

Análisis Automático de Programas

Análisis de points-to

Diego Garbervetsky, Guido de Caso
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 alias?

- 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

```
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;
```

- Útil también para entornos multithreading
 - Eliminación de sincronizaciones
 - Cálculo de dependencias

May alias vs Must alias

- May:

- Pares que pueden cumplirse para algún camino
 - (a,b), (a,c), (c,d), (a,d)

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

- Must:

- Pares que deben cumplir para todo camino
 - (c,d)

Como inferir aliasing

- Dataflow: $Out(N) = Gen(N) + (In(N) - Kill(N));$

- $a = b$
 - $Gen = \{ (a,x) \mid (b,x) \in In(N) \}; Kill = \{ (a,?) \}$
- $a = new A();$
 - $Gen = \{ \}; Kill = \{ (a,?) \}$
- $a = b.f?$ $a.f=b?$
 - 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 point-to graph

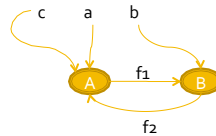
- $a \rightarrow A, b \rightarrow B, c \rightarrow A,$
 $A.f_1 \rightarrow B, B.f_2 \rightarrow A$

- Alias analysis calcula pares

- $\langle a, c \rangle, \langle c.f_1, b \rangle, \langle b.f_2, c \rangle,$
 $\langle a.f_1, b \rangle, \langle b.f_2, a \rangle$
 $\langle b.f_2.f_1, b \rangle$

- 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

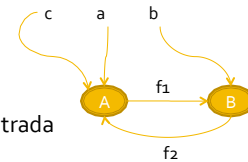
- Statements relevantes

- new: $x := \text{new } C()$
- copy: $x := y$
- load: $x := y.f$
- store: $x.f := y$

Points-to graphs (PTG)

- Un grafo dirigido $\langle N, E, L \rangle$

- Nodos: representan objetos
- Ejes (o_1, f, o_2) : objeto o_1 apunta a objeto o_2 usando f
- $L: \text{Var} \rightarrow N$ conjunto de variables locales. Entrada al grafo.



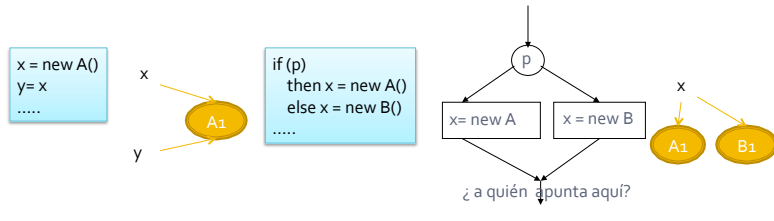
- $\text{pt}(x): (\text{Var} \cup N) \rightarrow P(N)$

- A quiénes apunta una variable o u objeto
 - $L(x)$ si x var
 - $\cup \{ E(x, f) \}$ si x nodo
- Puede cambiar en cada punto del programa!

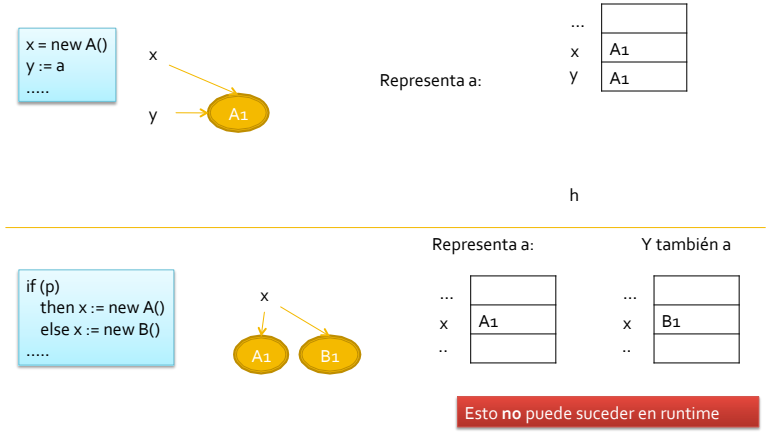
- $\text{pt}(x, f)$: A quién apunta una variable o u objeto avanzando por f

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

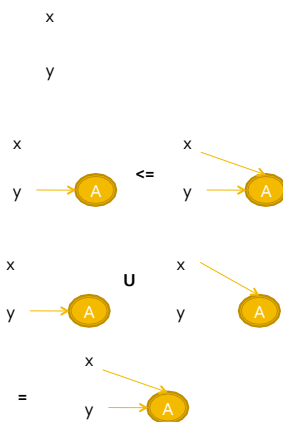


Que modela el points-to?

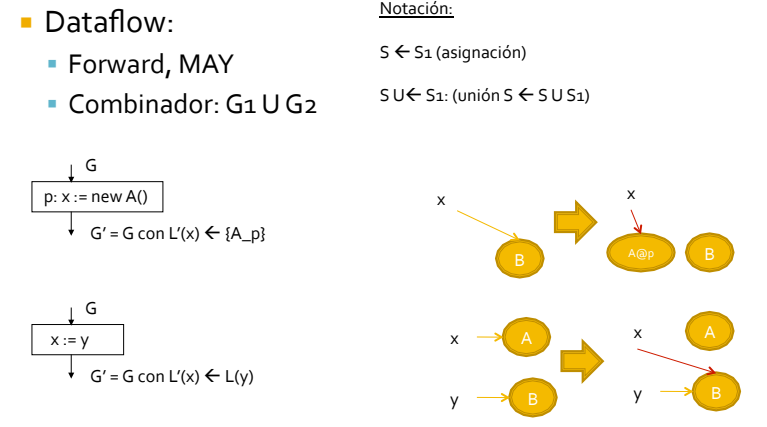


PTG como reticulado

- Dado un conjunto de variables
 - Bottom:** grafo sin ejes, ni nodos
- Dados G_1 and G_2
 - $G_1 \leq G_2$ si G_2 tiene todos los ejes que tiene G_1 o algunos más. Similar con los nodos.
 - $G_1 \cup G_2$: El PTG más chico que contiene todos los ejes de G_1 y G_2



Points-to sensitivo a flujo



Cómo abstraer desreferencias

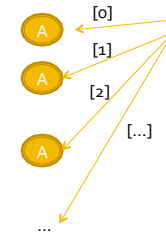
```
Class Cell {
    int value;
    Cell left,
        right;
}
Cell x,y;
```

- Ser field-sensitive
 - Reconocer x.left
- Ser field-independent
 - Tratar a x.left y x.right como x.*
- Ser field-based
 - Tratar a x.left e y.left como *.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;
        ...
    }
}
```



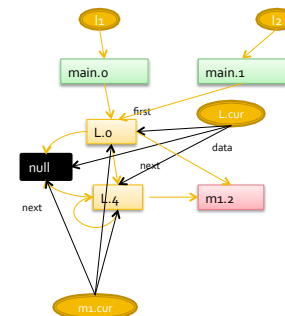
- No conocido en tiempo de compilación...
- Y potencialmente infinito...

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 allocaciones creadas en diferentes contextos

Un nodo por allocation site

```
class L {
    N first;
    A data;
}
L(int k) {
    0: N n = new NO();
    1: first = n;
    2: N cur = first;
    3: for(int i=1; i<k; i++) {
    4: N n2 = new NO();
    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);
}
```



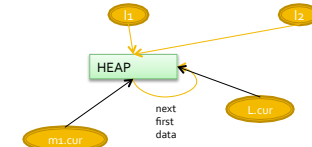
Abstracción del heap

- 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 alocaiones 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 NO();
    1: first = n;
    2: N cur = first;
    3: for(int i=1; i<k; i++) {
    4:   N n2 = new NO();
    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);
  }
}
    
```

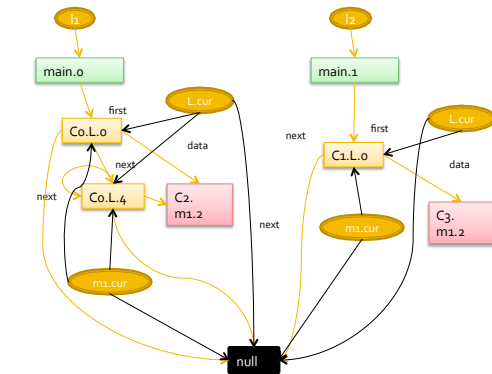


Un nodo por allocation site

```

class L {
  N first;
  A data
  L(int k) {
    0: N n = new NO();
    1: first = n;
    2: N cur = first;
    3: for(int i=1; i<k; i++) {
    4:   N n2 = new NO();
    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

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
 - $p = q, \text{Pts-to}(q) \subseteq \text{Pts-to}(p)$
 - 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)
 - $x = \text{new } A()$ — sólo sabemos que $A \in L(x)$
 - $x = y$ — sólo sabemos que $L(y) \subseteq L(x)$
- 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 restricciones

$x := \text{new } A()$ $\{A\} \subseteq L(x)$

$x := y$ $L(y) \subseteq L(x)$

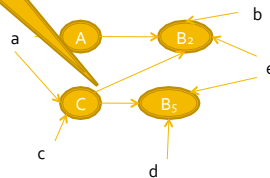
$x := y.f$ $\cup \{E(k,f) \mid k \in L(y)\} \subseteq L(x)$

$x.f := y$ $L(y) \subseteq \cup \{E(k,f) \mid k \in L(x)\}$

Al ser insensible al flujo no sabe que $a.f=b$ se hizo antes que $a=c$;

```

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
    
```



- $\{A\} \subseteq L(a)$
- $\{B_1\} \subseteq L(b)$
- $L(b) \subseteq \cup \{E(k,f) \mid k \in L(a)\}$
- $\{C\} \subseteq L(c)$
- $\{B_2\} \subseteq L(d)$
- $L(d) \subseteq \cup \{E(k,f) \mid k \in L(c)\}$
- $L(c) \subseteq L(a)$
- $\cup \{E(k,f) \mid k \in L(a)\} \subseteq L(e)$

- $\{A, C\} \subseteq L(a)$
- $\{B_1\} \subseteq L(b)$
- $\{B_2\} \subseteq E(A,f) \cap E(C,f)$
- $\{C\} \subseteq L(c)$
- $L(c) \subseteq L(a)$
- $\{B_2\} \subseteq L(d)$
- $\{B_2\} \subseteq E(C,f)$
- $E(A,f) \cup E(C,f) \subseteq L(e)$

Algoritmo de Steensgard's

- Basado en la unificación de restricciones
 - $p = q \rightarrow \text{Pt}(p) = \text{Pt}(q)$
 - 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 restricciones

$x := \text{new } A()$ $\{A\} = L(x)$

$x := y$ $L(y) = L(x)$

$x := y.f$ $\cup\{E(k,f) \mid k \in L(y)\} = L(x)$

$x.f := y$ $L(y) = \cap\{E(k,f) \mid k \in L(x)\}$

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

• $\{A\} = L(a)$
• $\{B_2\} = L(b)$
• $L(b) = \cap\{E(k,f) \mid k \in L(a)\}$
• $\{C\} = L(c)$
• $\{B_5\} = L(d)$
• $L(d) = \cap\{E(k,f) \mid k \in L(c)\}$
• $L(c) = L(a)$
• $\cup\{E(k,f) \mid k \in L(a)\} = L(e)$

• $\{A,C\} = [AC]$
• $[AC] = L(a)=L(c)$
• $\{B_2\} = L(b)$
• $\{B_2\} = E([AC],f)$
• $\{B_5\} = L(d)$
• $\{B_5\} = E([AC],f)$
• $E([AC],f) = L(e)$

• $\{A,C\} = [AC]$
• $[AC] = L(a)=L(c)$
• $\{B_2, B_5\} = [B_2B_5]$
• $[B_2B_5] = L(b)=L(d)$
• $[B_2B_5] = E([AC],f)$
• $[B_2B_5] = L(e)$

Algoritmo de Steengard

- Usando igualdad de restricciones

$x := \text{new } A()$ $\{A\} = L(x)$

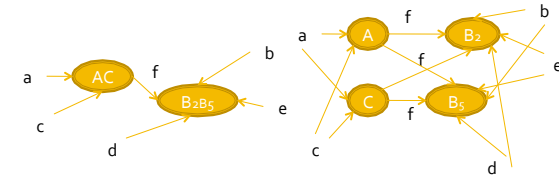
$x := y$ $L(y) = L(x)$

$x := y.f$ $\cup\{E(k,f) \mid k \in L(y)\} = L(x)$

$x.f := y$ $L(y) = \cap\{E(k,f) \mid k \in L(x)\}$

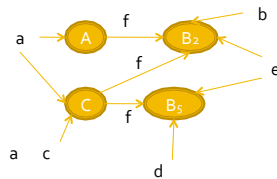
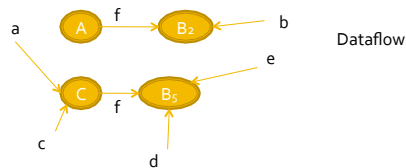
1: a = new A();
2: b = new B();
3: a.f = b;
4: c = new C();
5: d = new B();
6: c.f = d;
7: a = c;
8: e = a.f

• $\{A,C\} = [AC]$
• $\{B_2, B_5\} = [B_2B_5]$
• $[AC] = L(a)=L(c)$
• $[B_2B_5] = L(b)=L(d)$
• $[B_2B_5] = E([AC],f)$
• $[B_2B_5] = L(e)$

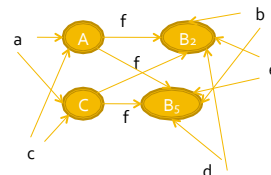


Comparando algoritmos

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



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 sensitivity?
 - Incomparable! (ver <http://www.cs.rutgers.edu/~ryder/CCoInvitedNew.pdf>)
 - CPA vs ∞-CFA?
 - CPA es más preciso! Se probó hace muy poco C
 - (PA beats ∞-CFA en FTJ'FP 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