Análisis Automático de Programas

Análisis de points-to

Diego Garbervetsky
Departamento de Computación
FCEyN – UBA

En detalle

- Aliasing
- Points-to analysis
  - Como dataflow
    - Sensible a flujo
  - Clásicos: Steengard / Andersen
    - No sensibles a flujo/ contexto
    - Adaptación a Java (son para C)

Aliasing

- Dos expresiones son alias (sinónimos) cuando representan la misma locación mutable
  - E.g: El mismo objeto

```
1: A a = new A();
2: B b= new B();
3: A c = a;
4: c.f1 = b;
5: b.f2 = c;
```

- Pares:
  - <a,c>, <c.f1, b>, <b.f2, c>
  - <a.f1, b>, <b.f2, a>
  - <b.f2.f1, b>, …

Como se pueden generar aliases?

- Punteros (en C)
  - p = q ;
  - p = &q
- Manejo de arrays (índices)
  - swap(a,j,i) {
    a[j] = a[j] + a[i] ; a[i] = a[j] – a[i]; a[j] = a[j]-a[i];
    }
- Asignación de referencias (Java)
- Pasaje por referencia (C, C#)
  - En Java es por copia pero son referencias

Por qué analizar aliasing?

- Necesario para ser "sound"
  - Optimización
    - Eliminar/Mover/Reemplazar código
  - Inferencia de propiedades
    - Precisión en el cálculo
      - Por ejemplo call graphs
      - Efectos de una asignación
  - Útil también para entornos multithreading
    - Eliminación de sincronizaciones
    - Cálculo de dependencias

```
r1 = A.f;
B.f = r2;
r3 = A.f;
r5 = r1 + r3;
```

Si A y B no son alias

```
r1 = A.f;
B.f = r2;
r5 = 2*r1;
```

May alias vs Must alias

- May:
  - Pares que pueden cumplirse para algún camino
    - (a,b), (a,c), (c,d), (a,d)
- Must:
  - Pares que deben cumplen para todo camino
    - (c,d)

```
object a,b,c,d;
c = d
if(B)
  a = b;
else
  a = c;
```

Como inferir aliasing

- Dataflow: Out(N) = Gen(N) +(In(N) - Kill(N));
  - a = b
    - Gen = { (a,x) | (b,x) ∈ In(N) } ; Kill = { (a,?) }
  - a = new A();
    - Gen = { } ; Kill = { (a,?) }
  - a = b.f? a.f=b?
    - Ignoramos el campo?
- Necesitamos algún modelo de la memoria

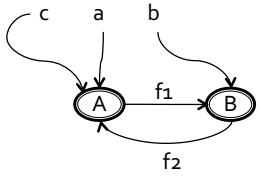
Points-to analysis

- Idea:
  - Determinar a donde apunta un puntero (o referencia)
- Modelado parcial de la memoria
- Es un problema fundamental en program analysis
  - Requerido por otros análisis, optimizadores, herramientas de program understanding, bug-finders, etc
  - Poder entender a qué objetos se puede referir una variable (o expresión)

Comparando con Alias analysis

- Points-to analysis:
  - Calcula el conjunto de locaciones en la memoria que un puntero puede referenciar
  - Análisis "May"
  - Computa un poinst-to graph
    - a --> A, b--> B, c--> A, A-f_1-> B, B-f_2->A
  - Alias analysis calcula pares
    - <a,c>, <c.f1, b>, <b.f2, c>, <a.f1, b>, <b.f2, a> <b.f2.f1, b>
  - Points-to graph puede utilizarse para calcular pares de aliasing

```
1: A a = new A();
2: B b= new B();
3: A c = a;
4: c.f1 = b;
5: b.f2 = c;
```



Tipos de análisis

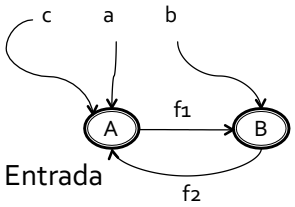
- **Sensitivos a flujo**
  - Dataflow
  - Computan points-to para cada punto
- **Insensitivos**
  - Computan un points-to para todo el programa
  - Básicamente dos clases:
    - Basados en inclusión (Andersen)
    - Basados en unificación (Steengard)
      - Menos preciso
- **Sensitivos a contexto**
  - Pueden distinguir entre diferentes llamadas a métodos

Modelo de un programa con punteros

- Comenzamos con Java simplificado
- Sin procedimientos
- Statements relevantes
  - new: x := new C()
  - copy: x := y
  - load: x := y.f
  - store: x.f := y

Points-to graphs (PTG)

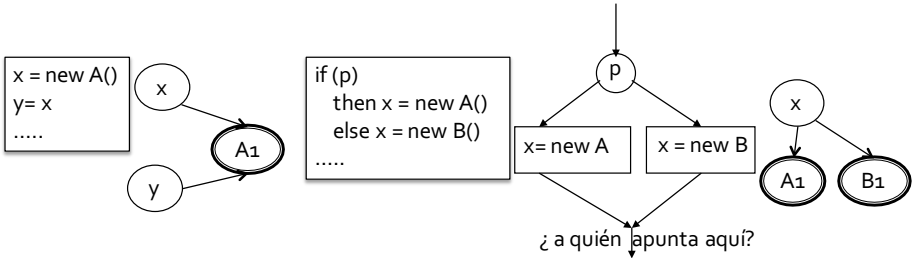
- Un **grafo dirigido** <N,E,L>
  - Nodos: representan objetos
  - Ejes (*o1*,f,*o2*): objeto o1 apunta a objeto o2 usando f
  - *L: Var → N* conjunto de variables locales. Entrada al grafo.



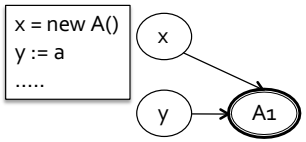
- pt(t): (Term)-> P(N) / (Term = Var.Field*)
  - A quiénes apunta una variable o expresión de camino
    - L(x) si *t es* var
    - U{ E(*n,f*) } si t = t1.f y n pertenece a pt(t1)
- pt(t), t: v.f* : A quién apunta un termino de la forma v, v.f, v.f.g, etc

Points-to graph (PTG)

- El grado de entrada de un nodo puede ser más que uno
  - Aliasing
- El grado de salida puede ser más que uno
  - Imprecisión
  - Si el points-to graph tiene ejes (*a*,*A*) y(*a*,*B*) una variable puede apuntar a *A* o B (MAY)
  - Se puede mejorar con sensibilidad a caminos



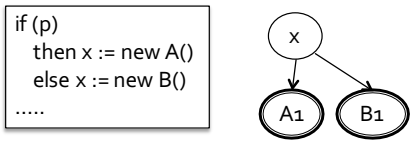
Que modela el points-to?



Representa a:

...	
x	A1
y	A1

h



Representa a:

...	
x	A1
..	

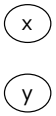
Y también a

...	
x	B1
..	

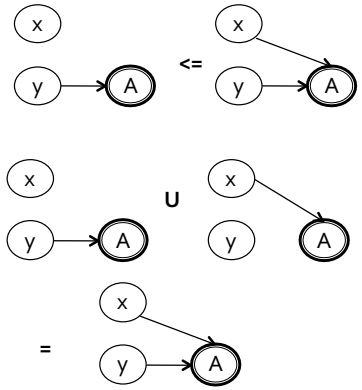
Esto **no** puede suceder en runtime

PTG como reticulado

- Dado un conjunto de variables
  - **Bottom**: grafo sin ejes, ni nodos



- Dados G1 and G2
  - **G1 <= G2** si G2 tiene todos los ejes que tiene G1 o algunos más. Similar con los nodos.
  - **G1 U G2**: El PTG más chico que contiene todos los ejes de G1 y G2



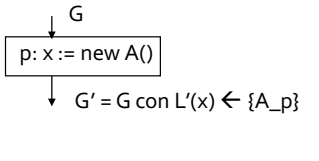
Points-to sensitivo a flujo

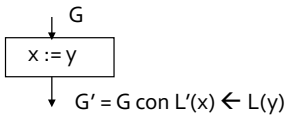
- Dataflow:
  - Forward, MAY
  - Combinador: G1 U G2

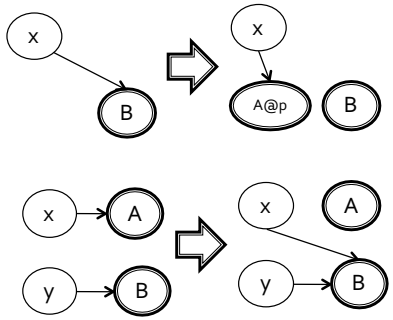
Notación:

S ← S1 (asignación)

S U← S1: (unión S ← S U S1)







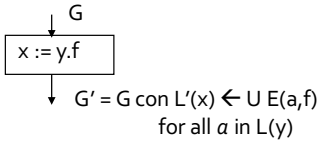
Ecuaciones dataflow

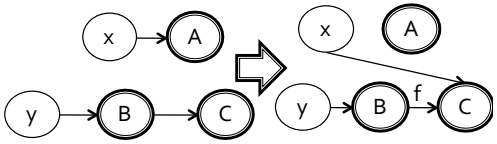
- Dataflow:
  - Forward, MAY
  - Combinador: G1 U G2

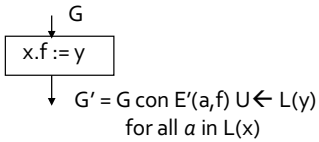
Notación:

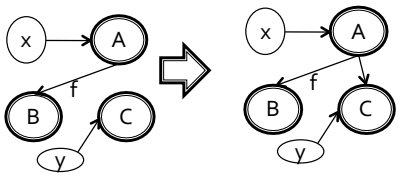
S ← S1 (asignación)

S U← S1: (unión S ← S U S1)

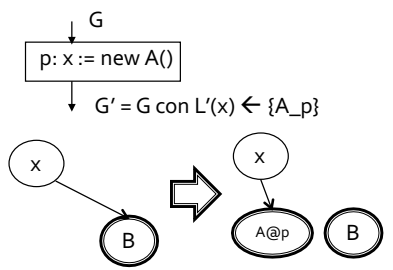


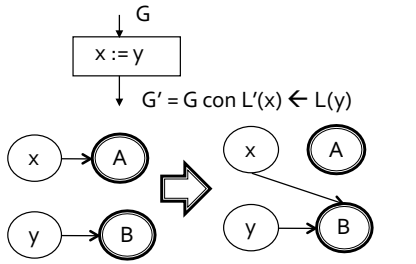


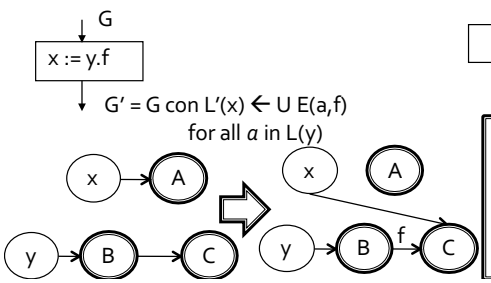


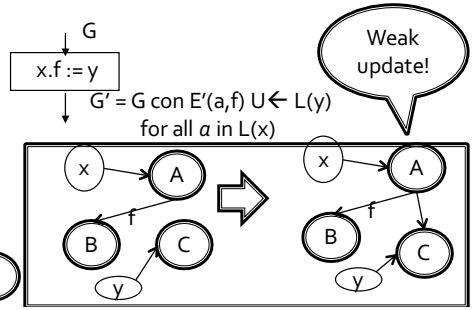


Ecuaciones dataflow



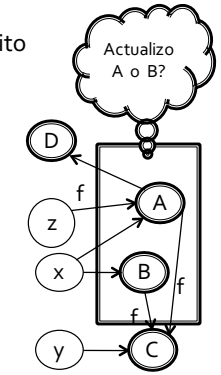






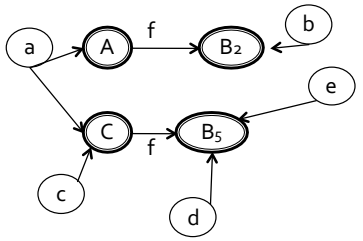
Strong vs. weak updates

- Strong update:
  - Cuando se sabe exactamente qué objeto está siendo escrito
  - Example: x := y
  - Se puede reemplazar la información de points-to de x
- Weak update:
  - No se sabe exactamente que objeto va a ser actualizado.
  - Ejemplo: x.f :=y
  - Durante el análisis puede no saberse a donde podría apuntar x
- Pequeña mejora: si el grado de salida es **1** se puede aplicar un strong update



Ejemplo

```
1: a = new A();
2: b = new B();
3: a.f = b;
4: c = new C() ;
5: d = new B();
6:c.f = d;
7: a = c;
8: e = a.f
```



L(a) ← {A}
L(b) ← {B2}
E(a,f) ={ B2}
L(c) ← {C}
L(d) ← {B5}
E(c,f) = {B5}
L(a) ={C}
L(e) ← E(a,f) ={B5}

p: x := new A()	G' = G con L'(x) ← {A_p}
x := y	G' = G con L'(x) ← L(y)
x := y.f	G' = G con L'(x) ← U E(a,f) for all α in L(y)
x.f := y	G' = G con E'(a,f) U← L(y) for all α in L(x)

Cómo abstraer desreferencias

- Ser field-sentitive
  - Reconocer x.left

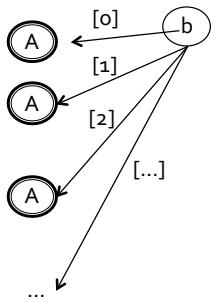
```
Class Cell {
      int value;
      Cell left,
           right;
}
Cell x,y;
```

- Ser field-independent
  - Tratar a x.left y x.right como x.*
- Ser field-based
  - Tratar a x.left e y.left como *.left
- Ser class based
  - Tratar a x.left e y.left como Cell.left

Abstracción del heap

- Un nodo por cada objeto creado

```
void m(int k) {
   int i=0;
   while(i<k) {
      A a = new A();
      b[i] = a;
      …
   }
}
```



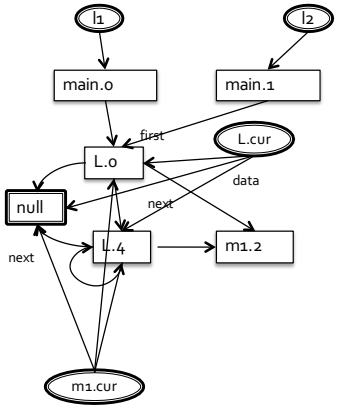
- No conocido en tiempo de compilación…
- Y potencialmente infinito…

Abstracción del heap

- Un nodo por cada objeto creado
  - Pueden ser infinitos
- Un nodo por cada "allocation site" (new statement)
  - Se puede ver como una variable global por new
  - ¿Qué pasa con las allocations sites en loops?
- Alternativas:
  - Menos precisa: Un solo nodo para todo el heap
  - Más precisa: diferentes nodos para alocaciones creadas en diferentes contextos

Un nodo por allocation site

```
class L {
 N first;
 A data
 L(int k) {
0: N n =new N();
1: first = n;
2: N cur = first;
3: for(int i=1; i<k; i++) {
4:  N n2 = new N();
5:  cur.next = n2;
6:  cur = cur.next;
   }
}
 void m0(L l) {
0: m1();
1: l.m1();
 }
 void m1(L l2) {
0: N cur = first;
1: while(cur!=null) {
2:   A a = new A();
3:   cur.data = a;
4:   cur = cur.next;
 }
 void main() {
0:  L l1 = new L(4);
1:  L l2 = new L(1);
2:  l1.m0(l2);
 }
}
```

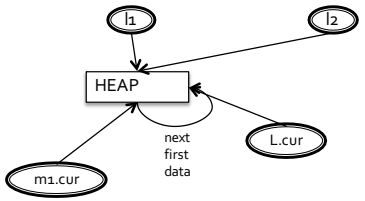


Abstracción del heap

- Un nodo por cada objeto creado
  - Pueden ser infinitos
- Un nodo por cada "allocation site" (new statement)
  - Se puede ver como una variable global por new
  - ¿Qué pasa con las allocations sites en loops?
- Alternativas:
  - Menos precisa: Un solo nodo para todo el heap
  - Más precisa: diferentes nodos para alocaciones creadas en diferentes contextos
- Estos modelos son imprecisos para estructuras recursivas
  - Terreno del shape analysis

Un único nodo para todo el heap

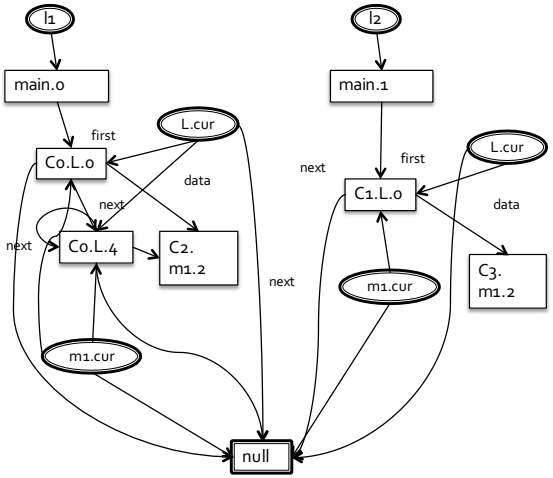
```
class L {
 N first;
 A data
 L(int k) {
0: N n =new N();
1: first = n;
2: N cur = first;
3: for(int i=1; i<k; i++) {
4:  N n2 = new N();
5:  cur.next = n2;
6:  cur = cur.next;
   }
}
 void m0(L l) {
0: m1();
1: l.m1();
 }
 void m1(L l2) {
0: N cur = first;
1: while(cur!=null) {
2:   A a = new A();
3:   cur.data = a;
4:   cur = cur.next;
 }
 void main() {
0:  L l1 = new L(4);
1:  L l2 = new L(1);
2: l1.m0(l2);
 }
}
```



Un nodo por allocation site

```
class L {
 N first;
 A data
 L(int k) {
0: N n =new N();
1: first = n;
2: N cur = first;
3: for(int i=1; i<k; i++) {
4:  N n2 = new N();
5:  cur.next = n2;
6:  cur = cur.next;
   }
}
 void m0(L l) {
0: m1();
1: l.m1();
 }
 void m1() {
0: N cur = first;
1: while(cur!=null) {
2:   A a = new A();
3:   cur.data = a;
4:   cur = cur.next;
 }
 void main() {
0:  L l1 = new L(4);
1:  L l2 = new L(1);
2: l1.m0(l2);
 }
}
```

Diferentes allocs por contexto



Co = main.1; C1 = main.2; C2 = main.2.mo.0 ; C3 = main.2.mo.1

Points-to analysis interprocedural

- Misma discusión que en Dataflow
- En LOO se suele hacer sensitivo a contexto y no a flujo
- Opciones:
  - Call strings (k-limiting)
  - Cartesian Product (CPA)
    - El contexto está dado por el valor de los argumentos
  - Object sensitivity
    - El contexto está dado por el valor del receiver (this)
  - Summaries

Análisis no sensitivos a flujo

- Sensitividad a flujo => un PTG para cada punto del programa
  - Caro: memoria y tiempo
- Flow-insensitive analysis
  - Calcular una única relación de points-to para el programa
  - Ignorar el flujo de control
  - Considerar todas las asignaciones como no destructivas
    - Cambiar strong updates por weak updates

Andersen

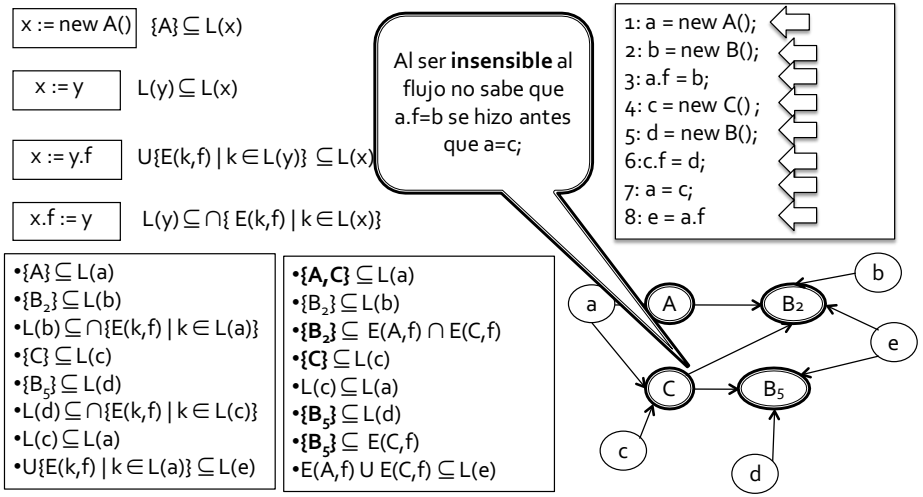
- Basado en restricciones basadas en inclusiones de conjuntos
  - p = q, Pts-to(q)⊆ Pts-to(p)
  - Complejidad cúbica (clausura de restricciones)
- **Caracterización**
  - Whole program
  - Flow-insensitive
  - Context-insensitive
  - May analysis
  - Alias representation: points-to

Andersen

- Intuición: quitar sensibilidad a flujo a un algoritmo dataflow
- Al no conocer el orden de las instrucciones, se puede asegurar menos (weak updates)
  - x = new A() — sólo sabemos que A∈*L(x)*
  - x = y — sólo sabemos que *L(y) ⊆L(x)*
- Algoritmo:
  - Recorrer el código y recolectar las restricciones
  - Para calcular los points-to set: resolver las restricciones aplicando un punto fijo

Algoritmo de Andersen

- Usando inclusiones de restriciones



Algoritmo de Steensgard's

- Basado en la unificación de restricciones
  - p = q →Pt(p) = Pt(q)
  - Calcula un PTG con un grado de salida<=1
  - Es menos preciso que Andersen
  - Es casi lineal en el tamaño del programa
- Caracterización
  - Whole program
  - Flow-insensitive
  - Context-insensitive
  - May analysis
  - Alias representation: points-to

Algoritmo de Steengard

- Usando igualdad de restriciones

x := new A()	{A} = L(x)
x := y	L(y) = L(x)
x := y.f	U{E(k,f) | k ∈ L(y)} = L(x)
x.f := y	L(y) = ∩{ E(k,f) | k ∈ L(x)}

```
1: a = new A();
2: b = new B();
3: a.f = b;
4: c = new C() ;
5: d = new B();
6:c.f = d;
7: a = c;
8: e = a.f
```

- {A} = L(a)
- {B2} = L(b)
- L(b) = ∩{E(k,f) | k ∈ L(a)}
- {C} =L(c)
- {B5} = L(d)
- L(d) = {E(k,f) | k ∈ L(c)}
- L(c) = L(a)
- U{E(k,f) | k ∈ L(a)} = L(e)

- {A,C} = [AC]
- **[AC]**= L(a)=L(c)
- {B2} = L(b)
- {B2} = E([AC],f)
- {B5} = L(d)
- {B5} = E([AC],f)
- E([AC],f) = L(e)

- {A,C} = [AC]
- **[AC]**= L(a)=L(c)
- {B2,B5} = [B2B5]
- [B2B5] = L(b)=L(d)
- **[B2B5]**= E([AC],f)
- **[B2B5]** = L(e)

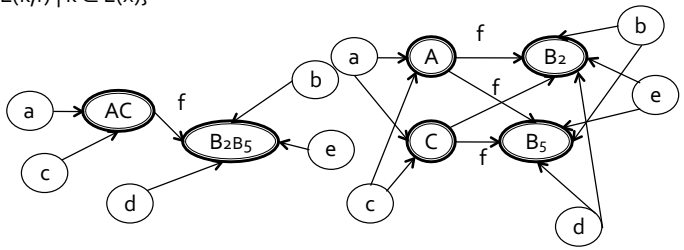
Algoritmo de Steengard

- Usando igualdad de restriciones

x := new A()	{A} = L(x)
x := y	L(y) = L(x)
x := y.f	U{E(k,f) | k ∈ L(y)} = L(x)
x.f := y	L(y) = ∩{ E(k,f) | k ∈ L(x)}

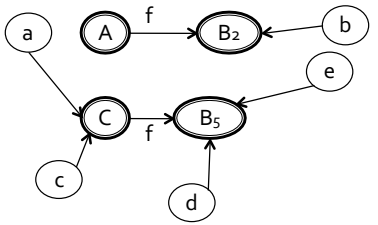
```
1: a = new A();
2: b = new B();
3: a.f = b;
4: c = new C() ;
5: d = new B();
4:c.f = d;
5: a = c;
6: e = a.f
```

- {A,C} = [AC]
- {B2,B5} = [B2B5]
- [AC]= L(a)=L(c)
- [B2B5] = L(b)=L(d)
- [B2B5]= E([AC],f)
- [B2B5] = L(e)

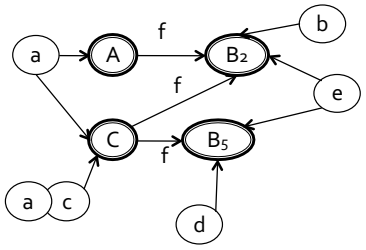


Comparando algoritmos

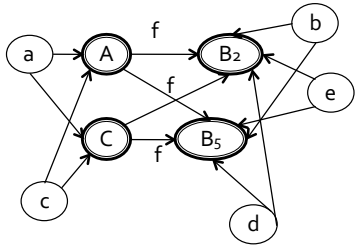
```
1: a = new A();
2: b = new B();
3: a.f = b;
4: c = new C() ;
5: d = new B();
4:c.f = d;
5: a = c;
6: e = a.f
```



Dataflow



Andersen



Steengard

Sensitividad a contexto

- Call strings
  - Modelan el stack de **control**
  - K-limiting o k-CFA
- CPA (Cartesian product Algorithm)
  - El contexto esta dado por la abstracción de los parámetros (**datos**)
- Object sensitivity (subclase de CPA)
  - El contexto esta dado por el valor abstracto de "this"
- Que es más preciso?
  - 1-CFA vs Object sensitivy?
    - Incomparable! (ver http://www.cs.rutgers.edu/~ryder/CCo3InvitedNew.pdf)
  - CPA vs ∞-CFA?
    - CPA es más preciso! Se probo hace relativamente poco
      - (CPA beats 00-CFA en FTjFP 2009)

Muchísimo para leer

- Hay una gran diversidad de points-to analysis
- Su utilidad varia según el lenguaje y contexto de uso
- Ejemplos
  - En Java:
    - Object sensitive mejora la capacidad de resolver virtual calls y comprobar cast pero no impacta tanto en el tamaño del call graph.
  - En C
    - Puede ser importante la sensibilidad a flujo
    - En algunos casos es crucial poder inferir direcciones (aritmética abstracta de punteros)

Bibliografía:

- Barbara Ryder: Reference Analysis slides
- Radu Rugina
- Alex Aiken
- Whaley, Rinard: "Compositional Pointer and Escape Analysis for Java Programs" OOPSLA 99
- Ondřej Lhoták and Laurie Hendren, "Context-sensitive points-to analysis: is it worth it?" CC 2006