(e) Since in this model a program is one big fat function, if we can get the least fix point of it, we get our abstract semantics

(f) We still need to compute it in finite time (we need a termination guarantee) and out pops the static analyzer

17 Widening Narrowing

(a) recall that we need an over-approximation of the original program to avoid running afoul of Rice's theorem.

(b) widening forces the convergence of fixed point iterations, which loses precision

(c) widening is what gets up the termination guarantee widening pushes us away from the ideal result.

(d) lost precsions results in increase the over-approximation

(e) but we don't want to have too much or it won't really be sound and useful anymore

(f) narrowing brings us back a little bit towards the ideal result but not enough to undo it. It refines but doesn't fix.

(g) both operators are safe and finite.

Part IV
Lecture 4

18 Lecture 4 Introduction

An exploration of the team's work, and how they got involved with JS and the winding path that led to automatic generation of JS static analyzers. There is also some advice for researchers at the end of the lecture 4 deck.
Nota Bene:

(a) Unfortunately the links are not clickable in the pdf.

(b) Notations across papers in the field are not consistent.

18.1 Papers Explored

(a) "SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript" (no public pdf, but might have it through your university library)

(b) Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling `https://ieeexplore.ieee.org/document/7372043`

(c) Practically tunable static analysis framework for large-scale JavaScript applications `https://dl.acm.org/doi/10.1109/ASE.2015.28`

(d) SAFEWAPI: web API misuse detector for web applications `https://dl.acm.org/doi/10.1145/2635868.2635916`

(e) Automatic Modeling of Opaque Code for JavaScript Static Analysis `https://link.springer.com/chapter/10.1007/978-3-030-16722-6_3`

(f) JISET: JavaScript IR-based semantics extraction toolchain `https://dl.acm.org/doi/10.1145/3324884.3416632`

(g) JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification `https://dl.acm.org/doi/abs/10.1109/ICSE43902.2021.00015`

(h) JSTAR: JavaScript Specification Type Analyzer using Refinement `https://dl.acm.org/doi/abs/10.1109/ASE51524.2021.9678781`

(i) Automatically Deriving JavaScript Static Analyzers from Language Specifications `https://dl.acm.org/doi/abs/10.1145/3540250.3549097` (JSAVER paper)

(j) Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations `https://dl.acm.org/doi/abs/10.1145/3591240` (JESTFS)

19 Intro Related Work

(a) Exception Analyzers, slide 4

(b) Debugging Everywhere slide 5

(c) 3 Big JS analyzer teams (including hers)

(d) Experimental High Performance Language: Fortress. Discontinued

20 How did this start, or why JS?

(a) JS (and its sibling TypeScript) were up and coming languages

(b) more importantly, a new student was very excited about JS

(c) JS (like PHP) has complex and unexpected semantics. See slide 9/10 for a pop quiz.

(d) "What could go wrong?"

21 Cracking the SAFE: Keeping up with a Changing Spec

(a) Group did several handwritten analyzers, and published

(b) the prose specification of ECMAScript (of which JS is an implementation) is well written and fairly consistent.

(c) But JS changes every year and analyzing it and implementing those changes are labor intensive

(d) Prose is not a thing a typechecker or machine understands

(e) So now they have a specification that machines understand based on the one humans understand.

(f) slide 15 is a conventional flow of an analyzer, so even if this one doesn't interest you, the diagram is still helpful.

(g) code at `https://github.com/sukyoung/safe`

(h) paper "SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript"

22 DOMs Gone Wild

(a) DOM stands for Document Object Model

(b) They had to include post processing to handle strange DOM behavior, so how common is that out in the wild? Answer: Common

(c) "Practical" really means "Bad" from a PL standpoint, but that's what's out in the wild.

(d) Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling `https://ieeexplore.ieee.org/document/7372043`

23 Tuning In, Turning up Bugs

(a) slide 17, diagram is important. Only interested in what goes on in the states in the red

(b) Since the other states are ignored, the analysis is unsound overall

(c) But sound within the red section

(d) since static analysis is an overapproximtion, conditional branches are a great place to lost some precision

(e) Practically tunable static analysis framework for large-scale JavaScript applications `https://dl.acm.org/doi/10.1109/ASE.2015.28`

24 The call is coming from inside the house: APIs Phone Home

(a) Work with a phone vendor and industry software engineers

(b) Engineers hate, hate, hate false positives (spurious reports)

(c) Engineers also don't have the same definition of 'bug' as PL people do! So that had to be negotiated. See slide 18.

(d) analysis becomes even more unsound, but still did the job

(e) SAFEWAPI: web API misuse detector for web applications `https://dl.acm.org/doi/10.1145/2635868.2635916`

25 Seeing inside the black (binary) box

(a) If you don't have the source, static analysis gets harder

(b) Dynamic analysis to build the model and statically analyze that

(c) if it doesnt work, more dynamic runs and build a bigger model

(d) so wouldnt work for things you can't run

(e) Automatic Modeling of Opaque Code for JavaScript Static Analysis `https://link.springer.com/chapter/10.1007/978-3-030-16722-6_3`

26 Get the machine to do your homework for you

(a) JS changes every year, but does have a consistent prose style.

(b) SAFE showed there was formal spec

(c) JISET automates the formal specification for you, so you don't have to read the new prose, just run the tool each year.

(d) uses an IR meta language for the human prose (made easier by the HTML of the spec document)

(e) then feed that into the tool chain to build the static analyzer

(f) JISET: JavaScript IR-based semantics extraction toolchain `https://dl.acm.org/doi/10.1145/3324884.3416632`

(g) code `https://github.com/kaist-plrg/jiset`

27 Surely you JEST!

(a) Differential testing is when you run different implementations of nominally the same thing against each other and see if they disagree. No oracle usually.

(b) However in this version they are testing for conformance to the specification using JISET as a defactor oracle.

(c) and also find bugs in the specification.

(d) GraalVM management is considering adding JEST to testing.

(e) JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification https://dl.acm.org/doi/abs/10.1109/ICSE43902.2021.00015

(f) code https://github.com/kaist-plrg/jest

28 Wish upon a shooting JSTAR

(a) Improve sensitibity and type awareness

(b) analyzer refinement for conditions

(c) find things like type mismatches in teh specification itself

(d) slide 53 starts the domains, semantics, details.

(e) JSTAR: JavaScript Specification Type Analyzer using Refinement https://dl.acm.org/doi/abs/10.1109/ASE51524.2021.9678781

(f) code https://github.com/kaist-plrg/jstar

29 Save more with JSAVER

(a) using the previous work, build the static analyzer

(b) chained into the rest of the system,, you can start with the prose spec and get eventually get out a tuned static analyzer

(c) Devs express curiosity, but not commitment

(d) Editors of spec glad someone beyond their W3C colleagues finally noticed the consistent prose. Express interest in integration.

(e) Automatically Deriving JavaScript Static Analyzers from Language Specifications https://dl.acm.org/doi/abs/10.1145/3540250.3549097

(f) code for meta language used in https://github.com/es-meta/esmeta

30 Surely you JEST again

(a) Coverage-guided mutation has done great things for fuzzing, so let's try it here.

(b) Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations https://dl.acm.org/doi/abs/10.1145/3591240 (JESTFS)

31 Whats next?

(a) Rust? Complex, gnarly target. Wise grad student knew trouble when he saw it.

(b) Verse? Not enough semantics yet to target

(c) WASM? Beautiful spec, collaboration. Latex is not as friendly as HTML so they gave up?!

(d) P4? Data plane of network, used on a lot of hardware. But the semantics of it are in someone's head, its hard to automate that

32 Advice

Advice from Guy L Steele Jr. Just g read the slide. Popular advice for a reason, but light on implementation details.