Control Flow Analysis

Objetivo: Analisar el flujo de llamados entre funciones. Es decir, modelar los posibles llamados a funciones en los distintos puntos del programa que utilizen funciones de primer orden (funciones como variables)

Para ello se buscara construir un Call graph que modele todos los posibles llamados a funciones. El reticulado que se usa es el del conjunto de partes de todos los nombres de funciones, por ejemplo $P(main, foo, inc, dec, ide), \subseteq$. Y las restricciones del dataflow se definen cómo:

CFA - Restricciones

Para cada nodo del AST introducimos una variable de restricción ⟦v⟧ definida de la siguiente manera:

- f ∈ ⟦f⟧
- X = E: ⟦E⟧ ⊆ ⟦X⟧
- $E(E_1,...,E_n)$: f ∈ ⟦E⟧ ⇒
  ⟦E_1⟧ ⊆ ⟦af_1⟧ ∧ … ∧ ⟦E_n⟧ ⊆ ⟦af_n⟧ ∧
  ⟦return f⟧ ⊆ ⟦$E(E_1…E_n)$⟧ ∀f con argumentos af_1,..., af_n

La última restricción sirve para llamados a funciones que no sabemos a priori a que función hace referencia (info que vamos obteniendo por el dataflow). Por ejemplo si la función a llamar viene como input. Luego si f pertenece a los posibles funciones a llamar, se define las restricciones por los parametros de la función y el valor de retorno.

Si ya se sabe que función se va a llamar podemos usar la restricción incondicional:

> We can optionally also introduce a simpler rule for the special case where the function being called is given directly by name (as in simple function calls before we added first-class functions to TIP). For a direct function call $f(E_1, \ldots, E_n)$ where f is a function with arguments $a_f^1, \ldots, a_f^n$ and return expression E_f', we have this (unconditional) constraint:
>
> $$[\![E_1]\!] \subseteq [\![a_f^1]\!] \wedge \cdots \wedge [\![E_n]\!] \subseteq [\![a_f^n]\!] \wedge [\![E_f']\!] \subseteq [\![E(E_1, \ldots, E_n)]\!]$$

Ejemplo:

CFA - Restricciones

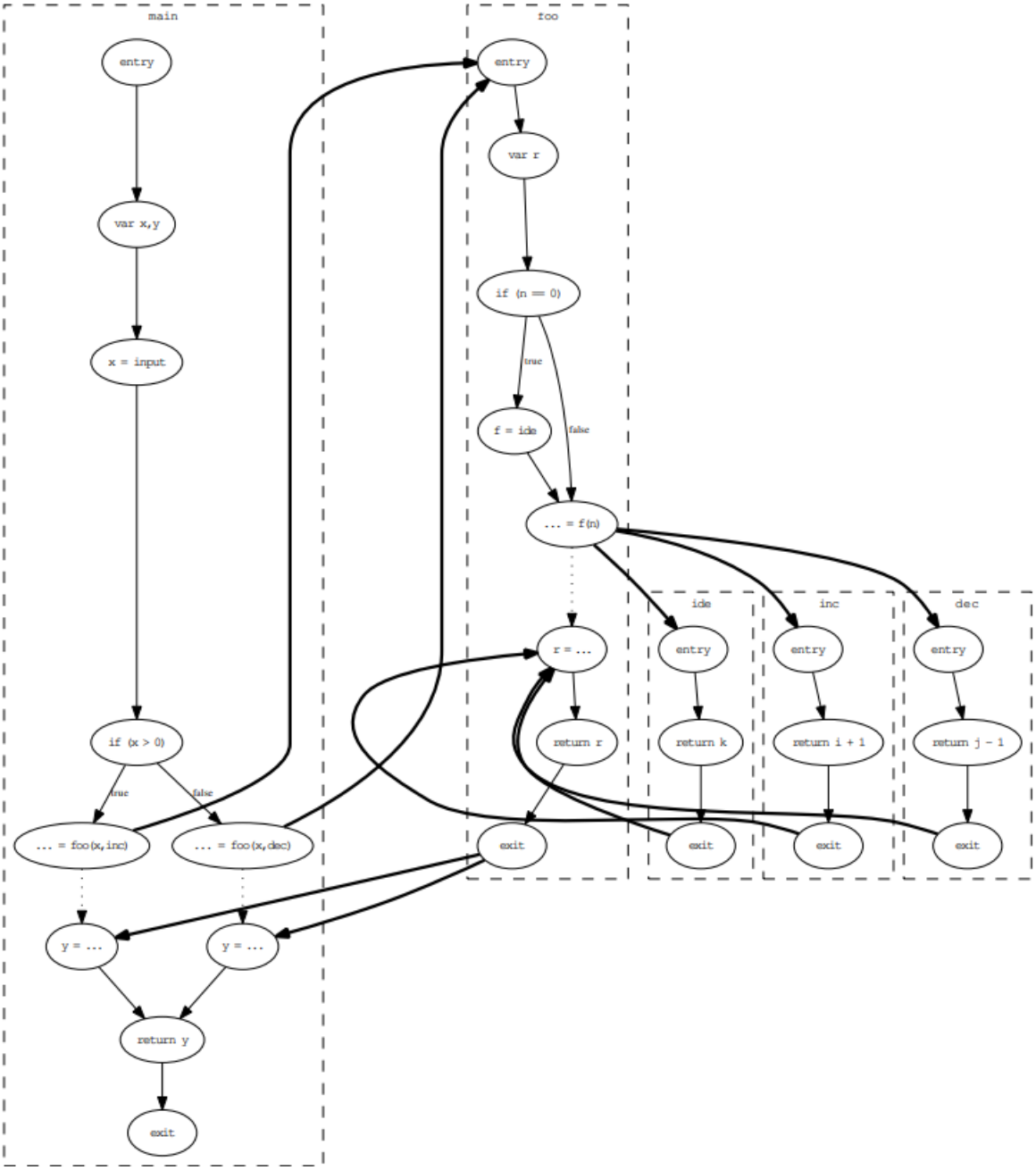
```
inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }

foo(n,f) {
  var r;
  if (n==0) { f=ide; }
  r = f(n);
  return r;
}

main() {
  var x,y;
  x = input;
  if (x>0) { y = foo(x,inc); } else { y = foo(x,dec); }
  return y;
}
```

inc ∈ ⟦inc⟧
dec ∈ ⟦dec⟧
ide ∈ ⟦ide⟧
⟦ide⟧ ⊆ ⟦f⟧
⟦f(n)⟧ ⊆ ⟦r⟧
inc ∈ ⟦f⟧ ⇒ ⟦n⟧ ⊆ ⟦i⟧ ∧ ⟦i+1⟧ ⊆ ⟦f(n)⟧
dec ∈ ⟦f⟧ ⇒ ⟦n⟧ ⊆ ⟦j⟧ ∧ ⟦j-1⟧ ⊆ ⟦f(n)⟧
ide ∈ ⟦f⟧ ⇒ ⟦n⟧ ⊆ ⟦k⟧ ∧ ⟦k⟧ ⊆ ⟦f(n)⟧
⟦input⟧ ⊆ ⟦x⟧
⟦foo(x,inc)⟧ ⊆ ⟦y⟧
⟦foo(x,dec)⟧ ⊆ ⟦y⟧
foo ∈ ⟦foo⟧
⟦x⟧ ⊆ ⟦n⟧ ∧ ⟦inc⟧ ⊆ ⟦f⟧ ∧ ⟦y⟧ ⊆ ⟦foo(x,inc)⟧
⟦x⟧ ⊆ ⟦n⟧ ∧ ⟦dec⟧ ⊆ ⟦f⟧ ∧ ⟦y⟧ ⊆ ⟦foo(x,dec)⟧
main ∈ ⟦main⟧

⟦inc⟧ = {inc}
⟦dec⟧ = {dec}
⟦ide⟧ = {ide}
⟦f⟧ = {inc, dec, ide}
⟦foo⟧ = {foo}



Para POO

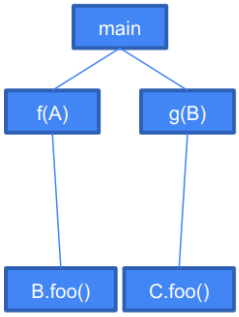
Objetivo: Analisar el flujo de llamado de métodos teniendo en cuenta la definición de clases. (polimorfismo y herencia)

Para el siguiente ejemplo la función f llama a $B.foo$ por el parámetro que le pasan, y g crea un objeto de tipo C y llama a $C.foo$

- 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(){…}
}
```

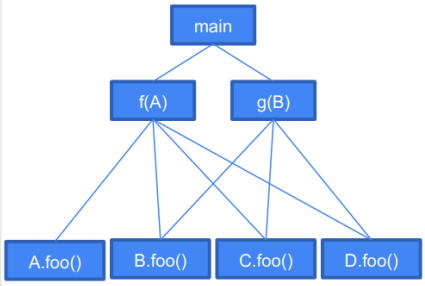


CHA recorre la estructura declarada de cada clase e infiere las posibles llamadas. Como f recibe un objeto de tipo A y este es superclase de B que a su vez es superclase de C y D, entonces se podría llamar a foo() de cualquiera de estas clases. Analogamente, en g se crea un objeto de tipo B que es superclase de C y D, por lo que puede llamar a foo() de dichas clases.

- 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(){…}
}
```



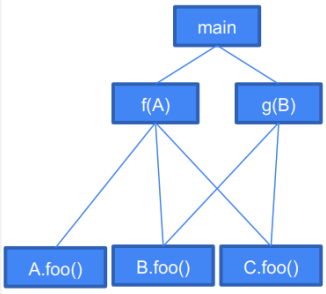
Calculado usando Class Hierarchy Analysis (CHA)

RTA recorre el programa buscando instancias de objetos y con ellos infiere los posibles llamados (además de usando los tipos de los objetos).

- 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)