論文紹介

Reading:

“Giga-Scale Exhaustive Points-to Analysis for Java in under a Minute”

(from OOPSLA’15)

発表者: 稲葉 一浩

複雑ネットワーク・地図グラフ セミナー 2017/1/19

About the paper

Giga-Scale Exhaustive Points-To Analysis for Java in Under a Minute

Jens Dietrich
Massey University, New Zealand
j.b.dietrich@massey.ac.nz

Nicholas Hollingum
The University of Sydney, Australia
nhol8058@uni.sydney.edu.au

Bernhard Scholz
Oracle Labs, Australia
Bernhard.Scholz@oracle.com

- Presented in OOPSLA:
  ACM SIGPLAN (= SIG on **P**rogramming **LAN**guages)
  International Conference on **O**bject-**O**riented
  Programming, **S**ystems, **L**anguages, and **A**pplications

Points-to Analysis?

- コンピュータープログラムの解析の一つ
  - 高速化や安全性の検査などに用いられる
- (典型的には)ラベル付き有向グラフでの制約付きreachability問題に帰着する近似解法で解かれる

Points-to Analysis?

- プログラム中のある箇所で確保したメモリ領域を指す可能性のある変数はどれか？(aやbやresultは⇗を指すか？

```
int[] newZeroVector() {
    return new int[3];
}
```

巨大なコンピュータプログラム

```
void crosProduct(
    int[] a, int[] b, int[] result
) {
    result[0] = a[1] * b[2] - b[2] * a[1];
    result[1] = a[2] * b[0] - b[0] * a[2];
    result[2] = a[0] * b[1] - b[1] * a[0];
}
```

Alias Analysis

- Points-to Analysisは “Alias Analysis” (二つの変数が同じメモリ領域を指す可能性があるか？) を含み、この解析の情報がよく使われる。

```
void crosProduct(
    int[] a, int[] b, int[] result
) {
    result[0] = a[1] * b[2] - b[2] * a[1];
    result[1] = a[2] * b[0] - b[0] * a[2];
    result[2] = a[0] * b[1] - b[1] * a[0];
}
```

a や b と result が同じ領域を指すか指さないかで大違い

About the paper

Giga-Scale Exhaustive Points-To Analysis for Java in Under a Minute

Jens Dietrich
Massey University, New Zealand
j.b.dietrich@massey.ac.nz

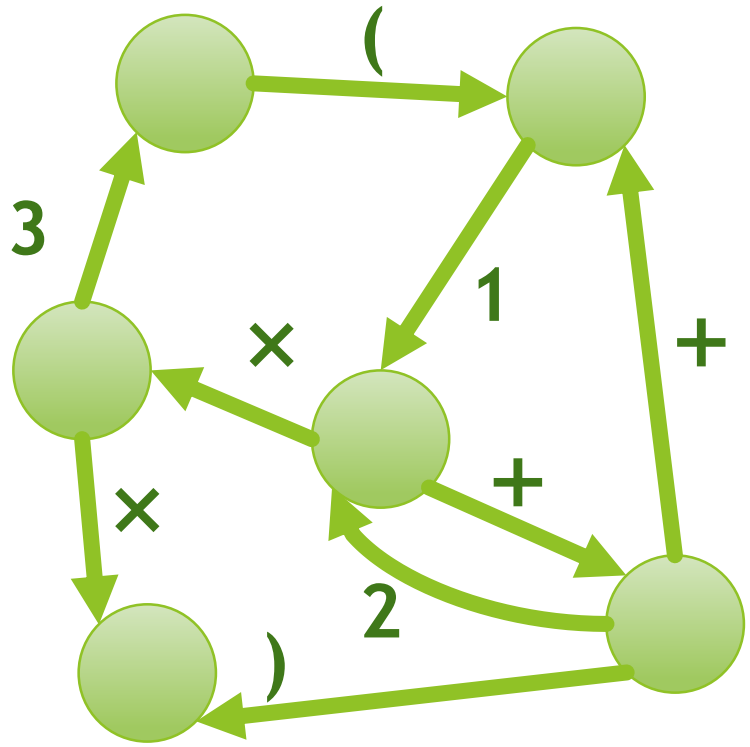
Nicholas Hollingum
The University of Sydney, Australia
nhol8058@uni.sydney.edu.au

Bernhard Scholz
Oracle Labs, Australia
Bernhard.Scholz@oracle.com

- Context-Free Points-to Analysis が
  In:|V|=160万　|E|=200万
  Out: 1.5G組
  のプログラムに対して 40秒 でできました

帰着先:
(All-Pairs) Context-Free Language Reachability

- 入力
  - 辺ラベル付き有向グラフ
  - 文脈自由文法
- 出力
  - 文法に従ったラベル列で到達可能な全点対

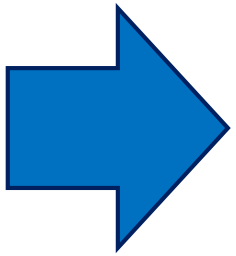


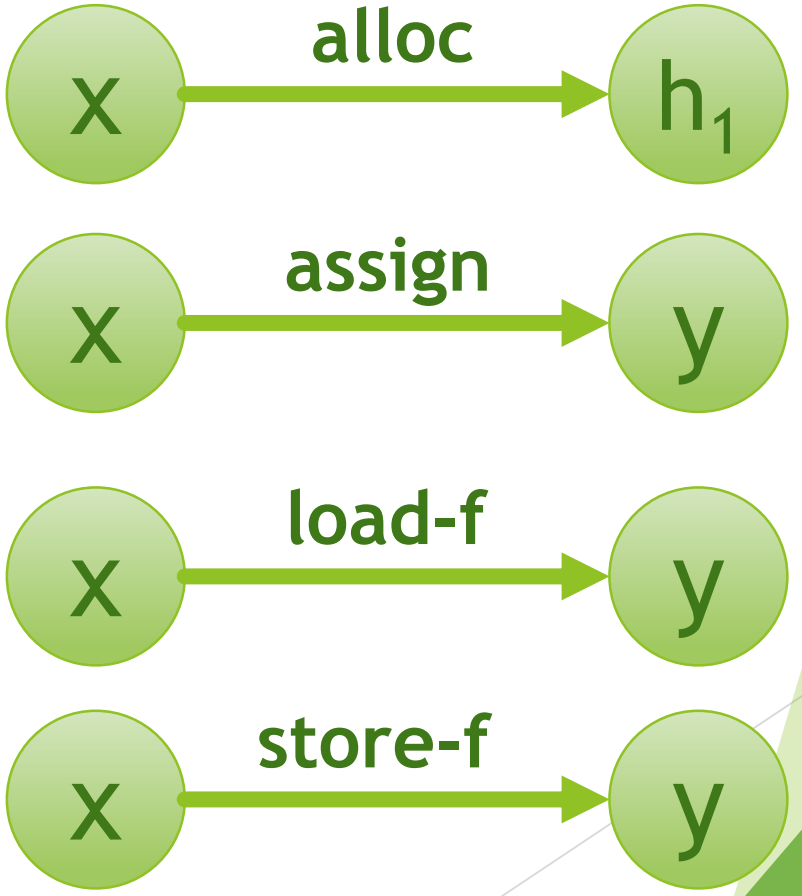
- Known Algorithm
  - O(|G| ・ |V|^3 / log|V|)

Expr	::= Term	|	Expr + Term
Term	::= Factor	|	Term × Factor
Factor	::= 1 | 2 | 3 | (Expr)		

Context-Free Points-to Analysis ⇒ Context-Free Graph Reachability

- x = new Object;
- x = y;
- x = y.f;
- x.f = y;

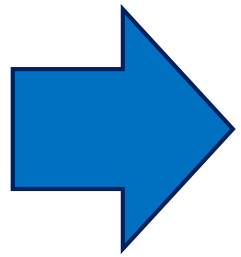


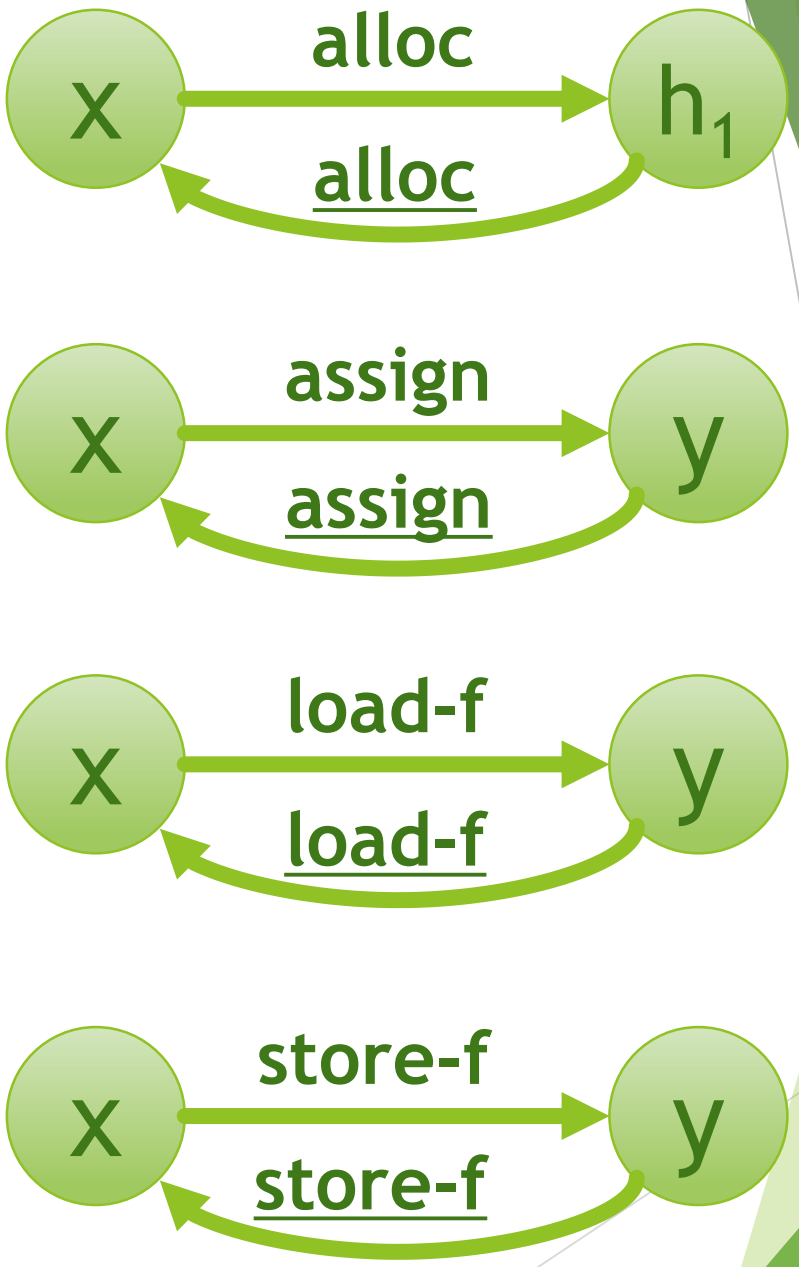


where h_1 is a unique ID for this program location

正確には

- x = new Object;
- x = y;
- x = y.f;
- x.f = y;



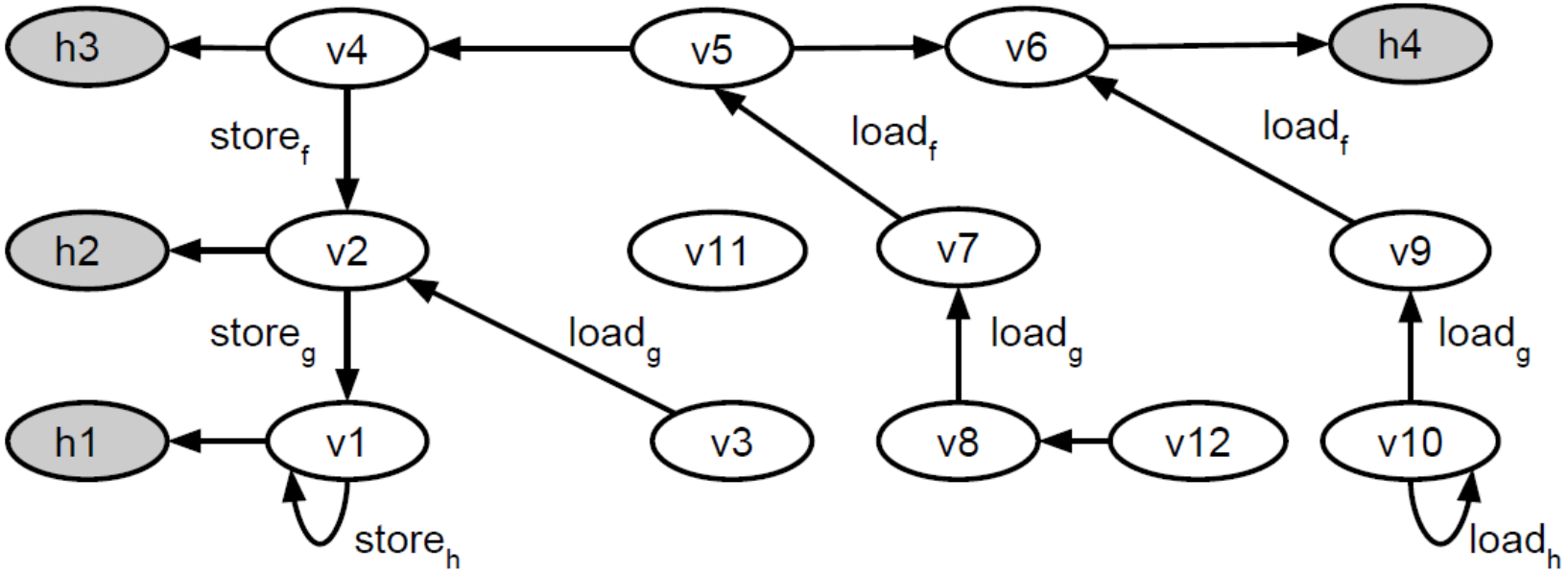


Example from the paper

```
v1 = new Obj(); // h1
v2 = new Obj(); // h2
v4 = new Obj(); // h3
v6 = new Obj(); // h4
v5 = v4;
v5 = v6;
```

```
v12 = v8;
v1.h = v1;
v2.g = v1;
v4.f = v2;
v7 = v5.f;
v9 = v6.f;
```

```
v3 = v2.g;
v8 = v7.g;
v10 = v9.g;
v10 = v10.h;
```



(To Java Experts)

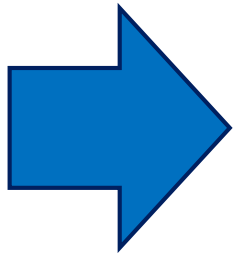
- プログラムの実行順序は無視
  （どちらも y may point-to h）

```
x = h;
y = x;
```

```
y = x;
x = h;
```

- 関数の呼び出しも代入で表現
  （異なる場所(context)での関数使用は混ざる）

```
Obj f(Obj x, Obj y) {
    return y;
}
z = f(h1, h2);
u = f(h3, h4);
```



```
x=h1; y=h2; z=y;

x=h3; y=h4; u=y;
```

Grammar

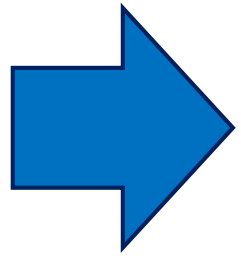
Alias ::= ***PointsTo PointsTo***

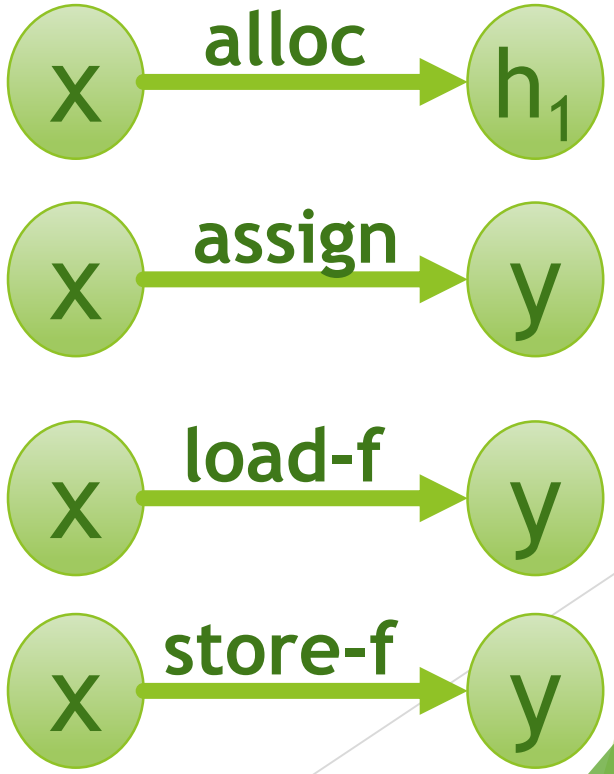
Bridge ::= load-f ***Alias*** store-f (for all f)

T ::= (assign | ***Bridge***)*

PointsTo ::= ***T*** alloc

- x = new Object;
- x = y;
- x = y.f;
- x.f = y;





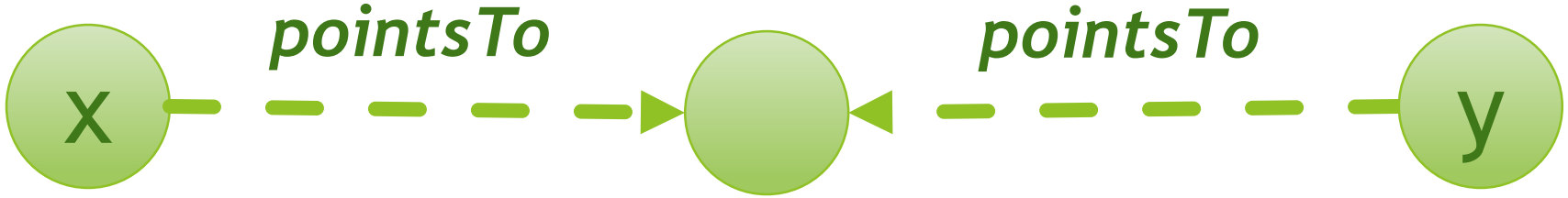
Grammar

Alias	::= *PointsTo* *PointsTo*	
Bridge	::= load-f *Alias* store-f	(for all f)
T	::= (assign | *Bridge*)*	
PointsTo	::= *T* alloc	

変数 x が h1 で確保された
メモリを指すかもしれない

Grammar

Alias	::= *PointsTo* *PointsTo*	
Bridge	::= load-f *Alias* store-f	(for all f)
T	::= (assign | *Bridge*)*	
PointsTo	::= *T* alloc	



変数 x と変数 y が同じ
メモリを指すかもしれない

Grammar

Alias ::= *PointsTo* *PointsTo*

Bridge ::= load-f *Alias* store-f (for all f)

T ::= (assign | *Bridge*)*

PointsTo ::= *T* alloc

y.f = w;

変数 x と変数 y が同じメモリを指すかも

z = x.f; ⇒ 要はz=w;と同じ効果

Grammar

Alias ::= *PointsTo* *PointsTo*
Bridge ::= load-f *Alias* store-f (for all f)
T ::= (assign | *Bridge*)*
PointsTo ::= *T* alloc

x=x1; x1=x2; …; xn=y; y=new Object;

変数 x がメモリ h を指すかも

h

アルゴリズム

```
1:  function POINTS-TO(bridge)
2:      W ← bridge
3:      T.add(assign)
4:      repeat
5:          N ← ∅
6:          for all (u, v) ∈ W do
7:              if IS-BRIDGE(u, v) then
8:                  N ← N ∪ {(u, v)}
9:              end if
10:         end for
11:         T.add(N)
12:         W ← W \ N
13:     until N = ∅
14:     pointsTo ← T ∘ alloc
15: end function
```

```
16: function IS-BRIDGE(u, v)
17:     for all (b_u, b_v) ∈ lsb(u,v) do
18:         if (T.query(b_u) ∩ T.query(b_v)) ◇ alloc ≠ ∅ then
19:             return true
20:         end if
21:     end for
22:     return false
23: end function
```

Alias ::= ***PointsTo*** PointsTo
Bridge ::= load-f ***Alias*** store-f
T ::= (assign | ***Bridge***)*
PointsTo ::= ***T*** alloc

Bridge の over approximaton (後述)

T は 推移閉包(ラベルなしの reachbility)を効率的に管理するデータ構造 [21]

現在の *T* から *Bridge*かどうか判定

Bridge の over approximation

- CFL Reachability の制限されたケース
  “Bidirected Dyck CFL Reachability” で近似
  - O(|E| log |E|)

- k-Dyck CFL とは、k 種類の括弧対 (2k種類の文字) 上の「括弧の対応がとれてる」を意味する文法
  - S ::= ε | SS | （S） | 【S】 | {S} | 「S」
- Bidirected: (a,【,b)∈E iff (b,】,a)∈E

- assign連結成分を潰し、load-f と store-f を括弧対と考える

実験結果

既存手法1

既存手法2

Datalog
商用ソルバ

提案手法

Bench	$\|V\|$	$\|E\|$	$\|E_{pointsTo}\|$
sunflow	15,464	15,957	16,354
lusearch	15,774	14,994	9,242
luindex	18,532	17,375	9,677
avrora	24,690	25,196	21,532
eclipse	41,383	40,200	21,830
h2	44,717	56,683	92,038
pmd	54,444	59,329	60,518
xalan	58,476	62,758	52,382
batik	60,175	63,089	45,968
fop	86,183	83,016	76,615
tomcat	111,327	110,884	82,424
jython	191,895	260,034	561,720
tradebeans	439,693	466,969	696,316
tradesoap	440,680	468,263	698,567
openjdk	1,621,634	1,964,146	1,570,820,597

[illegible]	WL_{time}	DP_{time}	LB_{time}	TC_{time}
sun	0.21	0.20	1.01	0.38
lus	0.19	0.18	1.01	0.42
lui	0.21	0.19	1.01	0.46
avr	0.51	0.32	1.02	0.60
ecl	0.48	0.40	1.04	0.74
h2	1.92	0.83	1.11	0.74
pmd	1.61	0.58	1.11	0.80
xal	1.72	0.71	1.09	0.84
bat	1.23	0.97	1.08	0.80
fop	2.70	2.05	1.12	1.08
tom	4.45	2.13	1.15	1.27
jyt	16.62	9.83	1.54	1.86
trb	122.90	52.36	1.95	4.34
trs	124.56	52.44	1.96	4.39
jdk	*	*	3,102.10	40.13

Bridgeの近似の精度

Bench	All	Field	Dyck	True
sun	$8.61{\cdot}10^5$	3,934	886	747
lus	$1.46{\cdot}10^6$	5,037	793	770
lui	$2.87{\cdot}10^6$	8,722	1,228	1,212
avr	$3.58{\cdot}10^6$	4,630	756	722
ecl	$4.39{\cdot}10^6$	6,444	1,430	1,096
h2	$8.31{\cdot}10^6$	12,174	3,492	3,281
pmd	$2.20{\cdot}10^7$	65,849	1,908	1,752
xal	$2.45{\cdot}10^7$	31,041	2,233	2,104
bat	$1.20{\cdot}10^7$	19,405	2,397	1,948
fop	$1.43{\cdot}10^7$	12,551	1,962	1,744
tom	$4.93{\cdot}10^7$	30,863	8,052	7,700
jyt	$6.12{\cdot}10^7$	55,776	8,646	8,082
trb	$6.15{\cdot}10^8$	149,642	30,863	29,155
trs	$6.17{\cdot}10^8$	149,724	30,894	29,173
jdk	$7.44{\cdot}10^9$	538,274	84,415	64,716