Scaling CFL-Reachability-Based Alias Analysis: Theory and Practice

Qirun Zhang

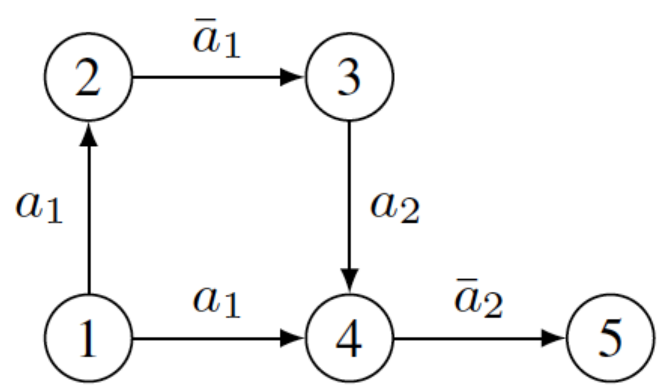
Thesis Defense

Overview

Scaling CFL-Reachability-Based Alias Analysis: Theory and Practice

Context-free language reachability

- Context-free language (CFL) reachability
  - A directed graph $G = (V, E)$
  - A context-free grammar $CFG = (\Sigma, N, P, S)$
  - Each $(u, v) \in E$ is labeled with $L(u, v) \in \Sigma$
  - Realized string $R(p)$ of a path p
    - $R(p) = \bar{a}_1 a_2 \bar{a}_2$, where $p = 2, 3, 4, 5$
  - S-path: $R(p)$ can be generated from start symbol S
    - Summary edge (u, S, v) and S-edge
- Query
  - Is there an S-path from u to v ?
  - Single-source-single-sink
  - All-pairs



Scaling CFL-Reachability-Based Alias Analysis: Theory and Practice

Alias analysis

- Alias analysis
  - Determine whether two pointer variables contain the same memory location in any execution of the program.
  - A fundamental static analysis problem.
- A client analysis
  - Buffer overflow

```
char *a, *b;
char c[10], d[11];

...
strcpy(a, b);
```

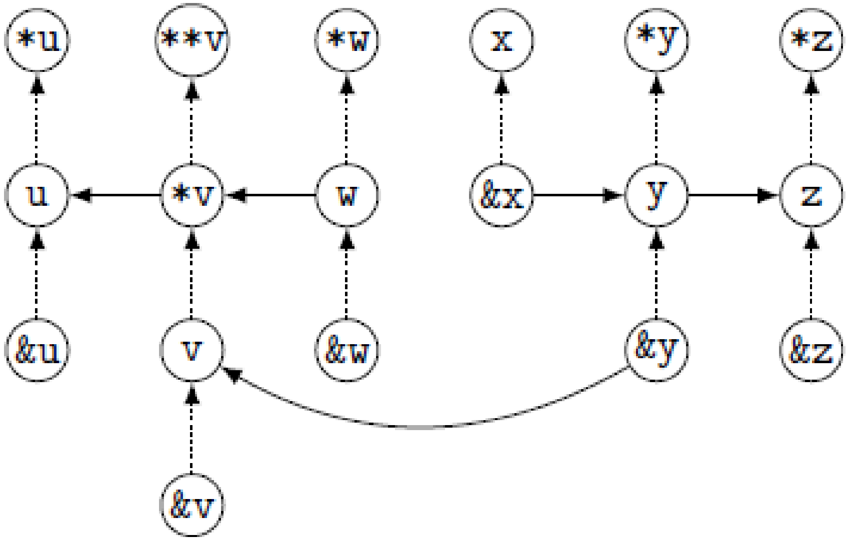
CFL-reachability-based alias analysis

- Alias analysis via CFL-reachability

```
int x;
int *u, *w, *y, *z;
int **v;

u = *v;
*v = w;
v = &y;
y = &x;
z = y;
```

(a) A code snippet.



(b) The corresponding PEG.

Main contributions

- The problem
  - CFL-reachability-based alias analysis does not scale in practice.
    - Traditional CFL algorithm has a cubic worst-case complexity
    - CFL-reachability-based alias analysis for C: 8 hours on linux
- Our contribution & take-away
  - Algorithmic
    - A set of fast CFL-reachability algorithms for alias analysis
  - Practical
    - The first subcubic implementation of CFL-reachability algorithm
      - Less than 80 seconds on the latest linux kernel

Scaling alias analysis for Java

Bidirectness	Type	Time	Space	Reference
Bidirected	Tree	$O(n \log n \log k)$	$O(n \log n)$	ESOP09
General	Graph	$O(k^3 n^3)$	$O(kn^2)$	PODS90, POPL95
General	Graph	$O(kn^3)$	$O(kn^2)$	PLDI04
General	Graph	$O(kn^3 / \log n)$	$O(kn^2)$	POPL08
Bidirected	Tree	$O(n)$	$O(n)$	Section 3.4
Bidirected	Graph	$O(n + m \log m)$	$O(n + m)$	Section 3.5
General	Graph	$O(mn + Sn)$	$O(n^2)$	Section 3.6

[ESOP09] H. Yuan and P. T. Eugster. **An efficient algorithm for solving the Dyck-CFL reachability problem on trees**

[POPL08] S. Chaudhuri. **Subcubic algorithms for recursive state machines**

[PLDI04] J. Kodumal and A. Aiken. **The set constraint/CFL reachability connection in practice**

[POPL95] T. W. Reps, S. Horwitz, and S. Sagiv. **Precise interprocedural dataflow analysis via graph reachability**

[PODS90] M. Yannakakis. **Graph-theoretic methods in database theory**

Scaling alias analysis for C

PEG Type	Time	Space	Reference
General	$O(n^3)$	$O(n^2)$	POPL08a
General	$O(n^3/\log n)$	$O(n^2)$	POPL08b
General	$O(mn + Mn)$	$O(n^2)$	Section 5.3
Well-Typed	$O(n(m + \widetilde{M}))$	$O(n^2)$	Section 5.4

[POPL08a] X. Zheng and R. Rugina **Demand-driven alias analysis for C**
[POPL08b] S. Chaudhuri. **Subcubic algorithms for recursive state machines**

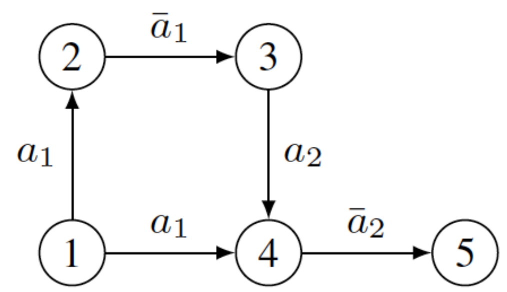
Agenda

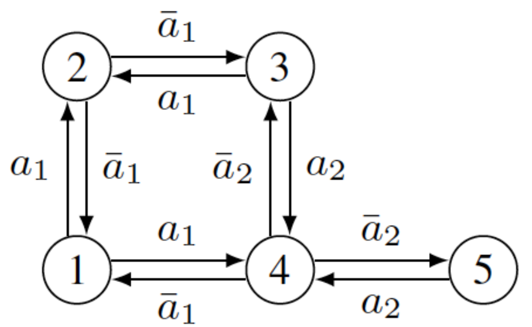
- Scaling alias analysis for Java
  - **[Ch.3]** Theory: Dyck-CFL-reachability algorithms
  - **[Ch.4]** Practice: scaling [ECOOP09], [ISSTA11]
- Scaling alias analysis for C
  - **[Ch.5]** Theory: CFL-reachability algorithms
  - **[Ch.6]** Practice: scaling [POPL08]
- Conclusion

Chapter 3
Scaling Alias Analysis for Java: Theory

Problem Definition

- Dyck-CFL-reachability
  - Dyck Grammar
    - $S \rightarrow S\ S\ \mid a_1 S\ \bar{a}_1 \mid a_k S\ \bar{a}_k\ \mid\ \varepsilon$
    - Properly matched parentheses: A and $\bar{A}$
    - Examples: ( ) ([ ]) and ([ )]
  - A bidirected graph
    - For each $L(u,v) \in A$, there is a reverse edge $L(v,u) \in \bar{A}$, and vice versa.





Scaling alias analysis for Java

Bidirectness	Type	Time	Space	Reference
Bidirected	Tree	$O(n \log n \log k)$	$O(n \log n)$	ESOP09
General	Graph	$O(k^3 n^3)$	$O(kn^2)$	PODS90, POPL95
General	Graph	$O(kn^3)$	$O(kn^2)$	PLDI04
General	Graph	$O(kn^3/\log n)$	$O(kn^2)$	POPL08
Bidirected	Tree	$O(n)$	$O(n)$	Section 3.4
Bidirected	Graph	$O(n + m \log m)$	$O(n + m)$	Section 3.5
General	Graph	$O(mn + Sn)$	$O(n^2)$	Section 3.6

[ESOP09] H. Yuan and P. T. Eugster. **An efficient algorithm for solving the Dyck-CFL reachability problem on trees**

[POPL08] S. Chaudhuri. **Subcubic algorithms for recursive state machines**

[PLDI04] J. Kodumal and A. Aiken. **The set constraint/CFL reachability connection in practice**

[POPL95] T. W. Reps, S. Horwitz, and S. Sagiv. **Precise interprocedural dataflow analysis via graph reachability**

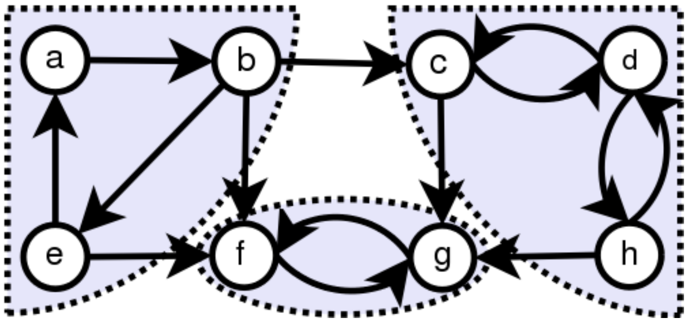
[PODS90] M. Yannakakis. **Graph-theoretic methods in database theory**

Applications of Dyck-CFL-Reachability

- Pointer Analysis
  - Java: matched field accesses (i.e., read/write)
  - C: pointer references and dereferences
- Interprocedural data flow analysis
  - Calls and returns
- Type-based flow analysis
  - In-flow and out-flow

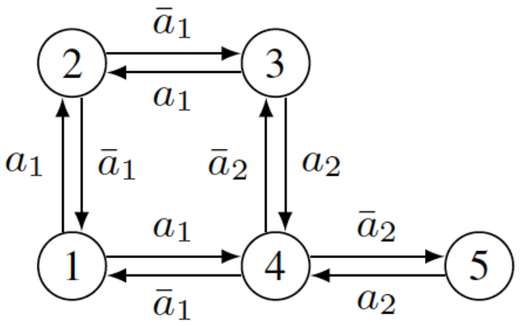
Key Insight

- An old result
  - **[PLDI98]** M. Fahndrich, J. S. Foster, Z. Su and A. Aiken. **Partial Online Cycle Elimination in Inclusion Constraint Graphs**
  - Basic idea: collapsing nodes in a strongly connected component (SCC)
- A new interpretation
  - View SCC as a binary relation C
    - i.e., $(u, v) \in C \leftrightarrow u$ and v are in the same SCC
  - C is reflexive, C is symmetric, and C is transitive
    - C is an equivalence class!
- How about Dyck-CFL-Relation?



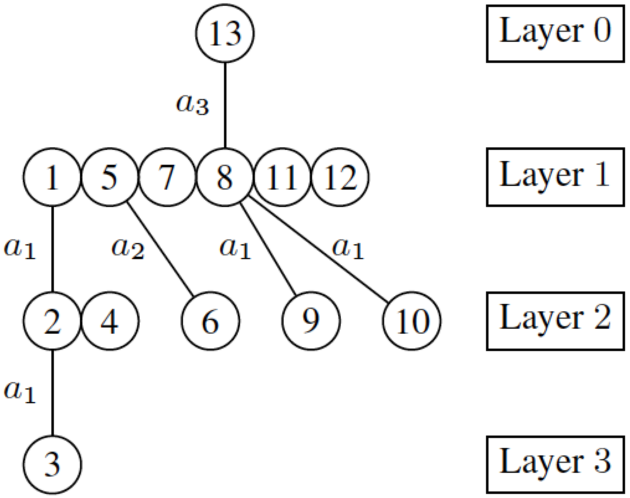
Dyck-CFL-Relation

- Consider the binary Dyck-CFL relation D on graph
  - $(u, v) \in D$ iff v is *Dyck*-reachable from u.
- Reflexivity of relation D
  - $S \rightarrow \varepsilon$
- Symmetry of relation D
- Transitivity of relation D
  - $S \rightarrow S\ S$
  - $S \rightarrow a_k\ S\ \bar{a}_k$
- D is an equivalence class as the SCC!
  - Basic idea: collapsing nodes in relation D
    - E.g., {1, 3, 5}



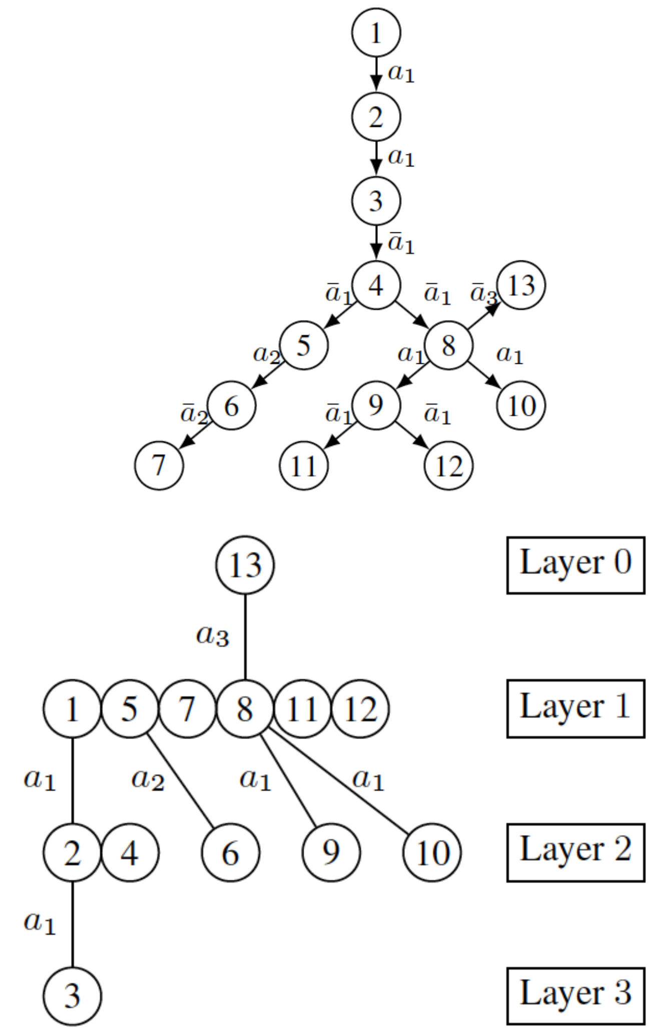
Tree Algorithm

- Input:
  - A bidirected tree
  - Dyck grammar of k kinds of parentheses
- Output:
  - Stratified set representation



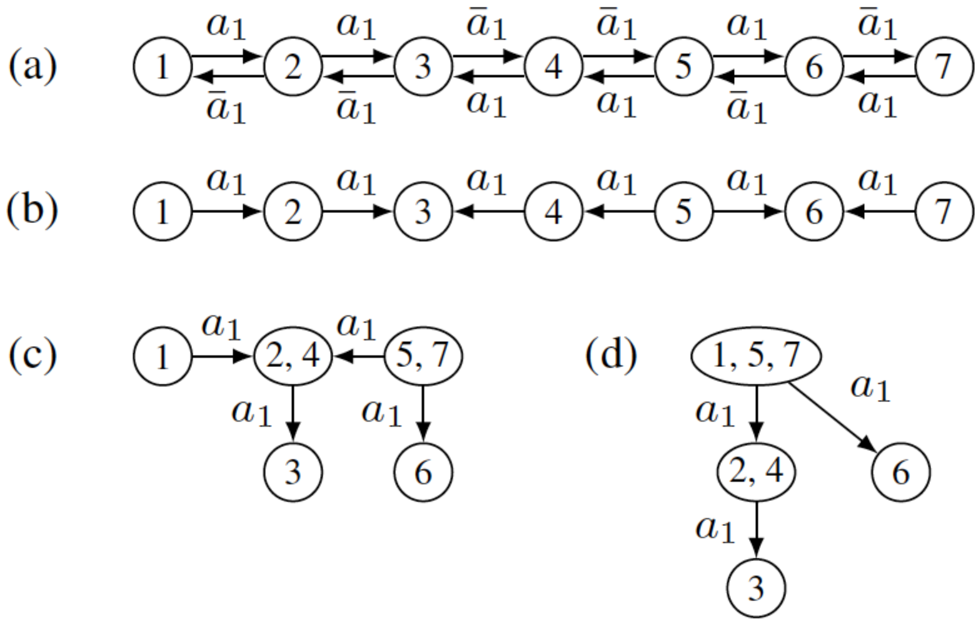
Tree Algorithm

- Major steps:
  - Pick a leaf node to start a DFS
  - Processing opening parentheses a_i
    - Allocate a new node in the lower layer
  - Processing closing parentheses $\bar{a}_i$
    - Collapse/assign the node in the upper layer w.r.t. index i
- The algorithm takes $O(n)$ time and $O(n)$ space



Graph Algorithm

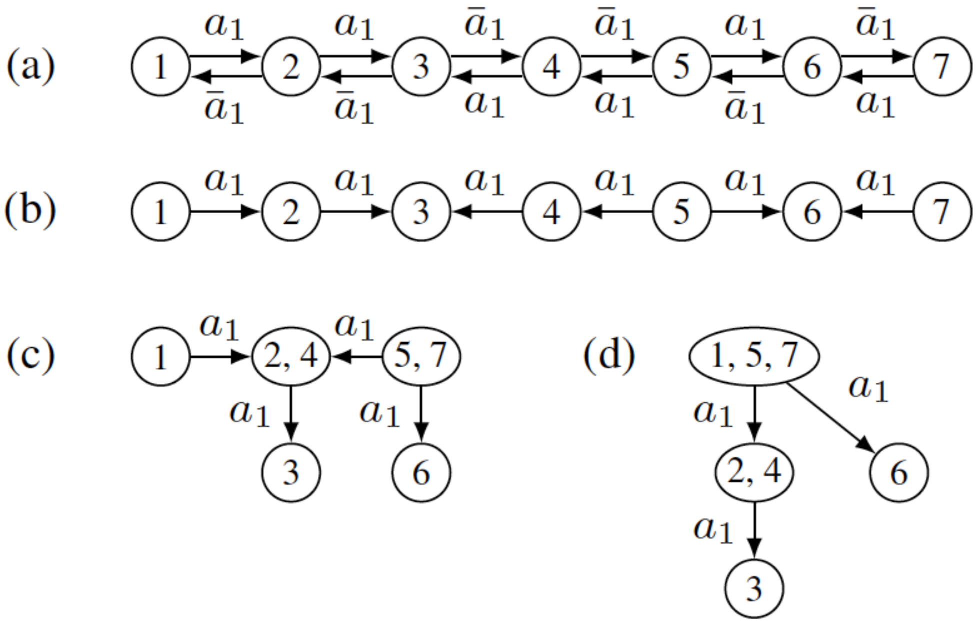
- An Dyck-path example



- Basic idea
  - Merging node z_i
    - Node number Z + index of incoming parentheses i
  - Tracking the changes of merging nodes
  - Merging the edges of merging nodes

Graph Algorithm

- An Dyck-path example



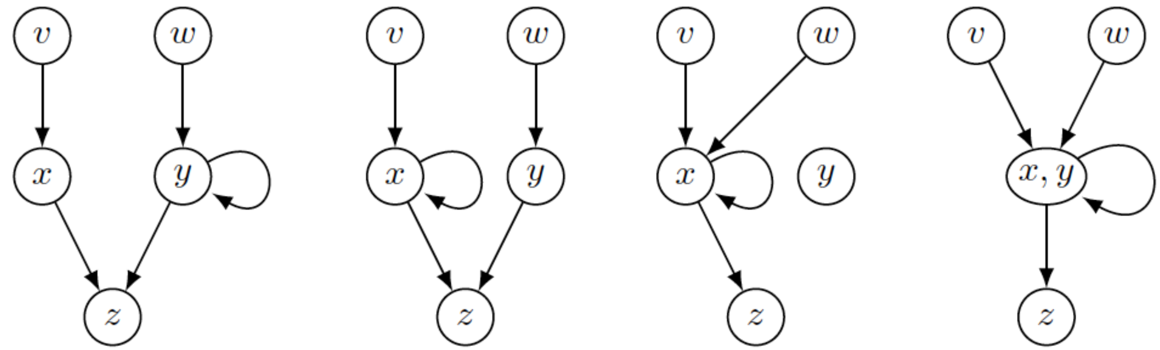
- A naïve approach
  - Pick a Merging node
  - Search edges
  - Time complexity $O(kn^2)$

Graph Algorithm: Node Tracking

- Fast doubly linked list (FDLL)
  - A DLL + a hash table
  - Insertion, deletion, and query all in *expected* $O(1)$ time
- We use three FDLLs
  - A list of merging node candidates
  - A list of nodes each merging node merges
  - A list of indexes of each node
- An example
  - $2_1 \rightarrow 1$
  - $3_1 \rightarrow 2, 4$
  - $4_1 \rightarrow 5$
  - $6_1 \rightarrow 5, 7$

Graph Algorithm: Edge Merging

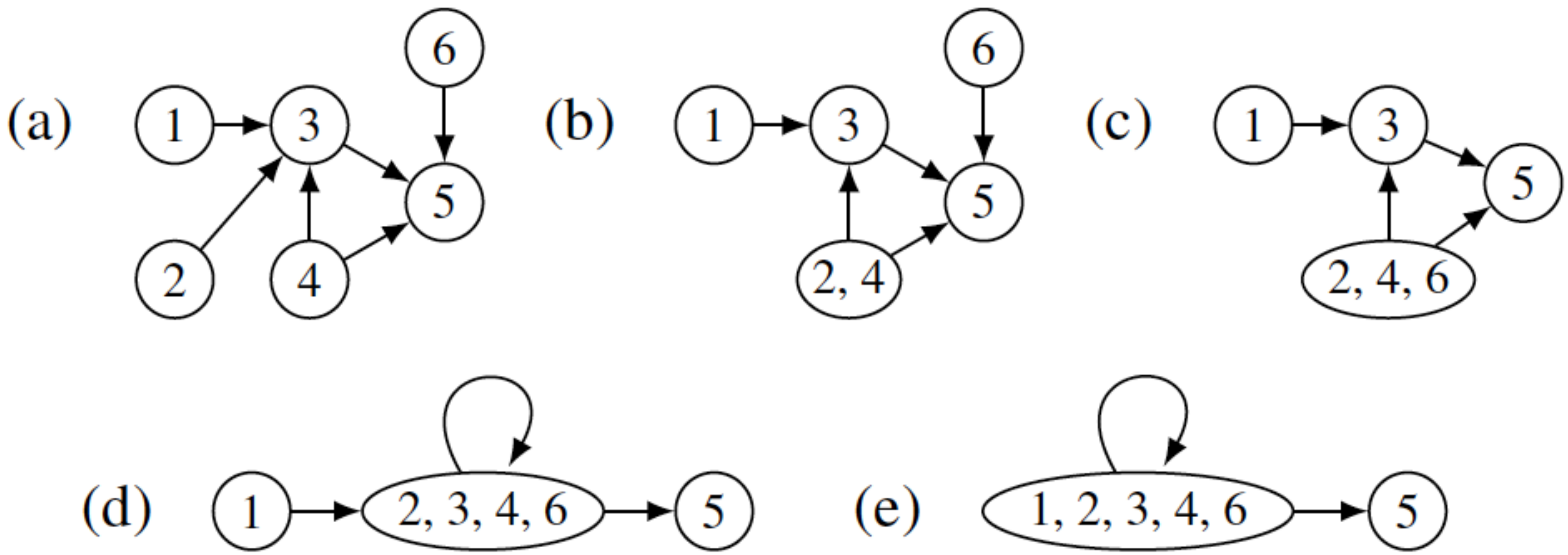
- Pick a merging node z_i
  - two nodes x and y from the FDLL of z_i
- Merging edges from y to x



(a) Before edge merging. (b) Collapsing self loops. (c) Collapsing all neighbors. (d) After edge merging.

- Special degree $\widehat{D}[x]$ of node x
  - Always merging from y to x, where $\widehat{D}[y] \leq \widehat{D}[x]$
  - Differences between $\widehat{D}[x]$ and $D[x]$

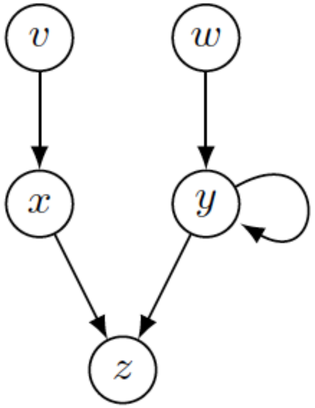
A Running Example



Complexity Analysis

- Key Claims
  - In each edge merging, an edge (x, y) is "moved"
  - For the "moved" edge (x, y), either $\widehat{D}[y]$ or $\widehat{D}[x]$ is doubled
  - $\widehat{D}[v] < 2m$ for all $v \in V$
  - An edge is "moved" for at most $2 \log 2m$ times

- The time complexity is $O(n + m \log m)$
- The space complexity is $O(n + m)$



Chapter 4
Scaling Alias Analysis for Java: Practice

Scaling an Alias Analysis for Java

- Context-sensitive flow-insensitive and field-sensitive
  - Demand-driven: single-source-single-sink Dyck-CFL-Reachabiltiy
- In our evaluation
  - Traditional Dyck-CFL-reachability algorithm vs. our algorithm
  - All-pairs Dyck-CFL-reachability
  - Context-insensitive flow-insensitive and field-sensitive

[ECOOP09] G. Xu, A. Rountev, M. Sridharan **Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis.**

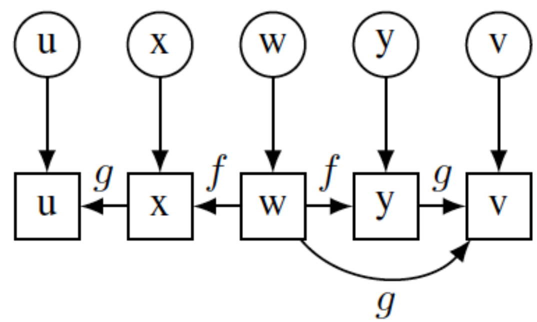
[ISSTA11] D. Yan, G. Xu, A. Rountev **Demand-driven context-sensitive alias analysis for Java**

Scaling an Alias Analysis for Java

- Symbolic points-to graph
  - Nodes: abstract heap locations
  - Edges: field accesses (read/write)

```
x = w.f;
w.f = y;
u = x.g;
v = y.g;
v = w.g;
```

(a) A code snippet.



(b) Its SPG.

- The grammar
  - $memAlias \rightarrow \bar{f}_1\ memAlias\ f_1 \mid \cdots \mid \bar{f}_k\ memAlias\ f_k$
    $\mid memAlias\ \ memAlias \mid \varepsilon$

Experiments

- Evaluation benchmarks
  - DaCapo-2006-10-MR2 and DaCapo-9.12bach

Benchmark	SPG				Time (s)		Memory (MB)	
	#Nodes	#Edges	#S-pair	#para	CFL	FAST-DYCK	CFL	FAST-DYCK
antlr	16735	13878	19385	1087	37.42	**0.041**	29.68	**20.21**
bloat	20320	16224	23080	1197	43.09	**0.048**	35.09	**23.89**
chart	44584	36329	50670	2948	253.06	**0.119**	76.75	**52.02**
eclipse	17527	14411	20335	1182	42.26	**0.042**	30.97	**21.19**
fop	39977	31515	45837	2724	219.53	**0.101**	67.99	**46.08**
hsqldb	15015	12693	17615	998	33.39	**0.038**	27.10	**18.22**
jython	21615	17381	24487	1240	49.57	**0.052**	37.20	**25.32**
luindex	16098	13336	18716	1071	35.15	**0.040**	28.64	**19.45**
lusearch	17003	14195	19911	1117	40.22	**0.043**	30.34	**20.73**
pmd	18167	14958	20843	1168	40.28	**0.046**	32.00	**21.90**
xalan	15030	12645	17608	996	32.93	**0.038**	26.93	**18.21**
batik	40273	32052	46225	2565	206.50	**0.100**	68.77	**46.60**
eclipse	37531	31889	54471	2221	366.39	**0.103**	70.82	**44.54**
jython	63516	49005	85552	2855	947.49	**0.163**	112.14	**72.18**
sunflow	39321	31339	45161	2484	196.23	**0.096**	67.22	**45.57**
tomcat	45966	37338	63414	3013	622.36	**0.124**	83.98	**53.56**

Discussions

- Understanding the performance

Benchmark	SPG				Time (s)		Memory (MB)	
	#Nodes	#Edges	#S-pair	#para	CFL	FAST-DYCK	CFL	FAST-DYCK
antlr	16735	13878	19385	1087	37.42	**0.041**	29.68	**20.21**
bloat	20320	16224	23080	1197	43.09	**0.048**	35.09	**23.89**
chart	44584	36329	50670	2948	253.06	**0.119**	76.75	**52.02**
eclipse	17527	14411	20335	1182	42.26	**0.042**	30.97	**21.19**
fop	39977	31515	45837	2724	219.53	**0.101**	67.99	**46.08**
hsqldb	15015	12693	17615	998	33.39	**0.038**	27.10	**18.22**
jython	21615	17381	24487	1240	49.57	**0.052**	37.20	**25.32**
luindex	16098	13336	18716	1071	35.15	**0.040**	28.64	**19.45**
lusearch	17003	14195	19911	1117	40.22	**0.043**	30.34	**20.73**
pmd	18167	14958	20843	1168	40.28	**0.046**	32.00	**21.90**
xalan	15030	12645	17608	996	32.93	**0.038**	26.93	**18.21**
batik	40273	32052	46225	2565	206.50	**0.100**	68.77	**46.60**
eclipse	37531	31889	54471	2221	366.39	**0.103**	70.82	**44.54**
jython	63516	49005	85552	2855	947.49	**0.163**	112.14	**72.18**
sunflow	39321	31339	45161	2484	196.23	**0.096**	67.22	**45.57**
tomcat	45966	37338	63414	3013	622.36	**0.124**	83.98	**53.56**

Discussions

- Interpreting the client analysis

Benchmark	SPG				Time (s)		Memory (MB)	
	#Nodes	#Edges	#S-pair	#para	CFL	FAST-DYCK	CFL	FAST-DYCK
antlr	16735	13878	19385	1087	37.42	**0.041**	29.68	**20.21**
bloat	20320	16224	23080	1197	43.09	**0.048**	35.09	**23.89**
chart	44584	36329	50670	2948	253.06	**0.119**	76.75	**52.02**
eclipse	17527	14411	20335	1182	42.26	**0.042**	30.97	**21.19**
fop	39977	31515	45837	2724	219.53	**0.101**	67.99	**46.08**
hsqldb	15015	12693	17615	998	33.39	**0.038**	27.10	**18.22**
jython	21615	17381	24487	1240	49.57	**0.052**	37.20	**25.32**
luindex	16098	13336	18716	1071	35.15	**0.040**	28.64	**19.45**
lusearch	17003	14195	19911	1117	40.22	**0.043**	30.34	**20.73**
pmd	18167	14958	20843	1168	40.28	**0.046**	32.00	**21.90**
xalan	15030	12645	17608	996	32.93	**0.038**	26.93	**18.21**
batik	40273	32052	46225	2565	206.50	**0.100**	68.77	**46.60**
eclipse	37531	31889	54471	2221	366.39	**0.103**	70.82	**44.54**
jython	63516	49005	85552	2855	947.49	**0.163**	112.14	**72.18**
sunflow	39321	31339	45161	2484	196.23	**0.096**	67.22	**45.57**
tomcat	45966	37338	63414	3013	622.36	**0.124**	83.98	**53.56**

Discussions

- Demand-driven vs. exhausted analysis

Benchmark	SPG				Time (s)		Memory (MB)	
	#Nodes	#Edges	#S-pair	#para	CFL	FAST-DYCK	CFL	FAST-DYCK
antlr	16735	13878	19385	1087	37.42	**0.041**	29.68	**20.21**
bloat	20320	16224	23080	1197	43.09	**0.048**	35.09	**23.89**
chart	44584	36329	50670	2948	253.06	**0.119**	76.75	**52.02**
eclipse	17527	14411	20335	1182	42.26	**0.042**	30.97	**21.19**
fop	39977	31515	45837	2724	219.53	**0.101**	67.99	**46.08**
hsqldb	15015	12693	17615	998	33.39	**0.038**	27.10	**18.22**
jython	21615	17381	24487	1240	49.57	**0.052**	37.20	**25.32**
luindex	16098	13336	18716	1071	35.15	**0.040**	28.64	**19.45**
lusearch	17003	14195	19911	1117	40.22	**0.043**	30.34	**20.73**
pmd	18167	14958	20843	1168	40.28	**0.046**	32.00	**21.90**
xalan	15030	12645	17608	996	32.93	**0.038**	26.93	**18.21**
batik	40273	32052	46225	2565	206.50	**0.100**	68.77	**46.60**
eclipse	37531	31889	54471	2221	366.39	**0.103**	70.82	**44.54**
jython	63516	49005	85552	2855	947.49	**0.163**	112.14	**72.18**
sunflow	39321	31339	45161	2484	196.23	**0.096**	67.22	**45.57**
tomcat	45966	37338	63414	3013	622.36	**0.124**	83.98	**53.56**

Chapter 5

Scaling Alias Analysis for C: Theory

Scaling alias analysis for C

PEG Type	Time	Space	Reference
General	$O(n^3)$	$O(n^2)$	POPL08a
General	$O(n^3/\log n)$	$O(n^2)$	POPL08b
General	$O(mn + Mn)$	$O(n^2)$	Section 5.3
Well-Typed	$O(n(m + \widetilde{M}))$	$O(n^2)$	Section 5.4

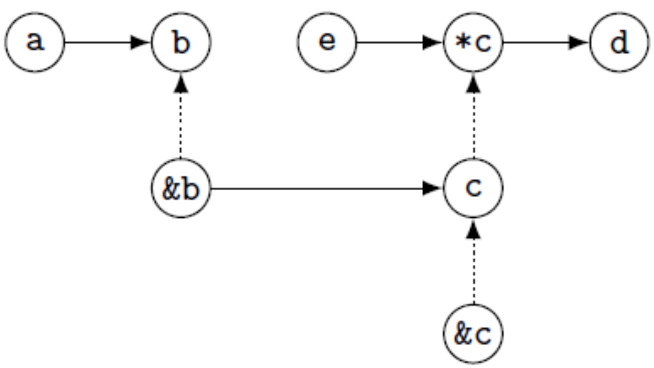
[POPL08a] X. Zheng and R. Rugina **Demand-driven alias analysis for C**
[POPL08b] S. Chaudhuri. **Subcubic algorithms for recursive state machines**

CFL-Reachability-Based Alias Analysis for C

- Precision is equivalent to Andersen-style analysis
  - **[POPL08]** X. Zheng and R. Rugina. **Demand-driven alias analysis for C.**
- The pointer expression graph (PEG)
  - A-edges and D-edges

```
int *a,*b;
int *c,*d,*e;
  b = a;
  c = &b;
  d = *c;
  *c = e;
```

(a) A code snippet.



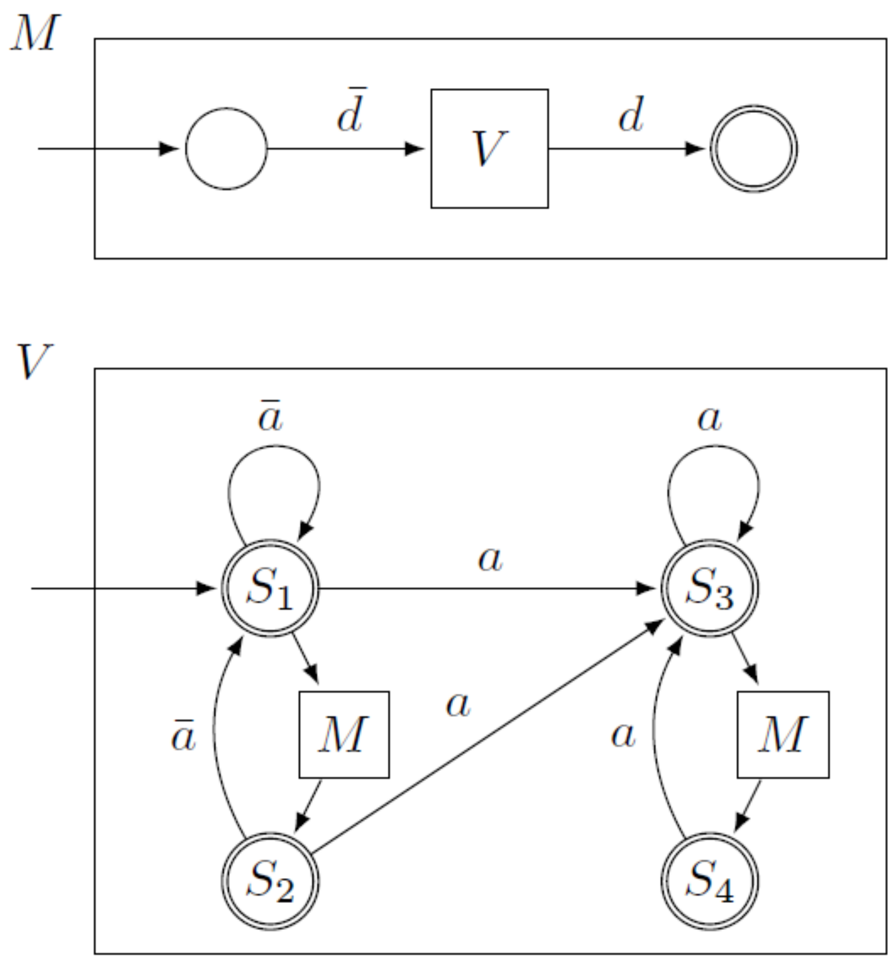
(b) Its PEG.

- The CFG

$$M ::= \bar{d}\ V\ d \tag{1}$$

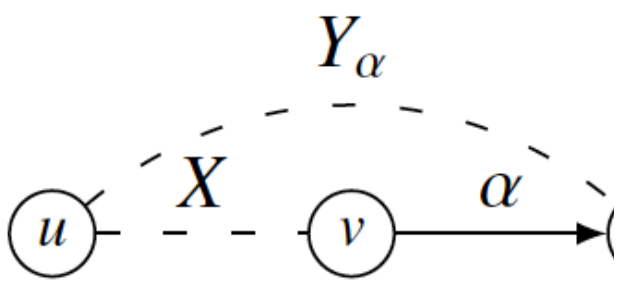
$$V ::= (M?\ \bar{a})^*\ M?\ (a\ M?)^* \tag{2}$$

The recursive state machine representation



Traditional Approach

- The traditional CFL-Reachability
  - CFG normal form
    - $A \rightarrow BC$ and $A \rightarrow B$
    - 15 rules
  - Adding new summary edge

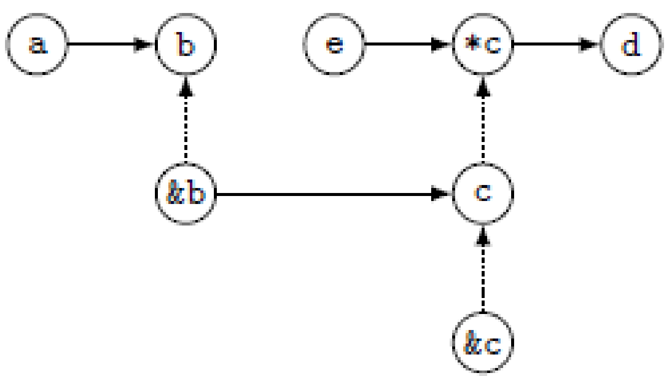


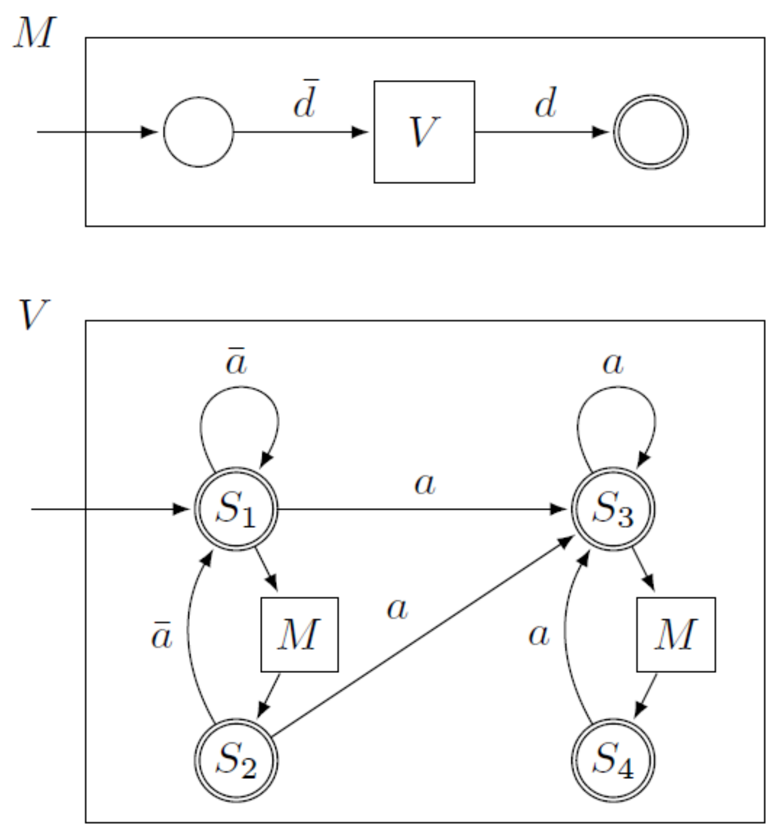
  - Cubic time worst-case comple

$$
\begin{array}{rcl}
M & ::= & DV\ d \\
DV & ::= & \bar{d}\ V \\
S_1 & ::= & S_1\ \bar{a} \\
S_2 & ::= & S_1\ M \\
S_1 & ::= & S_2\ \bar{a} \\
S_3 & ::= & S_2\ a \\
S_3 & ::= & S_1\ a \\
S_3 & ::= & S_3\ a \\
S_4 & ::= & S_3\ M \\
S_3 & ::= & S_4\ a \\
V & ::= & S_1 \\
V & ::= & S_2 \\
V & ::= & S_3 \\
V & ::= & S_4 \\
S_1 & ::= & \varepsilon
\end{array}
$$

Key Novelty

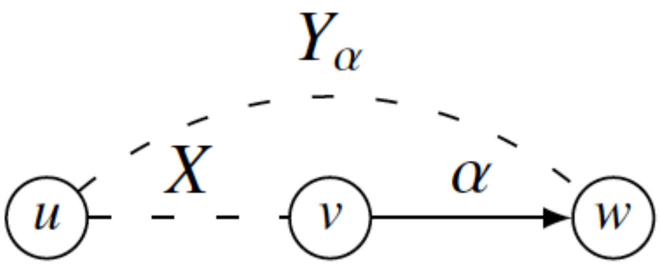
- Observed the same-layer-regular property
  - $\bar{a}^* a^*$-reachability





Key Novelty

- Propagating reachability information only among:
  - Original edges in the graph
  - Summary edges denoting memory alias (M)



- $O((m + M)n)$ worst-case time complexity
  - M and m are very sparse in practice!

Alias properties

- M and V are mutually dependent

$$M ::= \bar{d}\ V\ d \quad (1)$$
$$V ::= (M?\ \bar{a})^*\ M?\ (a\ M?)^* \quad (2)$$

- Fact 1: Each M-path is generated by prepending a $\bar{d}$-edge and appending a d-edge to a V-edge.

- Fact 2: Each M-path is generated by a path whose $R(p) = \bar{a}^* a^*$, injected with zero or more non-consecutive M-paths.

Alias properties

- M and V are mutually dependent

$$M \;::=\; \bar{d}\; V\; d \qquad (1)$$

$$V \;::=\; (M?\;\bar{a})^*\; M?\; (a\; M?)^* \qquad (2)$$

- Unique positions of M in V

(a) V-path

$$\boxed{1} \xrightarrow{\bar{a}} \cdots \xrightarrow{\bar{a}} \boxed{2} \xrightarrow{\bar{a}} \cdots \xrightarrow{\bar{a}} \boxed{3} \xrightarrow{a} \cdots \xrightarrow{a} \boxed{4} \xrightarrow{a} \cdots \xrightarrow{a} \boxed{5}$$

(b) V-path represented by a-edges

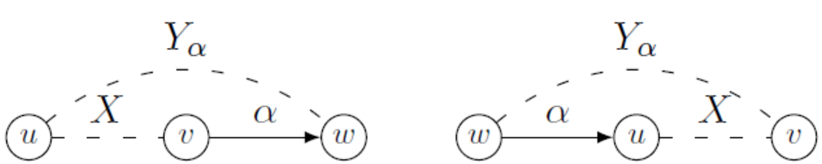
$$\boxed{1} \xleftarrow{a} \cdots \xleftarrow{a} \boxed{2} \xleftarrow{a} \cdots \xleftarrow{a} \boxed{3} \xrightarrow{a} \cdots \xrightarrow{a} \boxed{4} \xrightarrow{a} \cdots \xrightarrow{a} \boxed{5}$$

Propagating reachability summaries

- Unique positions of M in V

- Unique positions of M in V
- Due to bidirectness of the graph, positions 4 and 5 are symmetric to positions 2 and 1
- Two-phase propagation

- Due to bidirectness of the graph, positions 4 and 5 are symmetric to positions 2 and 1
- Two-phase propagation

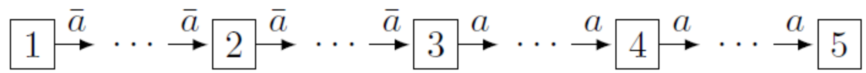


(a). Phase one. (b) Phase two.

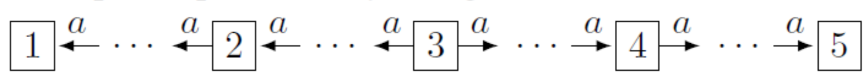
Two-phase propagation

- Unique positions of M in V

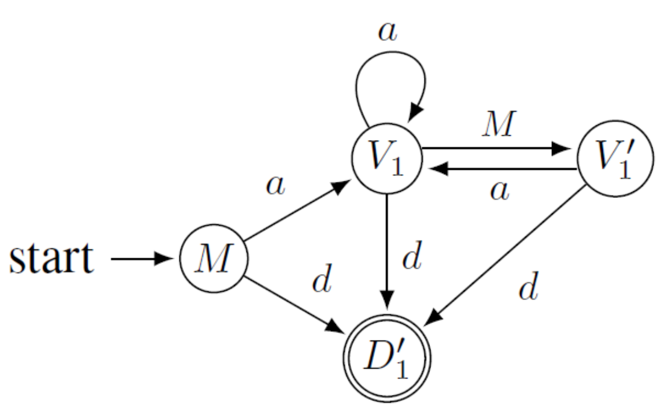
(a) V-path



(b) V-path represented by a-edges



- Phase one propagation, starting at positions 3, 4, 5.

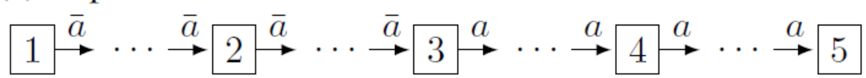


Input State	a	d	M
M	V_1	D_1'	$\times$
V_1	V_1	D_1'	V_1'
V_1'	V_1	D_1'	$\times$

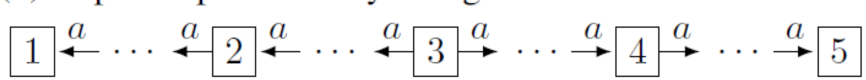
Two-phase propagation

- Unique positions of M in V

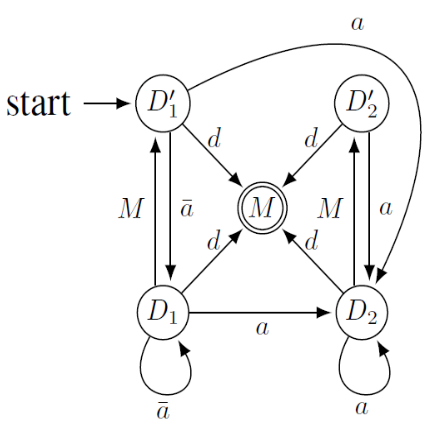
(a) V-path



(b) V-path represented by a-edges



- Phase two propagation.



Input State	a	$\bar{a}$	d	M
D_1'	D_2	D_1	M	$\times$
D_1	D_2	D_1	M	D_1'
D_2'	D_2	$\times$	M	$\times$
D_2	D_2	$\times$	M	D_2'

Complexity analysis

- For each summary edge (u, X, v), our algorithm searches:
  - a-, $\bar{a}$- , d- , $\bar{d}$-edges: node degree Δ in the original graph
  - M-edges: node degree of M-edges in the final graph

- Total steps

$$\Sigma_{(u,v)}\Delta_v = \Sigma_u(\Sigma_v\Delta_v) = O((m+M)\,n)$$

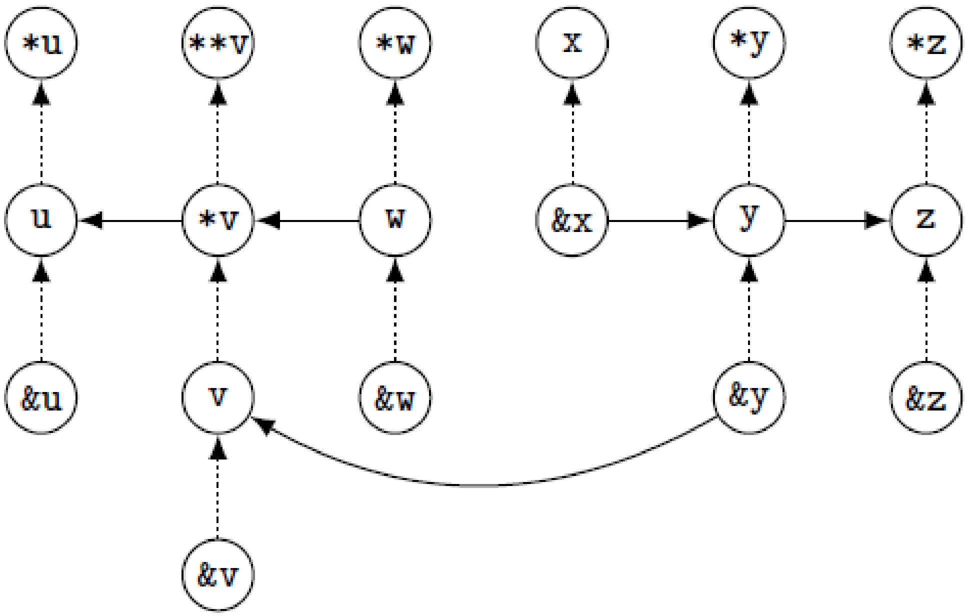
Well-typed alias analysis

- Well-typeness
  - Type qualifiers of each assignments are compatible
- Example

```
int x;
int *u, *w, *y, *z;
int **v;

u = *v;
*v = w;
v = &y;
y = &x;
z = y;
```

(a) A code snippet.



(b) The corresponding PEG.

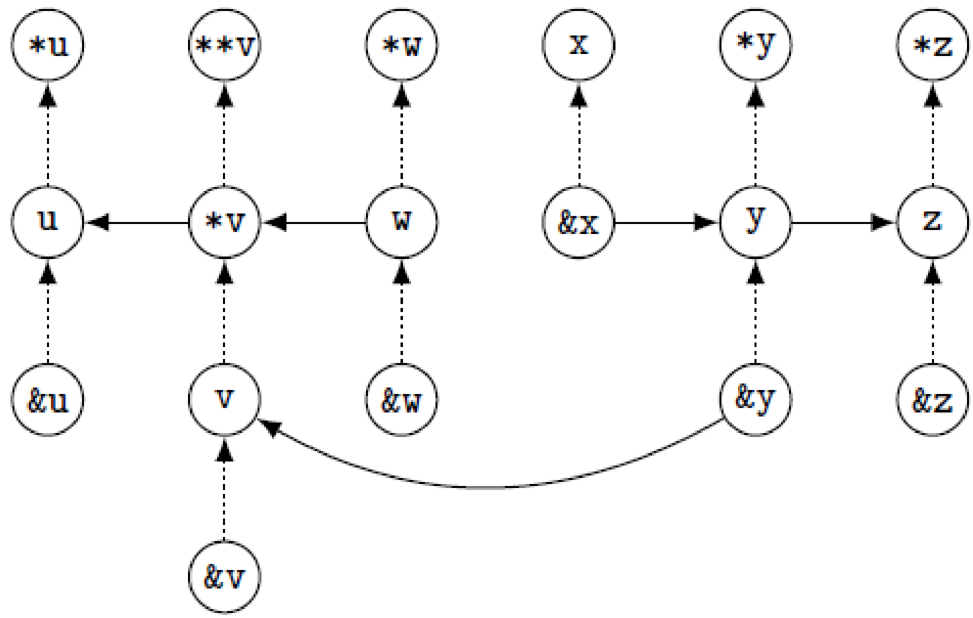
Basic idea

- A bottom-up approach

```
int x;
int *u, *w, *y, *z;
int **v;

u = *v;
*v = w;
v = &y;
y = &x;
z = y;
```

(a) A code snippet.



(b) The corresponding PEG.

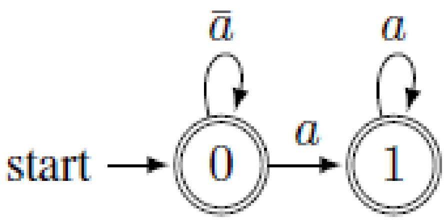
Computing bottom-layer reachability

- With no D-edges, with only A-edges
- Same-layer-regular property
  - A regular language reachability $\bar{a}^*a^*$
- Formulated as a dynamic path problem [SODA90]
  - A reachability matrix $m_{u_r v_q}$
  - A spanning tree $T(u_r)$ associated with each node
  - Amortized $O(n)$ time for each edge insertion

[SODA90] A. L. Buchsbaum, P. C. Kanellakis, and J. S. Vitter. **A data structure for arc insertion and regular path finding.**

Computing bottom-layer reachability

- Finite state automata representation



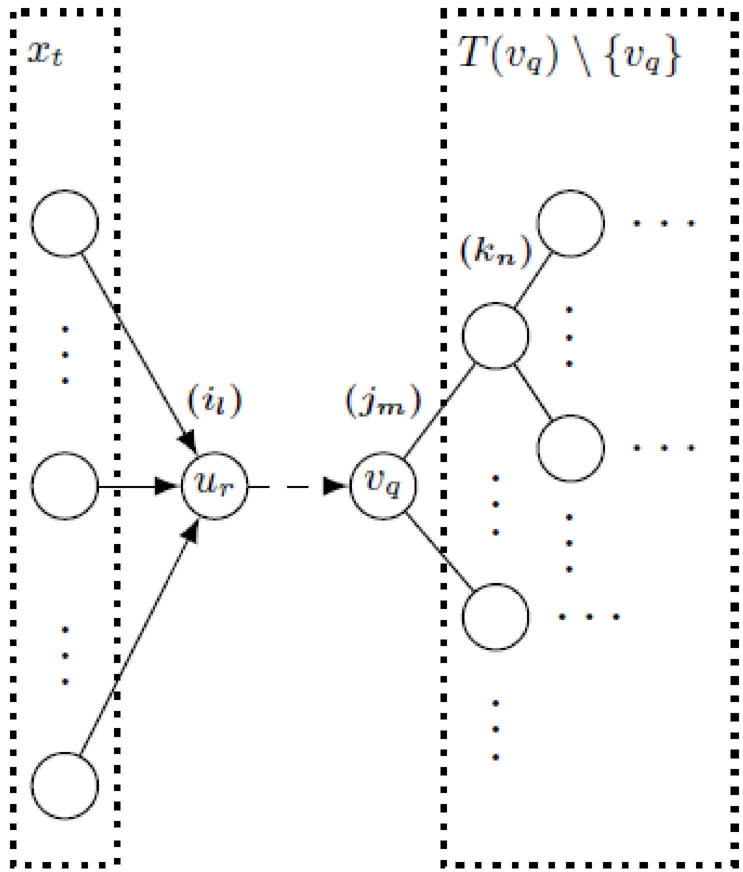
State \ Input	a	$\bar{a}$
0	1	0
1	1	$\times$

- Processing a-edges and $\bar{a}$-edges

Edge inserted	$m_{u_r v_q}$ updated	$T(x_t)$ affected
(u, a, v)	$m_{u_1 v_1}$	$T(x_0)$
	$m_{u_0 v_1}$	$T(x_0)$
	$m_{u_1 v_1}$	$T(x_1)$
$(u, \bar{a}, v)$	$m_{u_0 v_0}$	$T(x_0)$

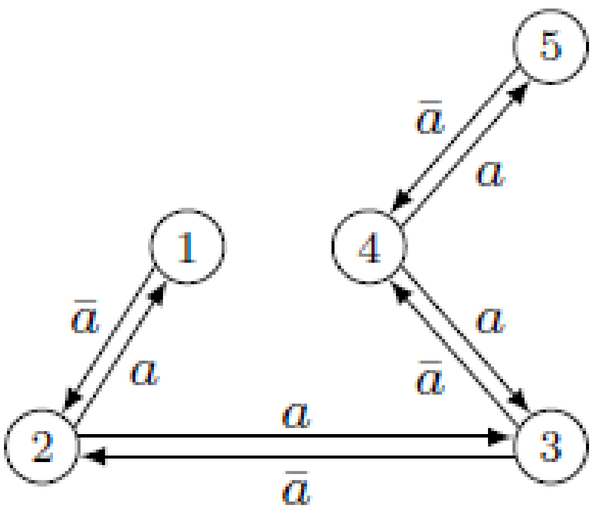
Computing bottom-layer reachability

- Updating reachability information

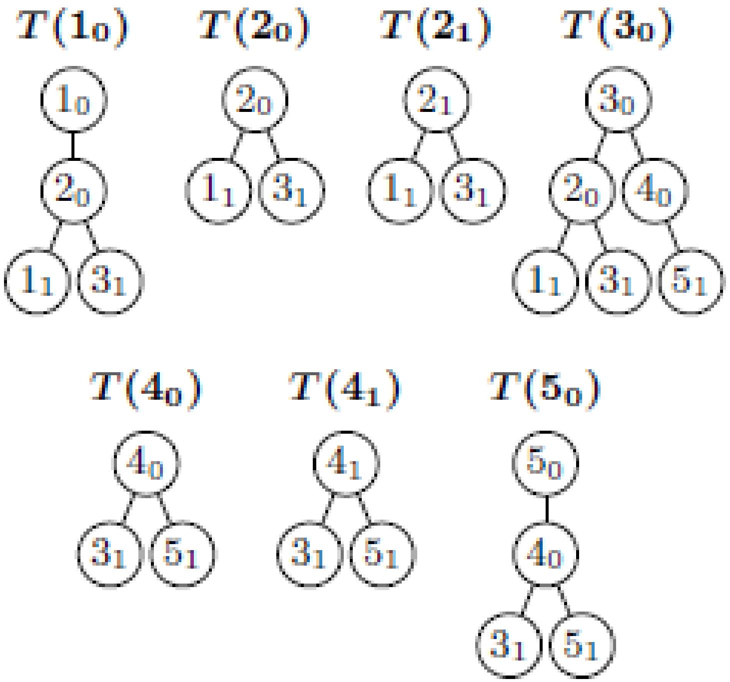


An example

- Before edge insertion



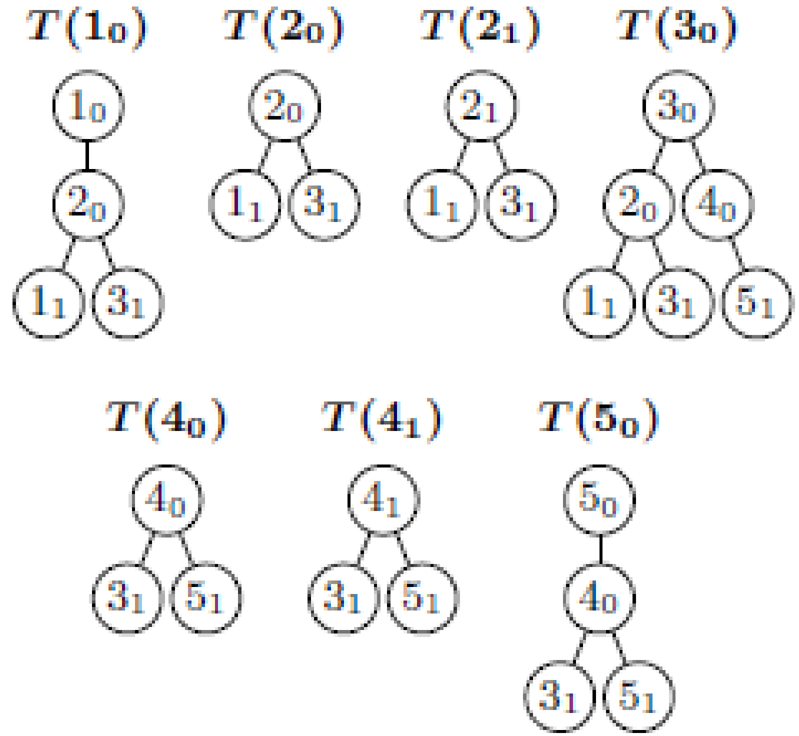
(a) The input PEG.

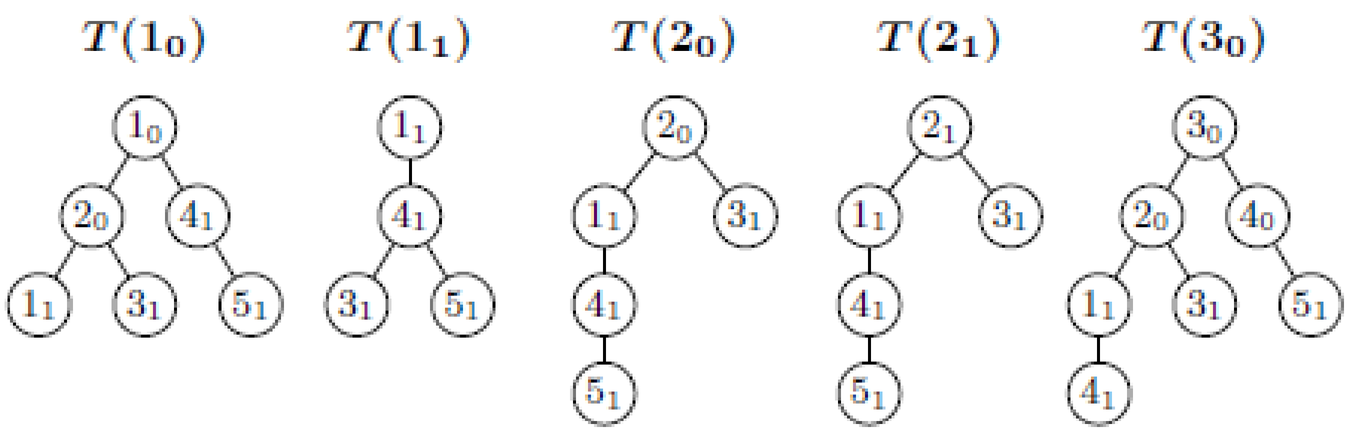


(b) The reachability spanning trees.

An example

- Inserting $(1, a, 4)$
- Updating $m_{1_0 4_1}$ and $m_{1_1 4_1}$
- For each x_t , the pruned cc

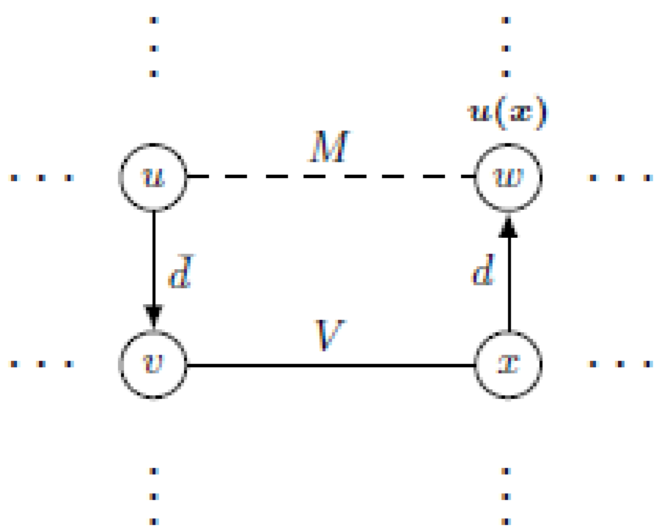




(a) New reachability trees.

Computing reachability among layers

- Layers connected via d-edges and $\bar{d}$-edges
- Assume layer $l-1$ is computed, consider layer l
- For each (x, d, w), search v at layer $l-1$
  - Update $m_{u_q w_r}$ using $m_{v_q x_r}$



Complexity analysis (sketch)

- a-, $\bar{a}$-, d- and $\bar{d}$-edges are original edges
- For each a-, $\bar{a}$- edges on layer l
  - Regular language reachability ($\bar{a}^* a^*$)
  - Each edge insertion in amortized $O(n)$ time [SODA90]
- For each d-, $\bar{d}$- edges (e.g., (x, d, w))
  - Update $m_{u_q w_r}$ using $m_{v_q x_r}$
  - Takes $O(n)$ time
- The whole algorithm takes $O(n(m + \widetilde{M}))$ time

[SODA90] A. L. Buchsbaum, P. C. Kanellakis, and J. S. Vitter. **A data structure for arc insertion and regular path finding.**

Chapter 6

Scaling Alias Analysis for C: Practice

Scaling an Alias Analysis for C

- Context-insensitive flow-insensitive and field-insensitive
  - Demand-driven: single-source-single-sink CFL-Reachabiltiy
- In our evaluation
  - Traditional CFL-reachability algorithms vs. our algorithm
  - All-pairs CFL-reachability
  - Context-insensitive flow-insensitive and field-insensitive

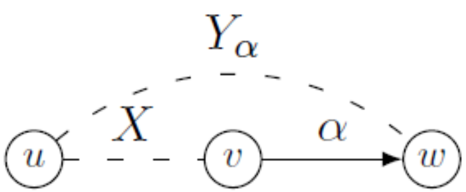
[POPL08] X. Zheng and R. Rugina **Demand-driven alias analysis for C**

Implementation

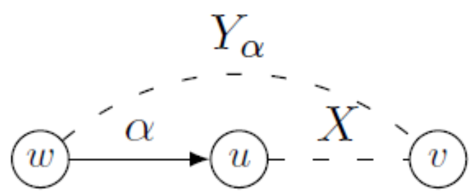
- Front end: gcc-4.6.3
- Applying the Four-Russains' trick:
  - Known technique in subcubic CFL-reachability algorithm, assuming RAM model with word size of $\theta(\log n)$
  - Store n elements using fast set in $O(\lceil n \log n \rceil)$ words
  - Set difference $Diff(X, Y)$ in $O(\lceil n \log n \rceil + v)$ time
  - Directly applicable to our algorithm
- Connect component decomposition

Applying the Four-Russians' Trick

- Propagating reachability summary



(a). Phase one.



(b) Phase two.

- Modifications:

foreach $w \in \textsc{Out}(v, \alpha)$ **do**
if $(u, Y_\alpha, w) \notin G$ **then**

becomes

foreach $w \in \mathit{diff}(\textsc{Out}(v, \alpha), \textsc{Out}(u, Y_\alpha))$ **do**

Results

- Benchmark programs

Program	SLOC	#procedures	PEG	
			#nodes	#edges
Gdb-7.5.1	1,828K	10,536	649,564	1,219,788
Emacs-24.2	254K	3,626	687,691	1,290,200
Insight-6.8-1a	1,742K	10,507	787,289	1,494,168
Gimp-2.8.4	702K	17,842	872,681	1,675,546
Ghostscript-9.07	851K	12,211	1,198,753	2,368,086
Wine-1.5.25	2,306K	70,923	4,652,983	8,472,950
Linux-3.8.2	10,601K	138,095	12,807,645	23,398,670

Connected component decomposition

- Connected component information

Program	#CC	PEG in Proc.		PEG in CC	
		Max.	Avg.	Max.	Avg.
Gdb-7.5.1	66,154	5,162	61.65	4,350	9.82
Emacs-24.2	67,608	22,654	189.66	15,690	10.17
Insight-6.8-1a	75,373	6,273	74.93	4,350	10.45
Gimp-2.8.4	79,572	3,599	48.91	2,693	10.97
Ghostscript-9.07	87,768	8,573	98.17	7,025	13.66
Wine-1.5.25	537,370	20,008	65.61	5,106	8.66
Linux-3.8.2	1,449,718	7,305	92.75	4,755	8.83

Results

- Performance comparisons

Program	Time (seconds)			Memory (MBs)		
	Cubic	Subcubic	Our	Cubic	Subcubic	Our
Gdb-7.5.1	2405.62	10.83	**5.69**	150.54	102.6	**56.85**
Emacs-24.2	54781.9	128.03	**90.8**	1095.5	1908.16	**659.16**
Insight-6.8-1a	665.06	8.65	**3.31**	116.29	149.57	**56.31**
Gimp-2.8.4	662.27	8.19	**3.01**	56.84	51.04	**22.58**
Ghostscript-9.07	4209.84	28.67	**16.23**	256.4	271.51	**140.79**
Wine-1.5.25	8234.29	64.07	**21.19**	451.28	1448.21	**76.14**
Linux-3.8.2	31997	160.81	**73.45**	236.44	197.98	**66.09**

Results

- Final graph density

Program	#orig. edges	$\#V$-edges	$\#M$-edges	#Final edges CFL-reachability
Gdb-7.5.1	1,219,788	12,904,372	**356,075**	29,961,321
Emacs-24.2	1,290,200	49,011,799	**758,925**	112,772,537
Insight-6.8-1a	1,494,168	12,560,471	**432,657**	27,573,822
Gimp-2.8.4	1,675,546	16,809,343	**518,511**	33,151,263
Ghostscript-9.07	2,368,086	35,910,829	**705,121**	76,022,402
Wine-1.5.25	8,472,950	79,613,731	**2,769,135**	161,447,212
Linux-3.8.2	23,398,670	234,930,383	**6,272,658**	482,622,025

Conclusion

Summary

- Fast Dyck-CFL-reachability algorithm to scale alias analysis for Java
- Fast CFL-reachability algorithm to scale alias analysis for C
- Built practical tools to handle real-world programs
  - Soot for Java
  - Gcc for C

Thanks!

Appendix

Alias on SPG

- $x \leftarrow z$ and $w \rightarrow y$
  - x=z.f and y=w.f
  - x=z.f and w.f=y
  - z.f=x and y=w.f
  - z.f =x and w.f=y

Traditional CFL-Reachability algorithm

Algorithm 1: CFL-Reachability Algorithm.

Input : Edge-labeled directed graph $G = (V, E)$; normalized $CFG = (\Sigma, N, P, S)$;

Output: the set of summary edges;

add E to W ;
foreach *production* $A \to \varepsilon \in P$ **do**
    foreach *node* $v \in V$ **do**
        if $(v, A, v) \notin G$ **then**
            insert (v, A, v) to G and to W ;
while $W \neq \emptyset$ **do**
    $(i, B, j) \leftarrow$ SELECT-FROM(W) ;
    foreach *production* $A \to B \in P$ **do**
        if $(i, A, j) \notin G$ **then**
            insert (i, A, j) to G and to W ;
    foreach *production* $A \to BC \in P$ **do**
        foreach $k \in \text{OUT}(j, C)$ **do**
            if $(i, A, k) \notin G$ **then**
                insert (i, A, k) to G and to W ;
    foreach *production* $A \to CB \in P$ **do**
        foreach $k \in \text{IN}(i, C)$ **do**
            if $(k, A, j) \notin G$ **then**
                insert (k, A, j) to G and to W ;

Dyck-CFL-Reachability tree algorithm

Procedure 3: Add(v, e) to add a node v to STRATIFIED-SETS according to the directed edge $e = (u, v)$.

if $\mathcal{L}(u, v) \in A$ **then**
- let $a_i = \mathcal{L}(u, v)$
- Set[v] = curset
- Up[curset][a_i] = Find (u)
- curset ++

if $\mathcal{L}(u, v) \in \bar{A}$ **then**
- let $\bar{a}_i = \mathcal{L}(u, v)$
- **if** Up[Find(u)][a_i] *does not exist* **then**
  - Set[v] = curset
  - Up[Find(u)][a_i] = curset
  - curset ++
- **else**
  - Set[v] = Up[Find(u)][a_i]

Well-typed algorithm

Procedure 13: Add$(u, \mathcal{L}, v)$ to insert an edge $(u, \mathcal{L}, v)$.

foreach $x \in V[l(v)]$ **do**
    if $\mathcal{L}(u, v) == a$ **then**
        if $m_{x_0 u_1} == 1$ *and* $m_{x_0 v_1} \neq 1$ **then**
            `Mix` (x_0, v_1, u_1, v_1)
        if $m_{x_0 u_0} == 1$ *and* $m_{x_0 v_1} \neq 1$ **then**
            `Mix` (x_0, v_1, u_0, v_1)
        if $m_{x_1 u_1} == 1$ *and* $m_{x_1 v_1} \neq 1$ **then**
            `Mix` (x_1, v_1, u_1, v_1)
    if $\mathcal{L}(u, v) == \bar{a}$ **then**
        if $m_{x_0 u_0} == 1$ *and* $m_{x_0 v_0} \neq 1$ **then**
            `Mix` (x_0, v_0, u_0, v_0)
    if $\mathcal{L}(u, v) == \bar{d}$ **then**
        foreach $q \in \{0, 1\}$, $r \in \{0, 1\}$, $s \in \{0, 1\}$ **do**
            if $m_{v_q x_r} == 1$ *and* $u(x)$ *exists* **then**
                $w \leftarrow u(x)$
                if $m_{u_s w_s} \neq 1$ **then** $m_{u_s w_s} \leftarrow 1$ and insert w_s as a child of $T(u_s)$

Well-typed algorithm

Procedure 14: Mix(x_t, v_q, i_l, j_m) to merge trees.

```
insert j_m in T(x_t) as a child of i_l
m_{x_t j_m} ← 1
foreach child k_n of j_m ∈ T(v_q) do
    if m_{x_t k_n} ≠ 1 then
        Mix (x_t, v_q, j_m, k_n)
```

Algorithm 15: Pointer analysis algorithm for well-typed C.

Input : Edge-labeled bidirected PEG $G = (V, E)$;
Output: the reachability matrix M

```
run pre-process pass described in Section 5.4.1
Init ()
for k ← bottom to top do
    foreach (i, a, j), (i, ā, j) ∈ E_k do Add (i, L(i, j), j)
    foreach (i, d, j) ∈ E_k do Add (i, L(i, j), j)
```

General Dyck-CFL-Reachability

Procedure 7: Add(i, j) to insert an edge (i, j).

if $m_{ij} == 0$ **then** // there are no previous path from i to j
    for $x \leftarrow 1$ **to** n **do**
        if $m_{xi} \neq 0$ *and* $m_{xj} == 0$ **then**
            // the edge (i, j) gives rise to a new path from x to j
            Meld (x, j, i, j)

Procedure 8: Meld(x, j, u, v) to merge trees.

insert v in $T(x)$ as a child of u
$m_{xv} \leftarrow 1$
insert (x, S, v) to G and to W
foreach *child* w *of* $v \in T(j)$ **do**
    if $m_{xw} == 0$ **then**
        Meld (x, j, v, w) // update by means of

Algorithm 9: Dyck-CFL-Reachability Algorithm.

Input : Edge-labeled directed graph $G = (V, E)$;
Output: the set of summary edges;

initialize W to be empty
foreach $(i, a_i, j) \in E$ **do** insert (i, A_i, j) to G and to W
foreach $(i, \bar{a}_i, j) \in E$ **do** insert $(i, \bar{A}_i, j)$ to G and to W

while $W \neq \emptyset$ **do**
    $(i, B, j) \leftarrow$ SELECT-FROM(W)
    if $B == \bar{A}_i$ **then**
        foreach $k \in \text{IN}(i, a_i)$ **do**
            if $(k, S, j) \notin G$ **then**
                `Add` (k, j)
                insert (k, S, j) to G and to W

    if $B == A_i$ **then**
        foreach $k \in \text{OUT}(j, \bar{a}_i)$ **do**
            if $(i, S, k) \notin G$ **then**
                `Add` (k, j)
                insert (i, S, k) to G and to W

    if $B == S$ **then**
        foreach $k \in \text{IN}(i, a_i)$ **do**
            if $(k, A_i, j) \notin G$ **then**
                insert (k, A_i, j) to G and to W

        foreach $k \in \text{OUT}(j, \bar{a}_i)$ **do**
            if $(i, \bar{A}_i, k) \notin G$ **then**
                insert $(i, \bar{A}_i, k)$ to G and to W

General Dyck-CFL-Reachability

- Amortized time a

$$a = t + \sum_{v \in V} \varphi'(v) - \sum_{v \in V} \varphi(v).$$

- Total running time for m insertion

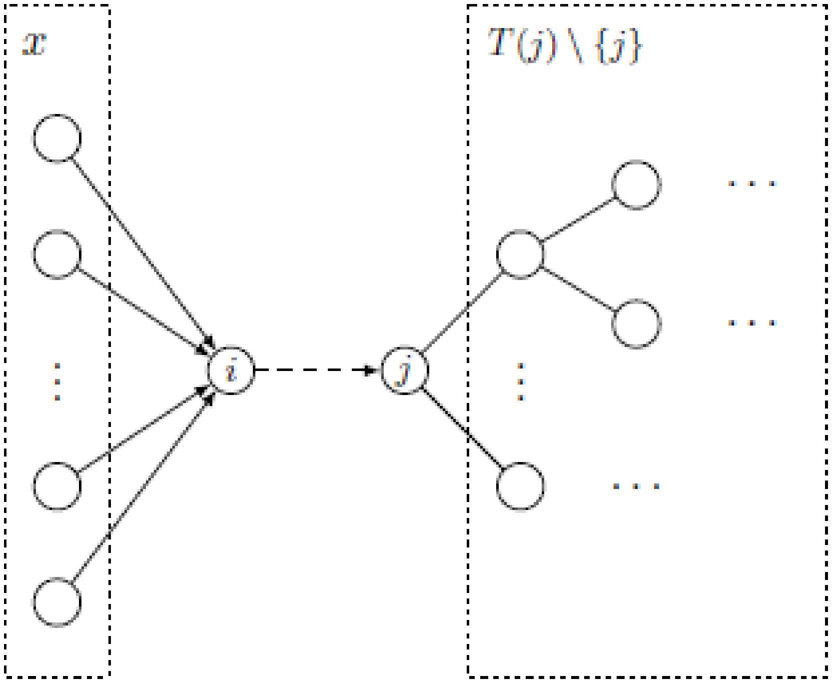
$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \sum_{v \in V} \varphi(v) - \sum_{v \in V} \varphi'(v).$$

- Define $vis(x) = \{(u, v) \in E |$u is a descendant of x$\}$
- Define potential function

$$\phi(x) = -(|vix(x)| + |T(x)|)$$

General Dyck-CFL-Reachability

- Meld() searches h_1 edges in $T(j)$ and adds h_2 edges to $T(x)$
- A net decrease of potential
  - Updating $T(x)$ does not sear
  - Each searched edge is beco
  - $T(x)$ is increased by h_2
- Inserting each edge to $T(x)$
- As a result for each x, each amortized time
  - We have n such x nodes



[TCS86] G.F. Italiano **Amortized efficiency of a path retrieval data structure**