Análisis Automático de Programas

Análisis dataflow interprocedural

Diego Garbervetsky
Departamento de Computación
FCEyN – UBA

Como analizar varios métodos?

```
int divByX(int x) {
    [result := 10/x]1;
}

void caller1() {
   [x := 5]1;
   [y := divByX(x)]2;
   [y := divByX(5)]3;
   [y := divByX(1)]4;
}
```

- ¿Cómo sabemos que divByX no falla?

```
float area(Cuadrado c) {
    [result := c.l*c.l]1;
}

void caller1(Cuadrado c2) {
   [Cuadrado c = newCuadrado()]1;
   [float a1= area(c)]2;
   [return area(c2)+a1]3;
}
```

- ¿Cómo sabemos que área no falla?

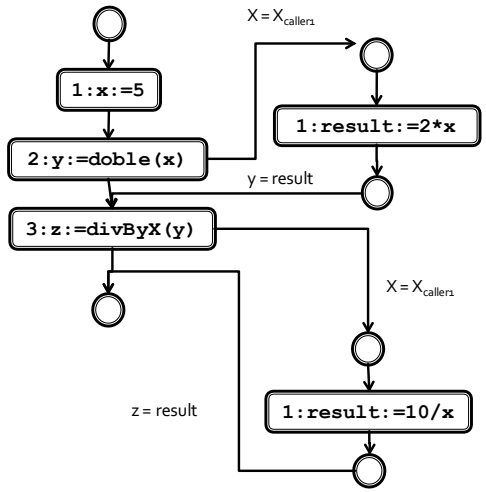
Análisis interprocedural

- Analizar un programa compuesto por varios métodos
- Estrategias:
  - **Construir CFG interprocedural**
  - Asumir/Verificar
  - Sentividad a contexto
    - Inlining
    - Call string
    - Calcular "resumenes"

CFG interprocedural

- Extender CFG para varios procedimientos
  - Un eje del **llamador** al *entry* del **llamado**
  - Un eje del retorno al punto siguiente al call

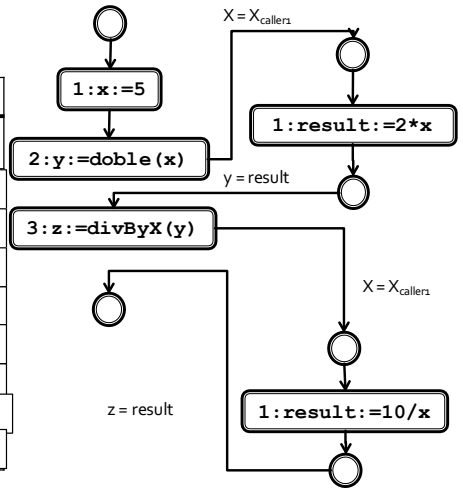
```
int doble(int x) {
    [result := 2*x]1;
}
int divByX(int x) {
    [result := 10/x]1;
}
void caller1() {
   [x := 5]1;
   [y := doble(x)]2;
   [z := divByX(y)]3;
}
```



CFG interprocedural

- Análisis de caller1

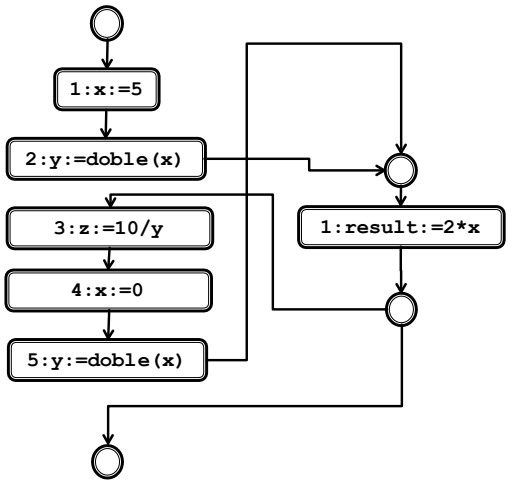
	caller1			doble		divByX	
Pos	x	y	z	x	res	x	res



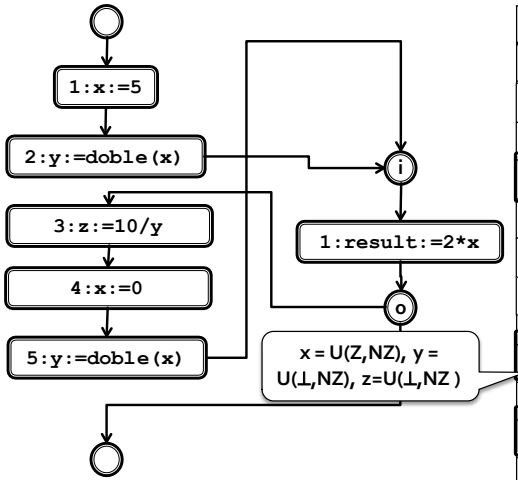
Otro ejemplo

```
int doble(int x) {
    [result := 2*x]1;
}
void caller1() {
   [x := 5]1;
   [y := doble(x)]2;
   [z := 10/y]3
   [x := 0]4;
   [y := doble(x)]5;
}
```

¿Le ven algún problema?



Otro ejemplo

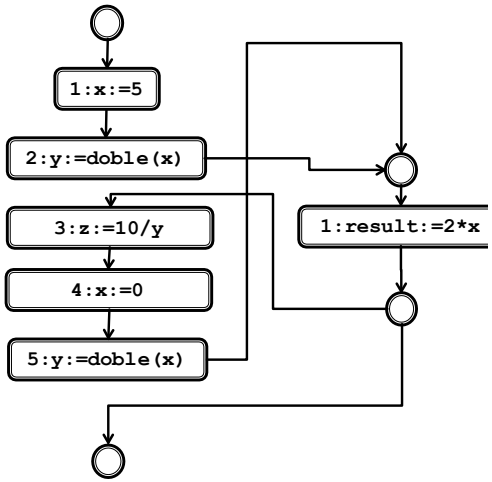


	Caller			doble	
Pos	x	y	z	x	Res

- Warning: div por cero!
- Falso positivo!

Cual fue el problema?

- El CFG interprocedural pierde precisión
- No distingue diferentes contextos de llamada
- En particular puede recorrer caminos que en realidad no son factibles



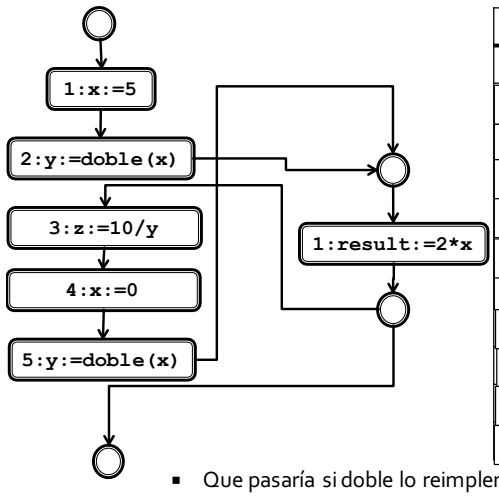
Sensibilidad al contexto

- Hay que distinguir entre los diferentes contextos de llamada a una función
- Varias técnicas
  - **Cadenas de llamadas**
  - Inlining/Cloning
  - Asumir / Verificar
  - Resúmenes por métodos

Cadenas de llamadas

- En runtime diferentes llamadas a un metodo (ej: doble) estan distinguidas por su call stack
  - En particular lo podríamos hacer por su parte de control (ej: *m1.4+m2.1+….*)
- Problema
  - El stack puede no tener cota
    - Recursión
- Idea:
  - Usar las las ultimas ***k*** llamadas para distinguir contextos
  - Se lo llama k-limit

Ejemplo revisitado – 1 limit



	Caller				doble	
Pos	x	y	z	CC	x	Res

- Que pasaría si doble lo reimplementamos haciendo una llamada a otro método?
  - `int doble(int x) { return g(x); }`
  - `int g(int x) { return (2*x); }`

Recursión y cadenas de llamadas

```
int doble(int x) {
    [result := 2*x]1;
}
int g(int v)
{
   if(v>2)
     [return g(v-1)]1T
   else
    [return doble(v)]1F
}
void m(int x) {
   [y := g(x)]1;
   [z := 10/y]2;
}
```

- Es posible hallar un k que permita analizar este problema con precisión?
  - m_1.g_1F.doble
  - m_1. g_1T.g_1F.doble
  - m_1. g_1T. g_1T.g_1F.doble
  - …
- Cadenas por llamadas requiere aproximación en casos recursivos

Cadenas de llamadas

Resumiendo:

- Se introduce al estado abstracto la noción de contexto utilizando una cadena que modela las últimas llamadas
  - Se conoce como ***k*-limiting**
- Para programas no recursivos se puede ser preciso
  - ¡Aunque no perfecto! Por qué?
    - (¡loops!)
- Para programas recursivos hay que aproximar varios contextos
  - Ejemplo: m_1. (g_1T)*.g_1F.doble
- Aunque parece una técnica interesante no se utiliza tanto
  - Costoso: multiplica la cantidad estados por la cantidad de caminos (¡puede ser exponencial!)
  - Depende fuertemente de tener un buen call graph

Sensibilidad al contexto

- Hay que distinguir entre los diferentes contextos de llamada a una función
- Varias técnicas
  - Cadenas de llamadas
  - **Inlining/Cloning**
  - Asumir / Verificar
  - Resúmenes por métodos

Inliling/Clonning

- Idea: Usar dataflow pero asegurandose que de no haya caminos incorrectos
- Cómo?
  - **Inlining**: Incluir el código del método llamado dentro del llamador.
    - Mirar el programa como un único procedimiento monolítico
    - Instanciar parametros de entrada y salida
  - **Clonning**: Crear una copia del método para cada llamada
    - Cada copia del método representa un contexto diferente

Inlining/Cloning

```
int divByX(int x) {
    [result := 10/x]1;
}

void caller1() {
   [x := 5]1;
   [y := divByX(x)]2;
   [y := divByX(5)]3;
   [y := divByX(0)]4;
}
```

Cloning

```
void caller1() {
   [x := 5]1;
   [y := divByX_1(x)]2;
   [y := divByX_2(5)]3;
   [y := divByX_3(0)]4;
}
int divByX_1(int x) {
    [result := 10/x]1;
}
int divByX_2(int x) {
    [result := 10/x]1;
}
int divByX_3(int x) {
    [result := 10/x]1;
}
```

Inlining

```
void caller1() {
   [x := 5]1;
   [y := divByX(x)]2;
      y := 10/x;
   [y := divByX(5)]3;
      y := 10/5;
   [y := divByX(0)]4;
     y := 10/0;
}
```

- Ventajas:
  - Mayor precisión
    - Una versión del método para cada contexto
- Desventajas
  - Mayor costo computacional
    - Explosión de código

Problemas Inliling / Cloning

- Uso de interfases

```
Interface I
{
   int getValue();
}
Class A implements I…
Class B implements I…

void process() {
   I i = null;
   if(…)
     i = new A();
   else
     i = new B()

    i.getValue();
}
```

```
Interface I
{
   int getValue();
}

void process(I i) {
   i.getValue();
}
```

- Recursión

```
void process(int v, int x) {
   if(v==0) return x;
   else return process(v-1,x*v)
}
```

Sensibilidad al contexto

- Hay que distinguir entre los diferentes contextos de llamada a una función
- Varias técnicas
  - Cadenas de llamadas
  - Inlining/Cloning
  - **Asumir / Verificar**
  - Resúmenes por métodos

Asumir/Verificar

- Idea: anotar el método con información sobre pide y lo que se espera el método
  - **Precondición**: Valores iniciales para todos los parámetros
  - **Postcondición**: Un valor al retorno (result)
  - Basadas en el conocimiento del programdor
    - Conocimiento individual sobre cada método.
  - O por "default"
    - Ejemplo: todos los equals
- Verificación
  - En el método anotado
    - Asumumir valores para parámetros
    - Verificar en el método anotado que resultado calculado $\subseteq$ asumido$_{result}$
  - En el método llamador
    - Verificar que arg$\subseteq$ asumido$_{arg}$
      - Parámetros actuales cumplen con lo asumido para parámetros formales
    - Usar el valor anotado para result.

Ejemplo

```
@NZ int divByX(@NZ int x)
{
    [result := 10/x]1;
}

void caller1() {
   [x := 5]1;
   [y := divByX(x)]2;
}
```

- Análisis de divByX

pos	x	Result
0	**NZ**	⊥
1	NZ	NZ

  - Cumple que σ(result)$\subseteq$NZ

- Análisis de caller 1

pos	x	y
0	⊥	⊥
1	NZ	⊥

- Verifica que σ(x)$\subseteq$NZ

Ejemplo

```
@NZ int doble(@NZ int x)
{
    [result := 2*x]1;
}

void caller1() {
   [x := 0]1;
   [y := doble(x)]2;
}
```

- Análisis de doble

pos	x	Result
0	**NZ**	⊥
1	NZ	NZ

  - Cumple que σ(result)⊆NZ

- Análisis de caller 1

pos	x	y
0	⊥	⊥
1	**Z**	⊥

- **σ(x)⊆NZ falla!**
- **No cumple con la precondición asumida**
- Es un falso positivo aunque el programa esté OK.

Ejemplo

```
@MZ int doble(@MZ int x)
{
    [result := 2*x]1;
}

void caller1() {
   [x := 5]1;
   [y := doble(x)]2;
   [z := 10/y]3;
}
```

- Análisis de doble

pos	x	Result
0	**MZ**	⊥
1	MZ	**MZ**

  - Cumple que σ(result)⊆MZ

- Análisis de caller 1

pos	x	y	Z
0	⊥	⊥	⊥
1	**NZ**	⊥	⊥

- **Verifica σ(x)⊆MZ**
- **Warning: div por cero!**
- Es un falso positivo!

Sensibilidad al contexto

- Hay que distinguir entre los diferentes contextos de llamada a una función
- Varias técnicas
  - Cadenas de llamadas
  - Inlining/Cloning
  - Asumir / Verificar
  - **Resúmenes por métodos**

Resumenes sentivos a contexto

- Idea:
  - Calcular un <u>único resumen por cada método</u>
  - Mapear información dataflow de entrada en información dataflow de salida
- Sensitivos a contexto
  - Dan <u>diferentes</u> resultados para diferentes entradas
  - Instanciar el resumen en el momento de la llamada

Generando Resumenes S a C

- Fuerza bruta
  - Analizar la función para cada posible valor de entrada
  - No escala porque los reticulados de los parámetros pueden ser enormes!
- Bajo demana
  - Analizar la función para cada valor que pueden tomar los parámetros, mirando las todas las llamadas en el código
  - Mejor pero aún impracticable
- Resúmenes abstractos
  - Representar de forma simbólica el efecto de la función sobre los elementos de los reticulados de los parámetros
  - Esto es lo que se hace en los análisis actuales

Resumenes bajo demanda

- Case x:NZ → result: NZ
- Case x:Z → result: Z

```
int doble(int x) {
    [result := 2*x]1;
}
void caller1() {
   [x := 5]1;
   [y := doble(x)]2;
   [z := 10/y]3
   [x := 0]4;
   [y := doble(x)]5;
}
```

- Análisis de caller 1

pos	x	Y	Z
0	⊥	⊥	⊥
1	**NZ**	⊥	⊥
3	NZ	NZ	**NZ**
4	**Z**	NZ	NZ

doble: caso x:NZ→result:NZ

pos	x	Result
0	NZ	⊥
1	NZ	**NZ**

doble: caso x:Z→result:Z

pos	x	Result
0	Z	⊥
1	Z	**Z**

Anotaciones sensibles a contexto

```
@Case("x:NZ -> result:NZ")
@Case("x:Z -> result:Z")
int doble(int x) {
    [result := 2*x]1;
}
void caller1() {
   [x := 5]1;
   [y := doble(x)]2;
   [z := 10/y]3
   [x := 0]4;
   [y := doble(x)]5;
}
```

- Verifico en caller 1
- Caso x:NZ

Pos	x	y	Z
1	**NZ**	⊥	⊥

- Verifico caller 1
- Caso x:Z

4	**Z**	NZ	NZ

Verifico : caso x:NZ→result:NZ

pos	x	Result
0	NZ	⊥
1	NZ	**NZ**

verifico: caso x:Z→result:Z

pos	x	Result
0	Z	⊥
1	Z	**Z**

Resumenes abstractos

Resumen:
Case x: A → result: A

```
int doble(int x) {
    [result := 2*x]1;
}
void caller1() {
   [x := 5]1;
   [y := doble(x)]2;
   [z := 10/y]3
   [x := 0]4;
   [y := doble(x)]5;
}
```

- Análisis de caller 1

pos	x	y	Z	
0	⊥	⊥	⊥	
1	**NZ**	⊥	⊥	
				A=NZ
3	NZ	NZ	**NZ**	
4	**Z**	NZ	NZ	A=Z

doble:

pos	x	Result
0	A	⊥
1	A	**A**

Calculando resumenes

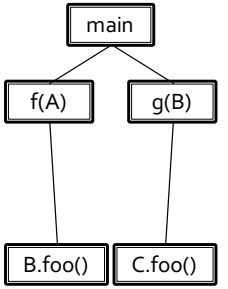
- Se recorre el call graph de la aplicación de forma Bottom-Up
  - Empezando por las hojas se asegura que no hay métodos a llamar
    - Realizar un análisis intraprocedural
    - Guardar resumen
  - Cuando un método llama a otro
    - Buscar resumen (esta por recorrido bottom up)
    - Instanciarlo con argumentos (top down)
- Para métodos recursivos requiere otro punto fijo
  - Sobre el subcomponente recursiva
    - Ciclo en el Call Graph

Call Graph

- Un "mapa" para saber que métodos analizar
  - Fundamental en programas orientados a objetos

```
static void main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  b3.foo();
}
```

```
class A {
  foo(){..}
}
class B extends A{
  foo() {…}
}
class C extends B{
  foo() {…}
}
class D extends B{
  foo(){…}
}
```

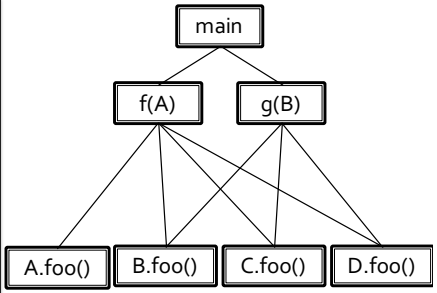


Call Graph

- Un "mapa" para saber que métodos analizar
  - Fundamental en programas orientados a objetos

```
static void main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  b3.foo();
}
```

```
class A {
  foo(){..}
}
class B extends A{
  foo() {…}
}
class C extends B{
  foo() {…}
}
class D extends B{
  foo(){…}
}
```



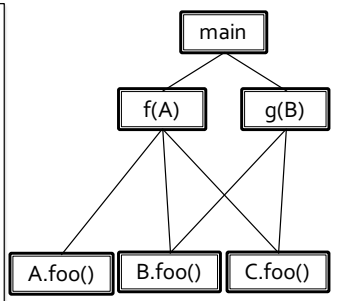
Calculado usando Class Hierarchy Analysis (CHA)

Call Graph

- Un "mapa" para saber que métodos analizar
  - Fundamental en programas orientados a objetos

```
static void main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  b3.foo();
}
```

```
class A {
  foo(){..}
}
class B extends A{
  foo() {…}
}
class C extends B{
  foo() {…}
}
class D extends B{
  foo(){…}
}
```



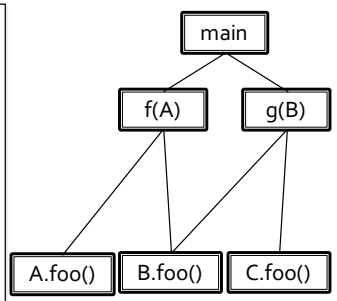
Calculado usando Rapid Type Analysis (RTA)

Call Graph

- Un "mapa" para saber que métodos analizar
  - Fundamental en programas orientados a objetos

```
static void main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  b3.foo();
}
```

```
class A {
  foo(){..}
}
class B extends A{
  foo() {…}
}
class C extends B{
  foo() {…}
}
class D extends B{
  foo(){…}
}
```



Calculado usando XTA

Sentividad a flujo vs. contexo

- Lo más preciso es ser sensitivo a ambos
- Pero si tuvieran que sacrificar sensitividad cual elegirían?
  - Flujo?
  - Contexo?
- En lenguajes OO se suele sacrificar flujo pero no contexto
  - Se asume que los métodos son cortos

Conclusiones

- Asunciones
  - Simple y eficiente
  - Impreciso (muy generales)
- Anotaciones
  - Requiere cierto esfuerzo
  - Suelen ser más precisas que las asunciones
  - Más eficiente que hacer un análisis interprocedural
- No requieren realizar un análisis de todo el programa!
- Interprocedural CFG
  - Fácil de implementar
  - Es impreciso
  - Puede ser costoso
  - As precise as simple
- Resúmenes
  - Muy precisos
  - Muy costosos si no se realiza una abstracción
- Requieren realizar un análisis de todo el programa

Bibliografia

- Compilers: Principles, Tecniques &Tools 2nd Edition: Aho, Lam, Sethi, Ullman
- Principles of Program Analysis . Flemming Nielson, Hanne Riis Nielson, Chris Hankin.
- Modern compiler implementation in Java. Andrew Appel. 2nd Edition.

- Cursos relacionados
  - Aldrich, Aiken, Palsberg, etc.