Precise Analysis of String Expressions

Aske Simon Christensen

Anders Møller

Michael I. Schwartzbach

University of Aarhus

BRICS

http://www.brics.dk/~amoeller/talks/strings.pdf

Motivation

Does this program always produce **syntactically correct** SQL queries?

```
public void printAddresses(int id) throws SQLException {
  Connection con = DriverManager.getConnection(“stud.db”);
  String q = "SELECT * FROM address";
  if (id != 0) q = q + "WHERE studentid=" + id;
  ResultSet rs = con.createStatement().executeQuery(q);
  while (rs.next()) {
    System.out.println(rs.getString("addr"));
  }
}
```

Motivation

How do we determine the **control flow** in programs that use **reflection** and `Class.forName`?

```
Sorter getSorter(int i) {
  String s = "algorithms.sorting.";
  switch(i) {
    case 0:  s = s+"Bubble";
             break;
    case 1:  s = s+"Merge";
             break;
    default: s = s+"Quick";
             break;
  }
  Class c = Class.forName(s);
  return (Sorter) c.newInstance();
}
```

Motivation

What are the possible outcomes of this program?

```
static String bar(int n, int k, String op) {
  if (k==0) return "";
  return op+n+"]"+bar(n-1,k-1,op)+" ";
}
static String foo(int n) {
  StringBuffer b = new StringBuffer();
  if (n<2) b.append("(");
  for (int i=0; i<n; i++) b.append("(");
  String s = bar(n-1,n/2-1,"*").trim();
  String t = bar(n-n/2,n-(n/2-1),"+").trim();
  return b.toString()+n+(s+t).replace(']',')');
}
public static void main(String args[]) {
  int n = new Random().nextInt(100);
  System.out.println(foo(n));
}
```

Goal for the Analysis

- Given a Java program,
  find for each **string expression** *E*
  an upper approximation of
  the set of **values** that *E* may have at runtime

- We want the results as **finite-state automata** (FAs)

- For a given program, we are typically interested in
  only *some* string expressions, which we call ***hotspots***

- Observation: **concatenation** is the central string operation

Use the Standard Dataflow Analysis Framework?

- The lattice of regular languages has infinite height
- Widening???

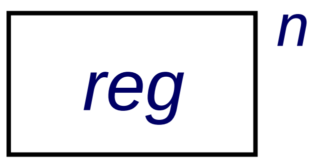
Our Approach

$$\underbrace{\text{Java} \;\rightarrow\; \text{Flow graph}}_{\text{front-end}} \;\rightarrow\; \underbrace{\text{CFGs} \;\rightarrow\; \text{FAs}}_{\text{back-end}}$$

Flow Graphs

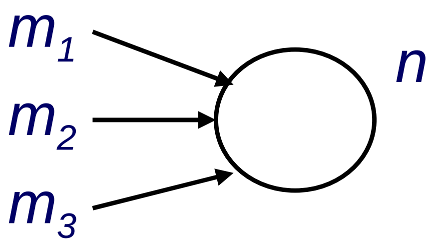
One node per expression, edges represent *def-use*:

- constant:



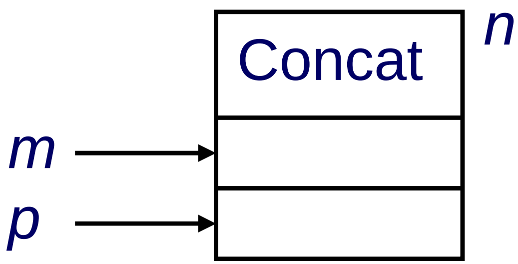
$L(n) \supseteq reg$

- join:



$L(n) \supseteq L(m_i)$

- concat:

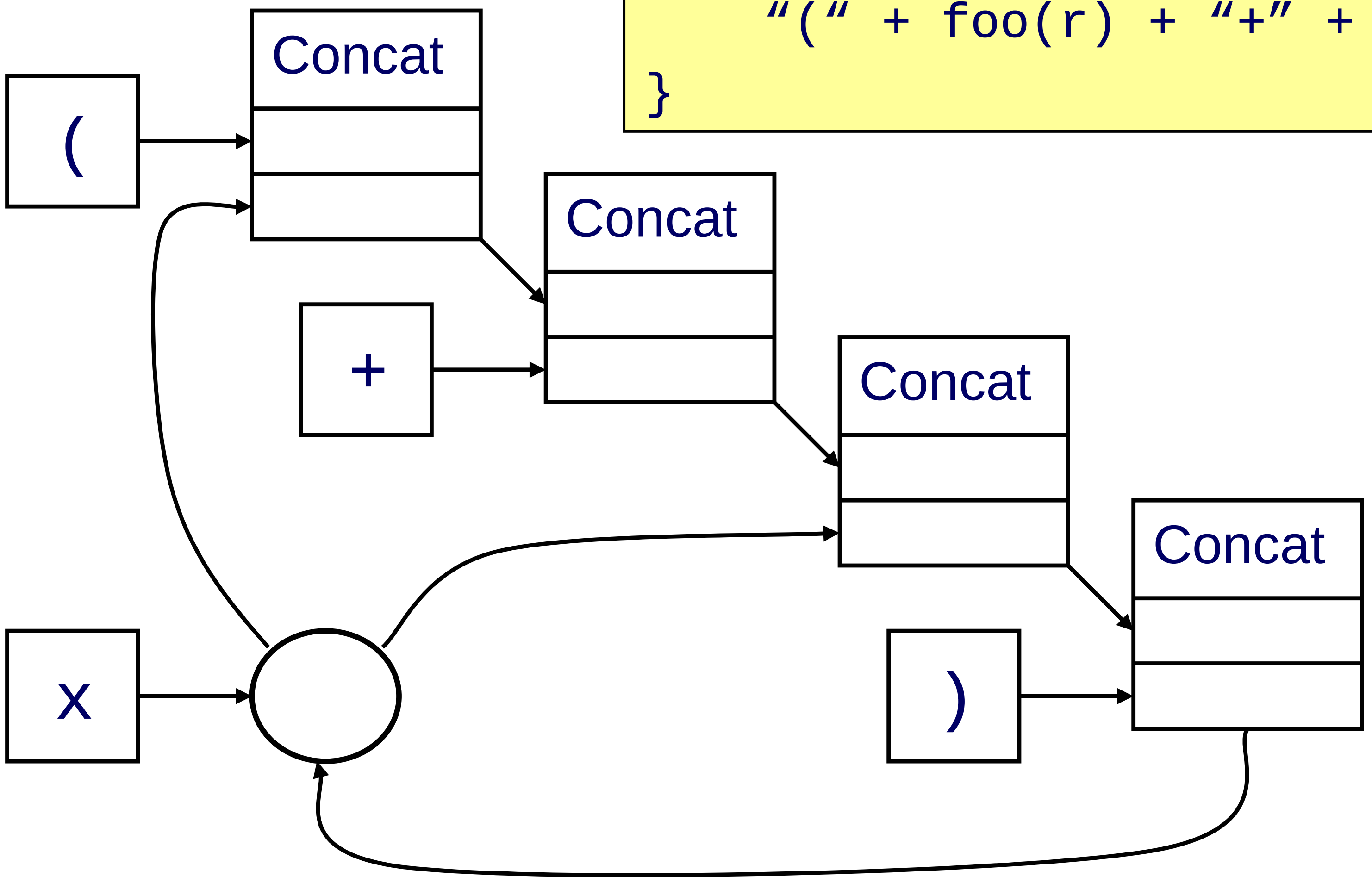


$L(n) \supseteq L(m)\ L(p)$

(We ignore other string operations for now...)

Example

```
String foo(Random r) {
  if (r.nextBoolean()) return “x”;
  else return
    “(“ + foo(r) + “+” + foo(r) + “)”;
}
```



Context-Free Grammars

Grammar $G = (\Sigma, N, P)$ with 3 kinds of productions:

- $A \rightarrow reg$ $\quad L(A) \supseteq reg$
- $A \rightarrow B$ $\quad L(A) \supseteq L(B)$
- $A \rightarrow B\ C$ $\quad L(A) \supseteq L(B)\ L(C)$

Obtaining the CFG from the flow graph is trival!

$L(A)$: Language derivable from G with A as start symbol

From CFGs to FAs

Sufficient conditions for $L(A)$ to be a regular language:

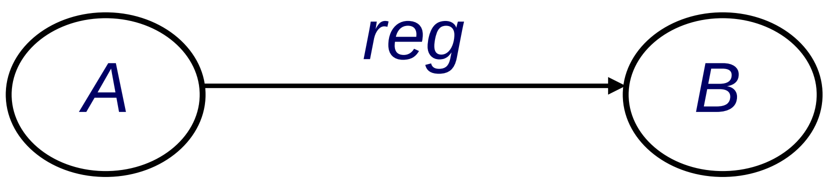
- G is *right-linear*, or
- G is *left-linear*, or
- G is *strongly regular* (every strongly connected component is left-linear or right-linear)

Right-Linear Grammars to FAs

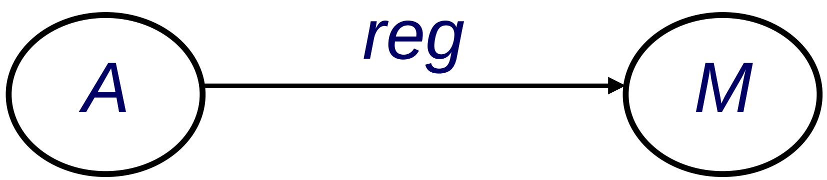
For a right-linear grammar:

- Make a state A representing $L(A)$ for each $A \in N$, plus an additional final state M

- $A \rightarrow reg\ B$



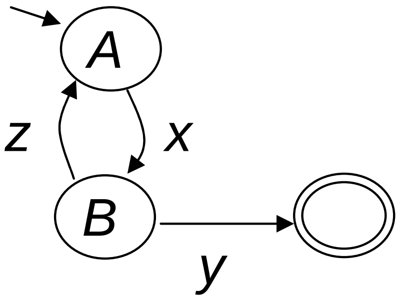
- $A \rightarrow reg$



Each hotspot corresponds to a specific **start** state

Example

$$A \rightarrow x\ B$$
$$B \rightarrow y$$
$$B \rightarrow z\ A$$



(Assume that *A* is the only hotspot here)

- for left-linear, just reverse the edges and swap start and final states

Strongly Regular Grammars to FAs

- Bottom-up traversal of the strongly connected components
- For each component, convert to FA by viewing nonterminals in "lower" components as terminals!

$A \to x\ B$

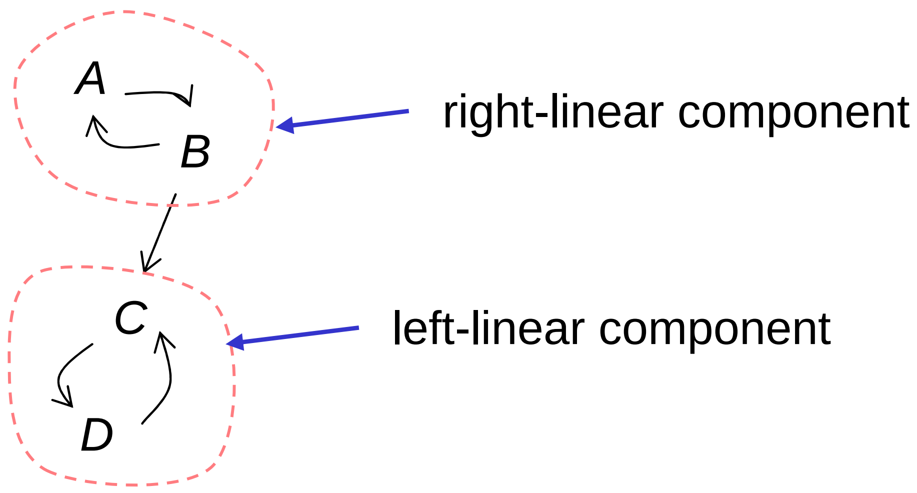
$B \to C$

$B \to z\ A$

$C \to D\ y$

$D \to z$

$D \to C\ w$



Approximation

Given a *non*-strongly regular grammar $G = (\Sigma, N, P)$, we need a grammar $G' = (\Sigma, N', P')$ such that

- G' is strongly regular
- $N \subseteq N'$
- For all $A \in N$ representing hotspots, $L_G(A) \subseteq L_{G'}(A)$
- $L_{G'}(A) \setminus L_G(A)$ is "small"

The Mohri-Nederhof Algorithm

Transforms each **non-**linear component into a **right-**linear:

- For each nonterminal *A*, add a "follows" nonterminal *A'*
- $A \rightarrow reg \quad \Rightarrow \quad A \rightarrow reg\ A'$
- $A \rightarrow B\ C \quad \Rightarrow \quad A \rightarrow B,\quad B' \rightarrow C,\quad C' \rightarrow A'$
- $A \rightarrow reg_1\ reg_2 \quad \Rightarrow \quad A \rightarrow R\ A',\quad R \rightarrow reg_1\ reg_2$
- $A \rightarrow reg\ B \quad \Rightarrow \quad A \rightarrow reg\ B,\quad B' \rightarrow A'$
- $A \rightarrow B\ reg \quad \Rightarrow \quad A \rightarrow B,\quad B' \rightarrow reg\ A'$
- if *A* is a hotspot or used in another component: add $A' \rightarrow \varepsilon$

(This elegant algorithm originates from speech recognition.)

Example

$B \to y\,A$

$C \to B\,z$

$D \to C\,A$

$E \to D\,w$

$A \to x$

$A \to E$

$B \to y\,A, \quad A \to B'$

$C \to B, \quad B' \to z\,C'$

$D \to C, \quad C' \to A, \quad A' \to D'$

$E \to D, \quad D' \to w\,E'$

$A \to x\,A'$

$A \to E, \quad E' \to A'$

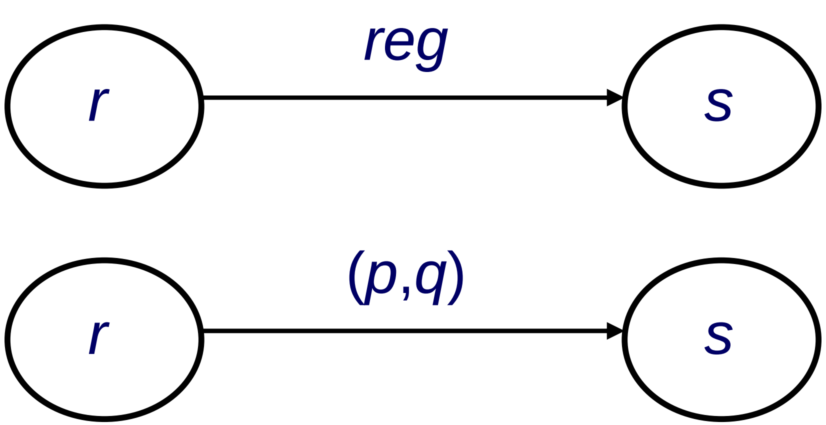
$A' \to \varepsilon$

(Assume that only *A* is a hotspot or used in another component)

Multi-Level Finite Automata

- From the strongly regular grammar,
  extracting an FA for a **single** hotspot is **easy**
- However, we typically have **many** hotspots

- An **MLFA** is an automaton with 2 kinds of transitions:



- every state is assigned a “**level**”
- *p* and *q* are states at a *lower* level than *r* and *s*
  (represent a start and a final state)

From Strongly Regular Grammar to MLFA

- Each grammar component corresponds to an MLFA level
- The (p,q) transitions are used for nonterminals that are defined in another component (at a lower level)

From MLFA to FAs

- Each **hotspot** corresponds to a grammar nonterminal

- Each grammar nonterminal corresponds to a **pair of MLFA states** (a start and a final state)

- For each state pair, we can **extract** an (N)FA
(by “unfolding” the (*p*,*q*) transitions)

- Apply **memoization** to avoid redundant computations

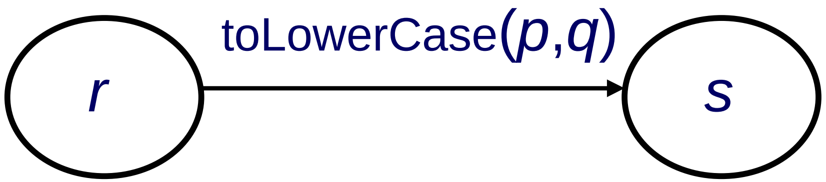
Handling Other String Operations

- `insert`, `substring`, `replace`, `trim`, `toLowerCase`, ...

- Extend the grammars with unary/binary ***operation productions***, e.g.:

  $A \rightarrow$ `toLowerCase`(B)

- Extend the MLFAs with ***operation transitions***, e.g.:



- To make this work, we must have
  1. no **cycles** containing special string operations
  2. automaton operations approximating the special string operations

Breaking Operation Cycles

- Example:

  $A \rightarrow \texttt{delete}(A)$ (`delete` deletes an unknown substring)

  $A \rightarrow A\ A$

  $A \rightarrow B$

- ***Character Set Approximation:***
  - pick an operation production $A \rightarrow \mathrm{op}(B)$ in the loop
  - replace it by $A \rightarrow (chars_{\mathrm{op}(B)})^*$
    where $chars_{\mathrm{op}(B)}$ is the set of individual characters that may appear in strings derived from $\mathrm{op}(B)$

Analysis Interface

- Implementation for Java, supporting all `String` and `StringBuffer` operations
- `analyze(string,regexp)`
  - indicates analysis points
  - no effect at runtime
- `cast(string,regexp)`
  - asserts values of strings, throws exception if violated
  - for “helping” the analysis
- `check(string,regexp)`
  - tests regular set membership, returns boolean
  - no effect for analysis

The Front End

Java → class files → Soot / Jimple → intermediate code → flow graph

- 3-address format for bytecode suitable for analysis
- intraprocedural control-flow graphs
- class hierarchy analysis for interprocedural flow
- null-pointer analysis
- alias analysis

compact representation that only considers expressions of type `String` and `StringBuffer` (and array variants) and control flow

Challenges:

- virtual method invocations, exceptions
- aliasing of mutable data (arrays, `StringBuffer`)
- escaping and intrusion (to/from unknown code)
- def-use edges (reaching definitions analysis on intermediate code)

Soundness and Complexity

- Java $\rightarrow$ flow graph
  extended CFG $\rightarrow$ strongly-regular grammar } conservative approximation
- flow graph $\rightarrow$ extended CFG
  strongly-regular grammar $\rightarrow$ MLFA $\rightarrow$ FAs } exact

- flow graph $\rightarrow$ MLFA: linear time
- MLFA $\rightarrow$ (N)FAs: exponential (worst case)

Applications

- **JWIG** / **XACT**: static analysis for extensions of Java for programming **Web services** and **XML transformations**
- Call graphs for Java programs that use **reflection** through the `Class.forName` method
- **Syntax checking** of expressions that are dynamically generated as strings (e.g. SQL / JDBC)
  - also foundation for *type* checking!

Conclusion

- Precise and efficient analysis of string expressions
- Implementation for full Java
- Convenient runtime system

- More information:
  - *Precise Analysis of String Expressions,*
    Christensen, Møller, and Schwartzbach,
    Proc. 10th International Static Analysis Symposium (SAS'03)
  - *Static Checking of Dynamically Generated Queries in Database Applications,*
    Gould, Su, Devanbu,
    Proc. 26th International Conference on Software Engineering (ICSE'04)
  - `http://www.brics.dk/JSA/`