ORACLE

# Experiments with String Analysis

Kostyantyn Vorobyov, Yang Zhao, Raghavendra Ramesh and Padmanabhan Krishnan

Oracle Labs, Brisbane
November, 2019

# Safe harbor statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

# String Analysis

- **Compute values a string expression can take at a given program point**
- **Potential uses**
  - Security vulnerability detection
  - Input sanitisation and validation
  - Query generation
  - Data format generation (XML, JSON, HTML etc.)
  - Dynamic code generation
  - Dynamic class loading

# Analysis Performance

- **Trade-offs between precision and scalability**
- **RegEx generation**
  - More precise, less scalable
- **Constant propagation**
  - More scalable, less precise
- **Practical perspective**
  - Do we need to compute regular expressions for every client analysis?
  - Would a less precise but more scalable technique suffice for some?

# Evaluation

- **Investigate performance trade-offs of different string analysis techniques**
- **Investigate precision of a client analysis**
- **String analysers**
  - Java String Analyser
  - Oracle Labs String Analyser

# Java String Analyser (JSA)

- **Christensen, Møller and Schwartzbach**

  *Precise Analysis of String Expressions, SAS 2003*

- **Goal**

  Compute over-approximation of values a string expression may take at runtime

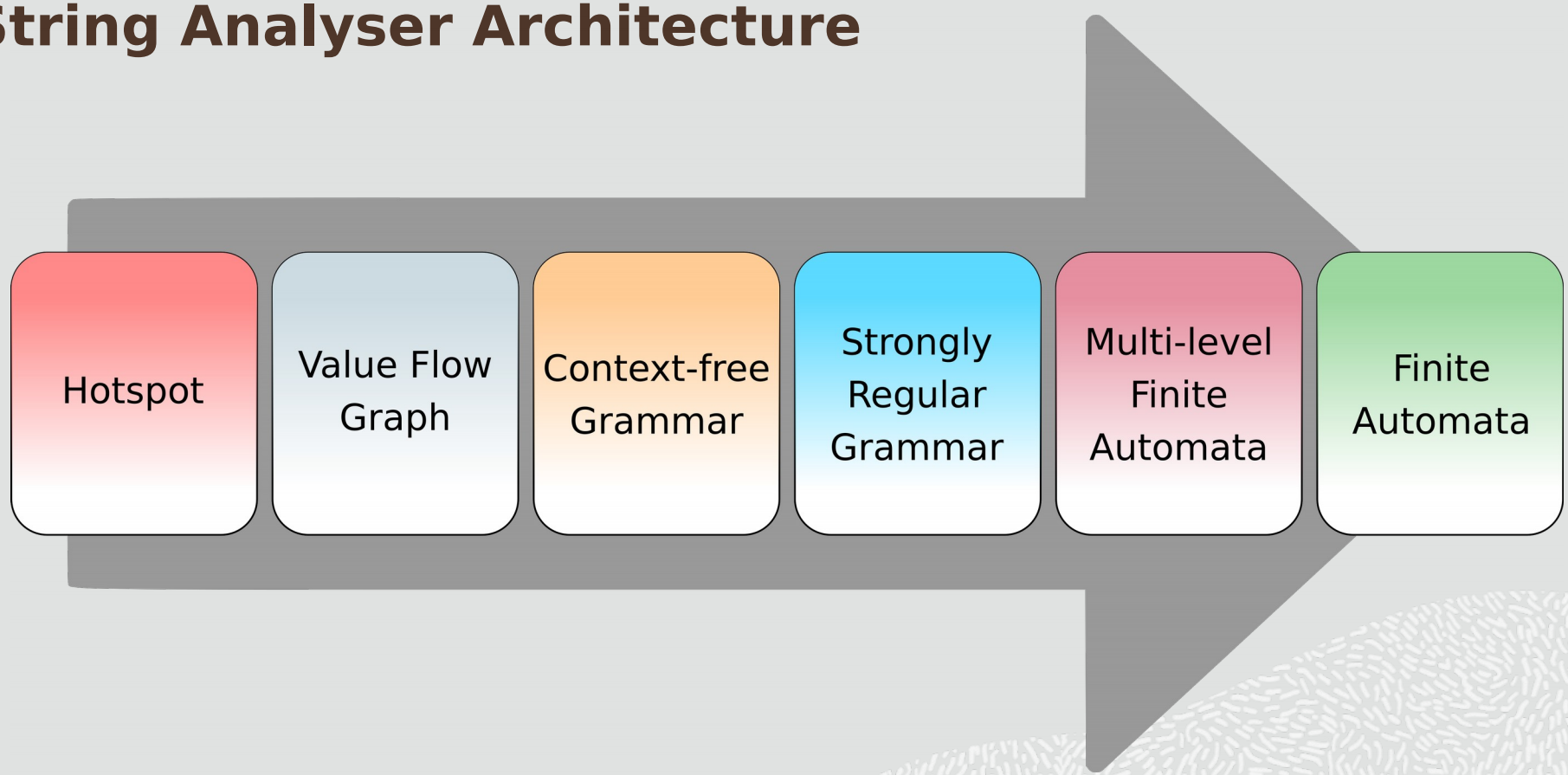
- **Target**

  Relevant string expressions (hotspots)

- **Outputs**

  Finite State Automata

# Java String Analyser Architecture



# Value Flow Graph

- **Edges**

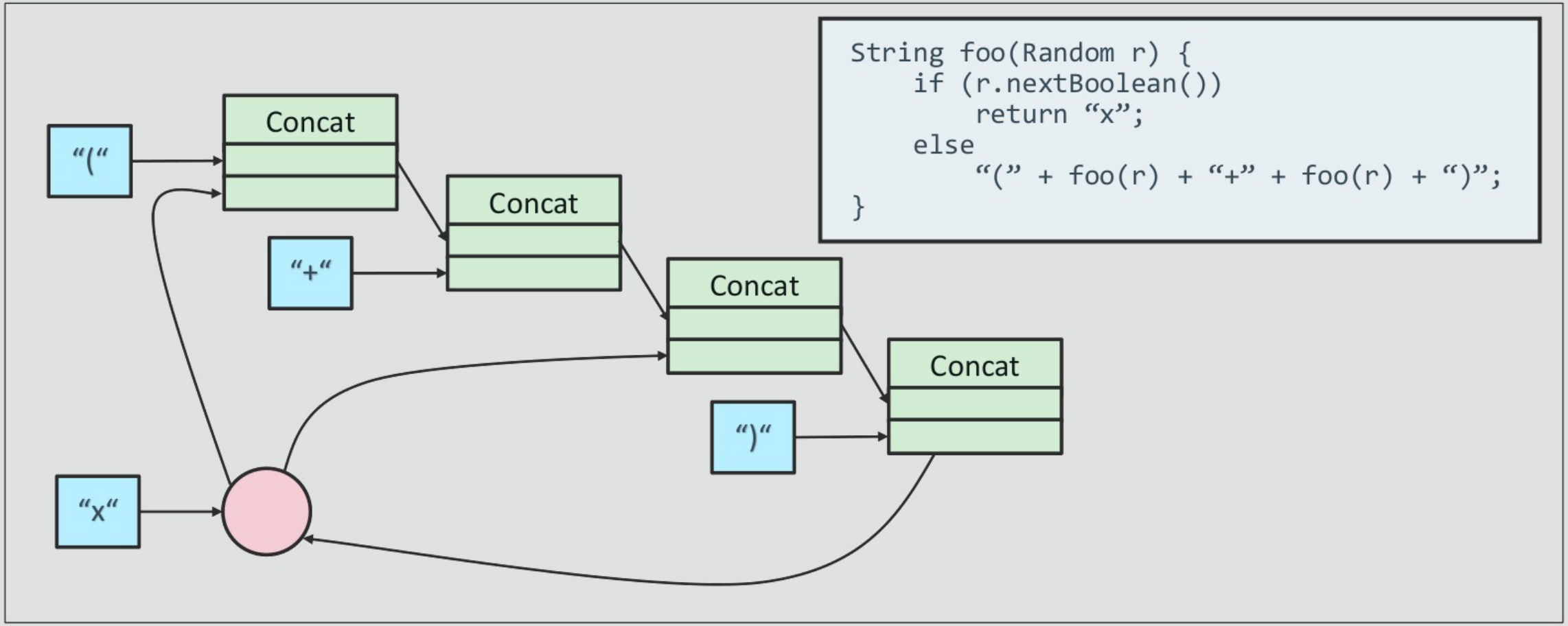
  directed def-use edges representing possible data flows

- **Nodes**

  variables or expressions

  - **Init** – string value from a constant
  - **Join** – assignment or other join location
  - **Concat** – string concatenation
  - ***UnaryOp*** – unary string operation
  - ***BinaryOp*** – binary string operation

# Value Flow Graph Example



# Value Flow Graph to Regular Expressions

- **Value Flow Graph to Context-free Grammar**

  Transform VFG to Context Free Grammar using transformation rules

- **Context-free Grammar to Finite Automata**

  Approximate CFG with a regular grammar containing original language

  Convert strongly regular grammar to Multi-level Finite Automata (MLFA, a hierarchical directed acyclic graph of NFA)

  Extract minimal FA for each hotspot from MLFA

# Oracle Labs String Analyser (OLSA)

- **Inspired by Java String Analyser**
- **Value Flow Graph extended with *Switch* nodes**
- **Context-sensitive constant propagation**

***Hotspot → VFG → Strings***

# Precision Evaluation

- **Compare precision of JSA and OLSA**
- **Subjects**

  Small test programs developed for JSA testing
- **Test program features**

  Single hotspot

  Hard-coded inputs
- **Ground truth obtained by executing test programs**

# Unit Test Results

**T** – computed string set

**G** – ground truth string set

**R(T)** – **T** is fully resolved

**U(T)** – **T** is unresolved (fully or partially)

| | OLSA | JSA |
|---|---|---|
| ***Complete:*** $T = G$ | 15% | 32% |
| ***Disjoint:*** $T \cap G = \emptyset \wedge U(T)$ | 53% | 34% |
| ***Incorrect:*** $T \cap G = \emptyset \wedge R(T)$ | 15% | 4% |
| ***Over-approximation:*** $G \subset T$ | 8% | 28% |
| ***Partial:*** $T \cap G \neq \emptyset$ | 7% | 1% |
| ***Under-approximation:*** $T \subset G$ | 2% | 1% |

# Reflection Analysis Results

- **Compute `java.class.forName` arguments**
- **17 DaCapo programs (20 – 700 KLOC)**

| | Programs | Runtime (sec) | Resolved |
|---|---|---|---|
| ***OSLA*** | 17 | 2 | 60% |
| ***JSA*** | 5 | 1020 | 39% |

- **Precision (over 5 programs) is similar (except 1 result)**

# Key Reasons of Imprecision

- User input (!)
- Semantics of string-manipulating functions
- Analysis of containers (e.g., arrays)
- Handling of loops and recursion calls
- Field-sensitivity

# Scalability Evaluation

- **17 DaCapo programs (3,058 KLOC combined)**
- **Only OLSA and JSA failed**
- **Different hotspot configurations**

  *Lightweight* (reflection)

  *Default* (I/O functions)

  *Heavyweight* (any string argument)
- **Results (string resolution)**

  Resolved, Partial, Unresolved

# DaCapo Results: Oracle Labs String Analyser

| Configuration | Runtime (sec) | Hotspots | Resolved | Unresolved | Partial |
|---|---|---|---|---|---|
| ***Lightweight*** | 2.13 | 318 | 60% | 9% | 30% |
| ***Default*** | 8.77 | 4,304 | 39% | 40% | 40% |
| ***Heavyweight*** | 221.94 | 156,502 | 18% | 61% | 21% |

# Conclusions

- **JSA is more precise but fails on large codebases**
- **OLSA scales well to large programs, even in extreme cases**
- **For reflection analysis, lightweight constant propagation could be as precise as regular expression generating techniques**

  Precision depends on analysed code features

# Thank you!