Análisis Automático de Programas

Análisis dataflow intraprocedural

Diego Garbervetsky
Departamento de Computación
FCEyN – UBA

Analizadores Dataflow

- Herramientas que evalúan directamente el código (sin ejecutarlo).
- Modelan **todas** las posibles ejecuciones
  - Sobre una abstracción del estado
    - Menos preciso que análisis dinámico
    - Más "seguro"
- Errores "mecánicos" (difíciles de encontrar via Testing y/o inspecciones):
  - Errores de uso de memoria (Des-referencias a null, datos no inicializados).
  - "Leaks" de recursos (memoria no liberada, locks, archivos).
  - Vulnerabilidades (buffers overruns, datos no validados).
  - Violaciones de uso de un API o Framework, no respeto de encapsulamiento .
  - Excepciones no manejadas, concurrencia (race conditions), etc.
- Inferencia de propiedades
  - Especificaciones, invariantes, uso de recursos

Análisis Dataflow

Motivación:

- Encontrar propiedades/errores interesantes
  - x es null, x es copiada a y, y es des-referenciada
- Optimizar
  - Tiempo de ejecución, uso de memoria, etc
- **Garantías de corrección**
  - Sistemas críticos
    - Asegurarse de la no existencia de cierto tipos de errores
  - Optimizaciones requieren garantías
    - Ej: eliminar código muerto, mover un statement afuera de un loop
- **Automaticidad**
  - Reducción de costo
  - Impracticable de forma manual

Dataflow vs. Verificacion (SMT/MC)

- Dataflow
  - Sound error detection (si hay un error lo encuentran)
  - Suelen aproximar (falsos positivos)
  - Propiedades más simples
    - No siempre pero en gral si
- Verificación:
  - Requieren una especificación/propiedad a verificar
  - No son completos (por incapacidad del SMT)
    - A veces no suelen ser sound (por asunciones)
  - Uso principal : verificar propiedades funcionales
    - No siempre…

Análisis dataflow: usos

- **Para optimizar código.**
  - Detectar variables no usadas;
  - Eliminar código muerto.
  - Detectar expresiones usadas frecuentemente.
  - Descubrir métodos sin efectos colaterales.
  - Des-referencias de objetos validas (evitar el chequeo de null).
  - …
- **También nos ayuda a entender programas.**
  - Inferir el tipo de una función.
  - Calculo pre/post, invariantes.
  - Uso de recursos
  - Reingeniería
    - Grafo de llamas de un programa OO
    - Obtención de modelos de comportamiento
    - …

Ejemplos

- Clásicos:
  - Live variable analysis, Reaching definitions, Available expressions, etc..
  - Para optimizar
- Safety / Program Understanding
  - Zero analysis
  - Null pointer
  - Intervalos: rango de arrays
  - Invariantes
- Base para otros análisis:
  - Points-to analysis
  - Call graph
  - Aliasing

Available expressions

- Detectar qué expresiones están disponibles en cada punto del programa
- Eliminar computos repetidos
- Una expresion (*x op y*) está *disponible* en un punto del programa si en **todos** los caminos que conducen al punto
  - *x op y* es calculada al menos una vez
  - *x* e *y* no son redefinidas desde el momento que se calculó *x op y*

```
…
{  }
int b = a + 2;
{ a + 2}
int c = b*b;
{ a + 2, b*b }
int d = c + 1;
{ a + 2, b*b, c+1}
c = 4;
{ a + 2, b*b }
if(b < c) b = 2;
else c = a+1;
{ a + 2 }
return d;
```

Live variable analysis

- Identificar qué variables están "vivas" (serán utilizadas más adelante)
  - *x* está *viva* desde que es definida hasta su **último** uso o redefinición
- Usos:
  - Asignación de variables a registros
  - Eliminación de código muerto
    - Eliminación de código relacionado a asignaciones a variables no vivas

```
…
{ a }
int b = a + 2;
{ b }
int c = b*b;
{ c }
int d = c + 1;
{ d }
c = 4;
{ d, c }
return d+c;
{ }
```

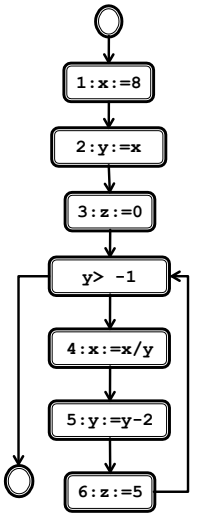
Zero analysis

- Inferir el valor que tiene cada variable
  - X es cero?
- Usos:
  - Detección temprana de bugs
    - División
    - Null
  - Propagación de constantes

```
x := 8;

y := x;

z := 0;

while y > -1 do

    x := x / y;

    y := y-2;

    z := 5;
```

Análisis Dataflow

- Es una de las técnicas de análisis estático más usadas.
- **Propósito**: Inferir **automáticamente** propiedades interesantes en un programa
  - En particular para cada punto de un programa
- **Principio**: Modelar la ejecución de un programa como la solución de un conjunto de ecuaciones que describen el flujo de valores a través de las instrucciones del mismo



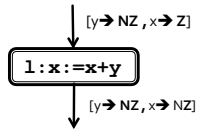
Analisis Dataflow: Intuición

- "ejecutar" el programa sobre valores abstractos.
- Colecta en cada punto del programa toda la información que fluye en ese punto.
  - Puede brindar información para cada posición del programa.
    - ¿Cuanto puede valer una variable Y al salir de la instrucción 5?
    - ¿puede "fluir" el valor null hacia x en alguna instrucción?
  - Puede distinguir el orden de las operaciones.
    - ¿Se leyó un archivo después de que éste fue cerrado?
- Es sensitivo a flujo
  - Requiere un grafo de control

Analisis Dataflow: elementos

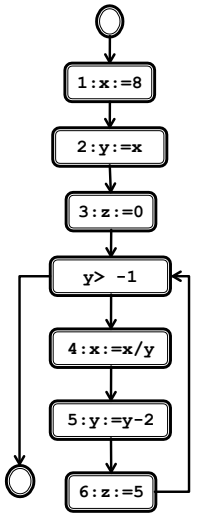
- **Grafo de control**: Representación del flujo de control del programa a analizar
- **Valores abstractos**: representan a la información que va a fluir

- **Función de transferencia**: Determina el efecto sobre el estado que tiene cada instrucción del programa
- **Ecuaciones de Dataflow**: Determinan cómo fluyen los valores abstractos de acuerdo al flujo de ejecución del programa



Grafo de control

- Muestra el orden de ejecución
- Se descomponen instrucciones en operaciones más simples
  - Ejemplo (3-address code)
    - a = b + c + d => t1 = b+c ; a = t1 + d
- Se eliminan las estructuras de iteración (while, for, repeat)
  - Se modelan con ciclos en el grafo
- Objetivo: analizar una operación simple por vez.



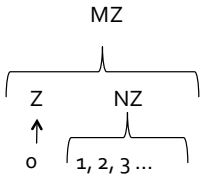
Valores abstractos

- Elegir una abstracción de acuerdo a la propiedad a inferir
  - ¿x puede ser cero?
  - La expresión a+b fue calculada previamente? Es necesario mantener la variable x en este punto del programa?
  - ¿De donde proviene este valor que asigno a x?
  - ¿Un archivo está abierto?
  - ¿La variable *x* es null cuando es desreferenciada? x e y representan el mismo objeto?

- **Clave:** El espacio de estados debe ser tratable computacionalmente
  - Deben conformar un **reticulado**.
  - Típicamente finito (al menos en altura)

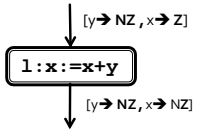
Abstracción requiere aproximación

- Abstraer => no manejar el estado completo
  - No manejamos la información real
- Ejemplo: Naturales positivos
  - 3 – 3 = 0
  - Abs(3) = NZ
  - Cuanto es NZ – NZ?
    - Respuesta es Z o NZ => MZ



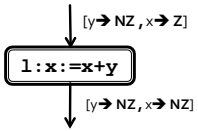
Función de transferencia

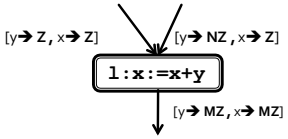
- Determina el efecto que tiene cada instrucción del programa sobre el estado abstracto
- Dada un nodo (instrucción) y un estado abstracto produce un nuevo estado abstracto
  - $F_{stmt}(\sigma) = \sigma'$
- Ejemplo:
  - $F_{x:=x+y}([y \rightarrow \mathbf{NZ}, x \rightarrow \mathbf{Z}]) = [y \rightarrow \mathbf{NZ}, x \rightarrow \mathbf{NZ}]$
- Algunas propiedades:
  - Debe ser monótona : x <= y → f(x) <= f(y)
  - Cerrada bajo composición (que f(f(x)) sea definible)



Analisis Dataflow: elementos

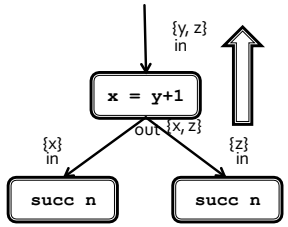
- Ecuaciones de dataflow:
  - Define cómo se calcula la salida de un nodo en función de sus entradas
  - En qué orden fluyen los datos y de qué manera se combinan
    - **Forward**: Desde el inicio del programa hacia el fin
      - Zero analysis, available expressions, etc
    - **Backward**: Desde el fin hacia el inicio
      - Live variables analysis
  - Cómo interpretar los datos
    - Qué hacer cuando hay datos que provienen de 2 lugares:
      - Aplicar el "supremo"/Aproximar para arriba: Análisis **MAY**
      - Aplicar el "ínfimo"/Aproximar para abajo: Análisis **MUST**



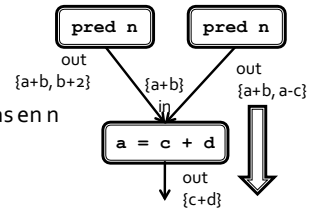


Ejemplos de ecuaciones

- Live variables analysis
  - in[n], out[n] = conjunto de variables
  - *transfer[n](X) = gen(n) U (X – kill(n))*
    - gen(n) = variables leidas en n
    - kill(n) = variables escritas en n
  - ⊕ = ∪ (de conjuntos)
  - ***out***[*n*] := ∪{ ***in***[*m*] | *m* ∈***succ***(*n*) }
  - ***in***[n] := ***transfer***[*n*](***out***[*n*])



- Available expressions
  - in[n], out[n] = conjunto de expresiones
  - *transfer[n](X) = gen(n) U (X ∩ kill(n))*
    - gen(n) = expression creada
    - kill(n) = exprs que contienen las variables escritas en n
  - ⊕ = ∩ (de conjuntos)
  - ***in***[*n*] := ∩ { ***out***[*m*] | *m* ∈***pred***(*n*) }
  - ***out***[n] := ***transfer***[*n*](***in***[*n*])



Framework Dataflow

Para cada nodo *n:*

- in[*n]:* valor/es en el punto del programa antes de *n*
- out[*n]:* valor/es en el punto del programa despues de *n*
- transfer[n]*:* operación a aplicar sobre el valor en el nodo *n*

Para cada análisis:

- ⊕**:** Operación de combinación (para combinar varios valores de entrada/salida)

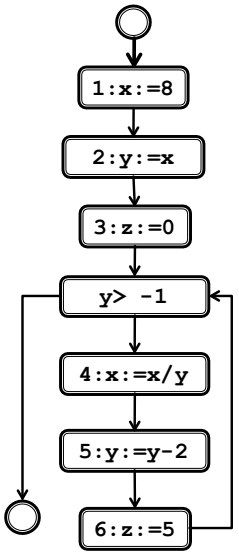
Recorrido\⊕	**∪ (MAY)**	**∩ (MUST)**
Forward	Dado in[n], calcula out[n] Aplica transfer[n] a pred[n] La propiedad aplica a algún camino (**reaching defs, zero analysis**)	Dado in[n], calcula out[n] Aplica transfer[n] a pred[n] La propiedad aplica a todo camino (**available expressions**)
Backward	Dado out[n], calcula in[n] Aplica transfer[n] a succ[n] La propiedad aplica a algún camino (**live variable analysis**)	Dado out[n], calcula in[n] pplica transfer[n] a succ[n] La propiedad aplica a todo camino (**very busy expressions**)

Algoritmo iterativo

Calcula out[*n] para cada n ∈N:*
out[*n] :=* ⊥ (o top si es Must)
Repetir
  Para cada *n*
    in[*n*] := ⊕{ ***out***[*m*] | *m* ∈***pred***(*n*) }
    out[*n*] := ***transfer***[*n*](***in***[*n*])
Hasta que no haya más cambios

Zero analysis

```
x := 8;
y := x;
z := 0;
while y > -1 do
    x := x / y;
    y := y-2;
    z := 5;
```

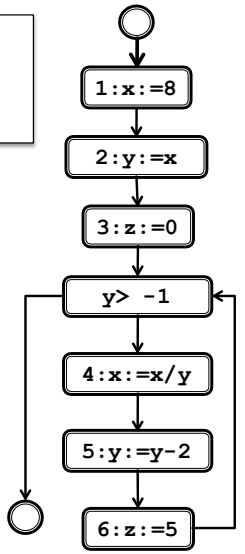


Zero analysis

reticulado : (var → {bot,Z,NZ,MZ})
- -bot = [] , top={x→MZ,y→MZ,z→MZ}
- - ∪ , ⊆

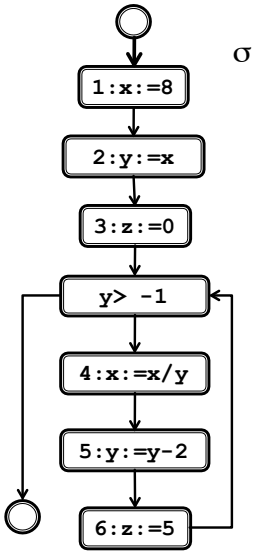
-**transfer**:
- -$F_{x:=e}(\sigma) = \sigma[x \leftarrow \sigma(e)]$
- -σ(Z+NZ)=NZ
- -σ(Z+Z)=Z
- -σ(Z-Z)=Z
- -σ(NZ-NZ)=MZ
- -…

in[*n*] := ∪{ ***out***[*m*] | *m* ∈***pred***(*n*) }
out[n] := ***transfer***[*n*](***in***[*n*])



Ejemplo

σ = []

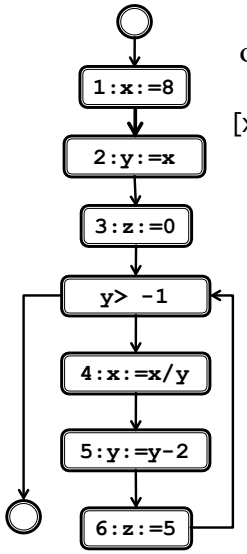


Ejemplo

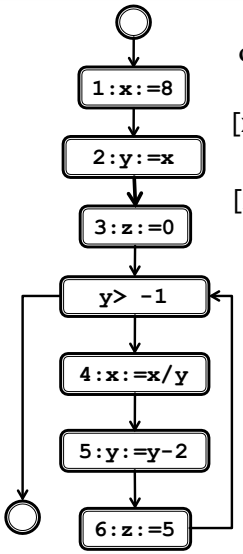
σ = []

α(8)=NZ

[x➔ **NZ**]



Ejemplo

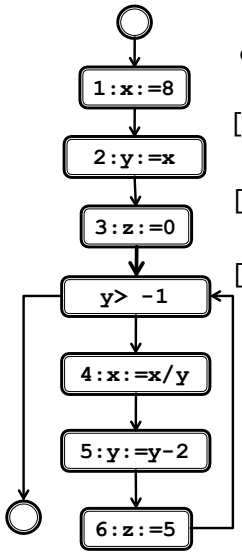


σ = []

[x➔ NZ]

[x➔ NZ, y ➔ **NZ**]

Ejemplo



σ = []

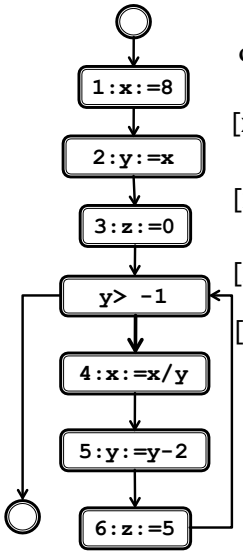
[x➔ NZ]

[x➔ NZ, y ➔ NZ]

α(o)=Z

[x➔ NZ, y ➔ NZ, z ➔ **Z**]

Ejemplo



σ = []

[x➔ NZ]

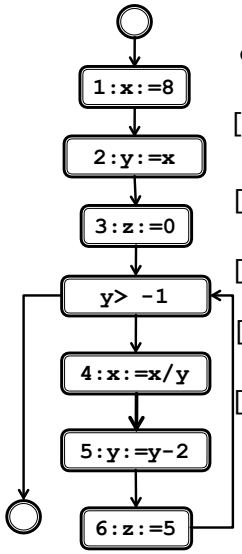
[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ NZ, z ➔ Z]

La consulta del *while* no realiza asignaciones

Ejemplo



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

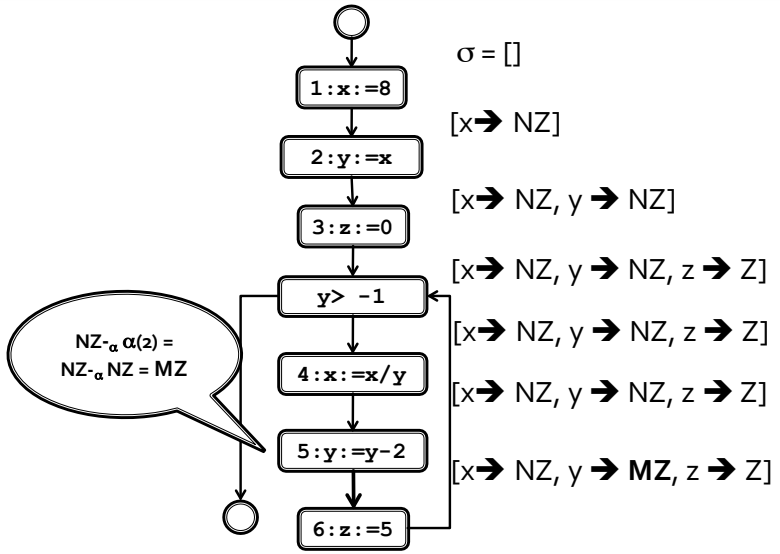
[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ **NZ**, y ➔ NZ, z ➔ Z]

$NZ /_{\alpha} Y = NZ$

Ejemplo



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

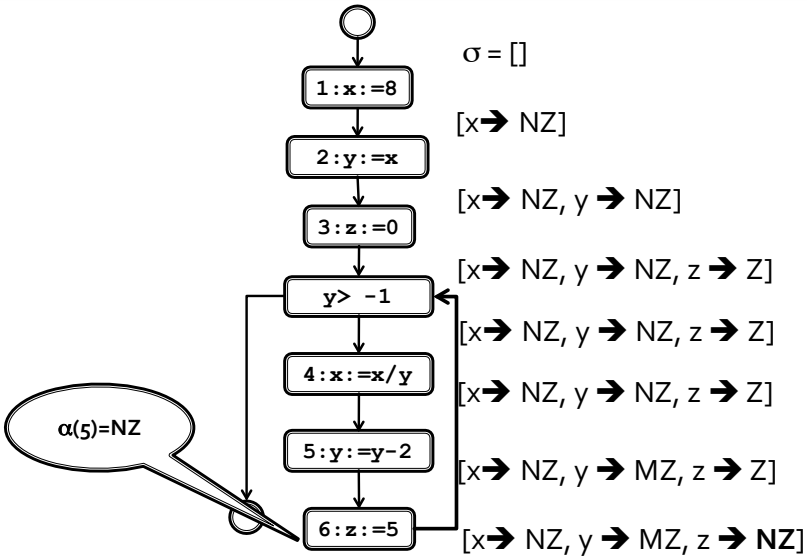
[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ **MZ**, z ➔ Z]

Ejemplo



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

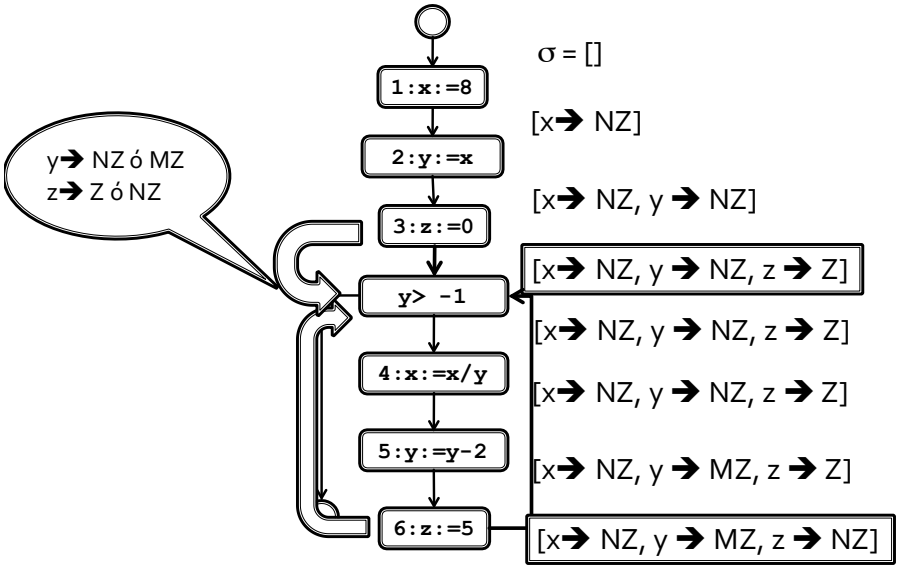
[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ **NZ**]

Ejemplo



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

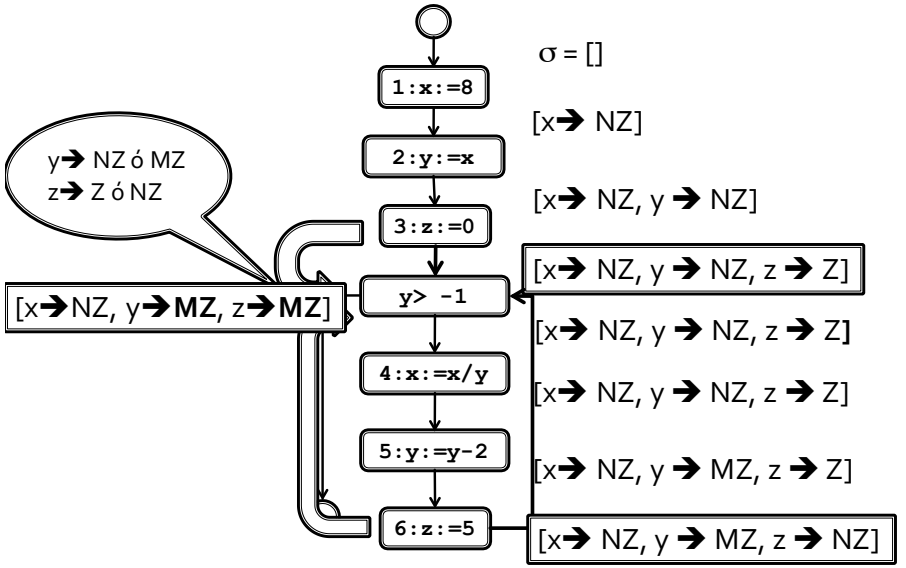
[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ NZ]

Ejemplo



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

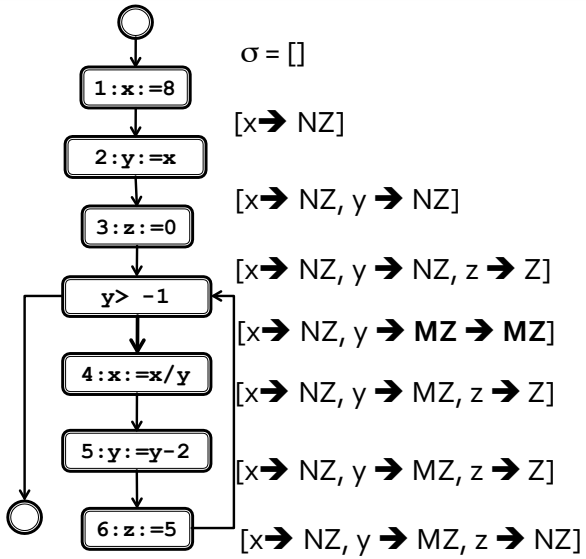
[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ NZ]

Ejemplo



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

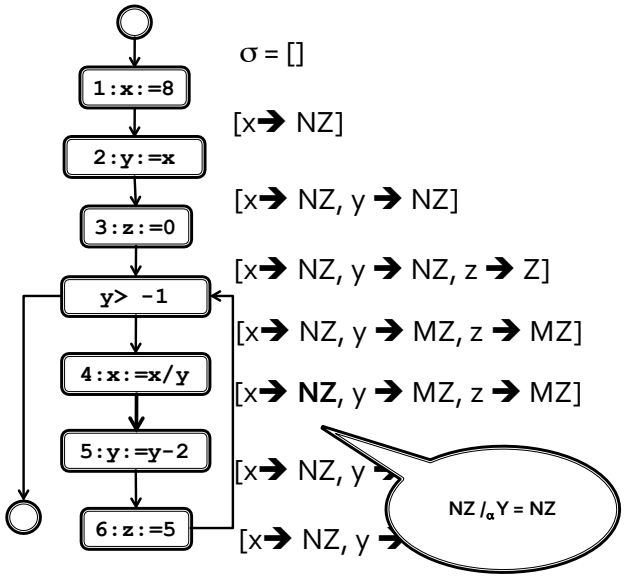
[x➔ NZ, y ➔ **MZ** ➔ **MZ**]

[x➔ NZ, y ➔ MZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ NZ]

Ejemplo



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

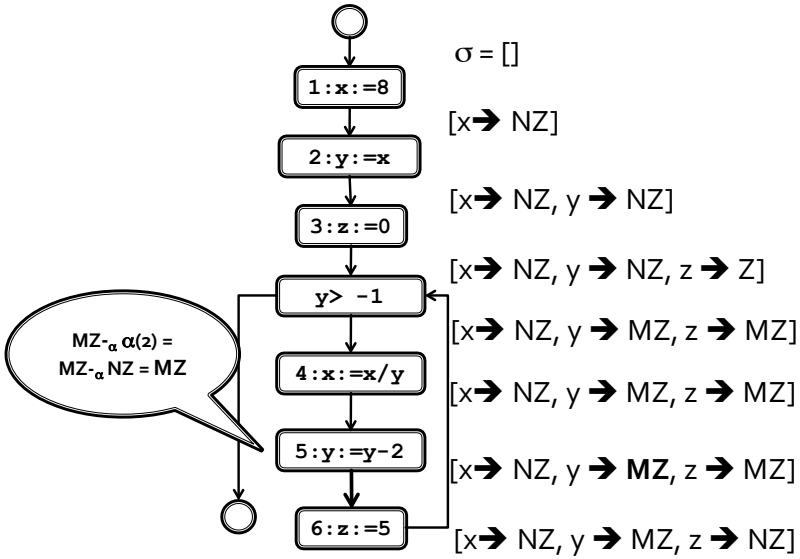
[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ **NZ**, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔

[x➔ NZ, y ➔

Ejemplo



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

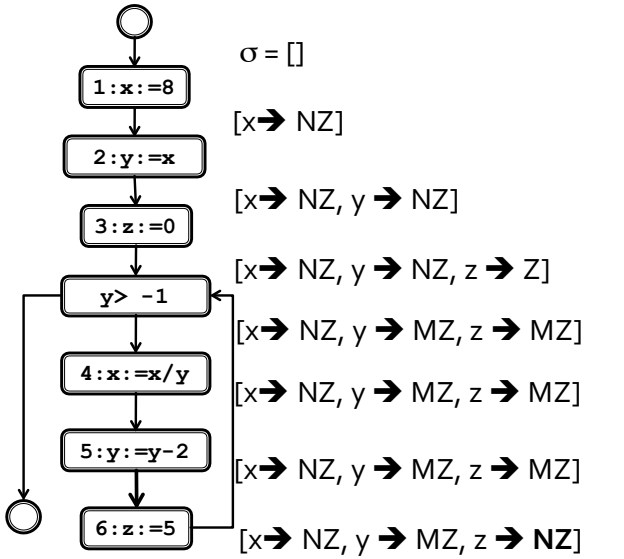
[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ **MZ**, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ NZ]

Ejemplo



σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ **NZ**]

Convergencia del análisis

Iteración 2

σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ NZ]

¡No hubo cambios! !Punto fijo!

Iteración 3

σ = []

[x➔ NZ]

[x➔ NZ, y ➔ NZ]

[x➔ NZ, y ➔ NZ, z ➔ Z]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ MZ]

[x➔ NZ, y ➔ MZ, z ➔ NZ]

Ejemplo: Utilizando el resultado

σ = []
`x := 8;`
σ = [x➔ NZ]
`y := x;`
σ = [x➔ NZ, y➔ NZ]
`z := 0;`
σ = [x➔ NZ, y➔ NZ, z➔ Z]
`while y > -1 do`
  σ = [x➔ NZ, **y➔ MZ**, z➔ MZ]
  `x := x / y;`
  σ = [x➔ NZ, y➔ MZ, z➔ MZ]
  `y := y-2;`
  σ = [x➔ NZ, y➔ **MZ**, z➔ MZ]
  `z := 5;`
  σ = [x➔ NZ, y➔ MZ, z➔ NZ]

Warning: Este programa ***quizás*** realiza una una división por cero

Usando el análisis de Ceros

Interpretación del resultado:

- Visitar cada expresión de la forma X/Y en el programa
- Mirar el resultado del cero análisis para ese punto para el divisor Y
  - Si Y = NZ, OK!

Propiedad fundamental: Dado un programa **P** y un análisis **A** que busca un tipo de errores E.
"Sound error detection":
•Si existe un error e en P => **el A lo encuentra**

Mejorando la precisión

- ¿Qué pasa con este ejemplo?
  - y = 3;
    y:= y-1;
    x = 6/y
  - El análisis es conservador…. Da warning cuando claramente es válida
- ¿Solución?: Utilizar una abstracción mas fina.
  - Ejemplo intervalos:
  - σ = []
  - y = 3;
  - σ = [y➔ [3,3]]
  - y:= y-1;
  - σ = [y➔ [2,2]]
  - x = 6/y
  - σ = [x➔ [3,3], y➔ [2,2]]
- OJO! Mayor precisión puede implicar más costo computacional.
  - En el ejemplo iteraría 9 veces en vez de 2.
  - Ejercicio!

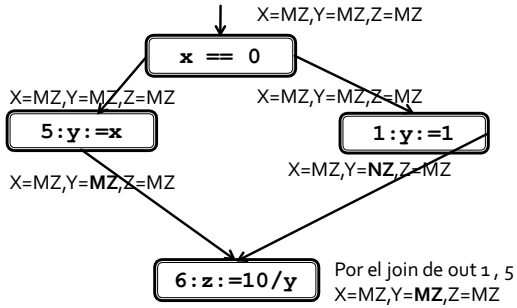
Algoritmo iterativo

Calcula out[*n] para cada n ∈N:*
out[*n*] := ⊥ (o top si es Must)
Repetir
  Para cada *n*
    in[*n*] := ⊕{ ***out***[*m*] | *m* ∈***pred***(*n*) }
    out[*n*] := ***transfer***[*n*](***in***[*n*])
Hasta que no haya más cambios

- Preguntas:
  - Produce una solución?
  - Produce la mejor solución posible?
  - El análisis termina?
- Depende de las propiedades del reticulado de valores y la función de transferencia

Problemas de precisión

- Qué pasa en el caso de los condicionales?



Función de transferencia:
- f(σ, [x := y]) = [x -> σ(y)] σ
- f(σ, [x := n]) = if n==o
  then [x-> Z]σ
  else [x-> NZ]σ
- f(σ, [x := y op z]) = [x- > MZ] σ
- f(σ, /* any other */) = σ

- El análisis no aprovecha la información proveniente del flujo

Mejorando Precisión

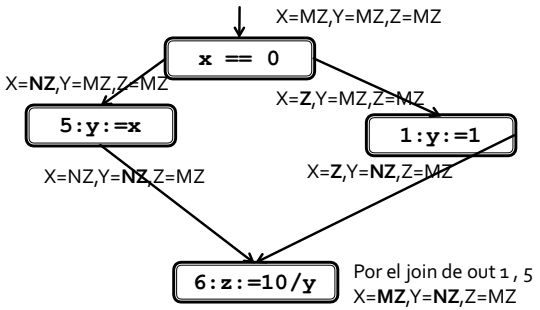
- Idea: Propagar información en las ramas

Función de transferencia:

- f(σ, [x := y]) = [x -> σ(y)] σ
- f(σ, [x := n]) = if n==o
  then [x-> Z]σ
  else [x-> NZ]σ
- f(σ, [x := y op z]) = [x- > MZ] σ
- f(σ, /* any other */) = σ

- fT(σ, [x == o]) = [x -> Z] σ
- fF(σ, [x == o]) = [x -> NZ] σ

En general:
- fT(σ, [x == y]) = [x -> σ(y)] σ
- fF(σ, [x == y]) = [x -> !σ(y)] σ



Algoritmo work-list

- Es más eficiente

Calcula out[*n] para cada n ∈N:*
out[*n*] := ⊥
work.add= {entry}
Mientras *work* no vacio. Hacer:
  n:= *work*.pop();
  in'[*n*] := ⊕{ ***out***[*m*] | *m* ∈***pred***(*n*) }
  out'[*n*] := ***transfer***[*n*](***in'***[*n*])
  si !(***out'***[*n*] ⊆***out***[*n*])
    para todo m ∈***succ***(*n*) *work.add(m);*
  out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

top si es Must

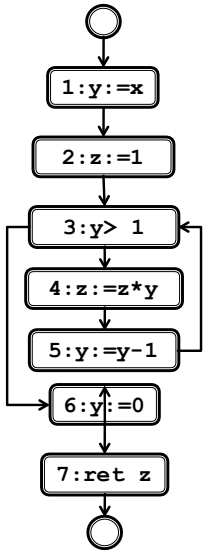
Exit si es backward.

Succ si es backward y cambiar in x out

Pred si es backward

Ejemplo: Live variables

```
y := x;
z := 1;
while y>1
{
    z := z * y;
    y := y - 1;
}
y := 0;
return z;
```

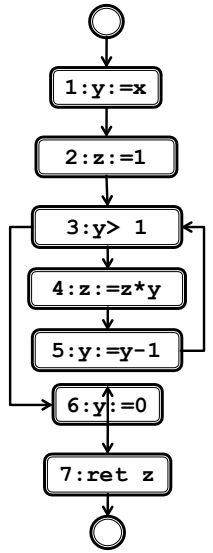


Live variable analysis

Reticulado : P({x,yz})
-bot = {}, top={x,y,z}, ∪ , ⊆

-Transfer:
- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x:=e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, otro caso) = σ

out[*n*] := ∪{ ***in***[*m*] | *m* ∈***succ***(*n*) }
in[n] := ***transfer***[*n*](***out***[*n*])



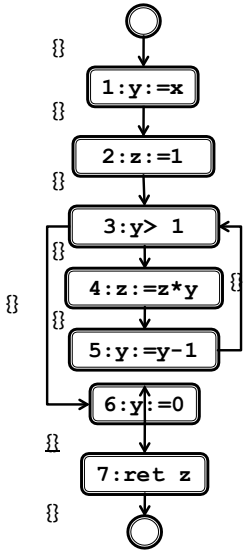
Live variable analysis

Reticulado : P({x, y, z})
Función de transferencia:
- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x:=e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, otro caso) = σ

out[*n*] := ∪{ ***in***[*m*] | *m* ∈***succ***(*n*) }
in[n] := ***F***(***out***[*n*],*stm*[*n*])

Calcula in[*n*] *para cada n* ∈*N:*
out[*n*] := ⊥
work.add= {exit}
Mientras *work* no vacio. Hacer:
n:= *work*.pop();
out'[*n*] := ∪{ ***in***[*m*] | *m* ∈***succ***(*n*) }
in'[*n*] := ***transfer***[*n*](***out'***[*n*])
si !(***in'***[*n*] ⊆***in***[*n*])
para todo m ∈***pred***(*n*) *work.add(m);*
out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {exit}



Live variable analysis

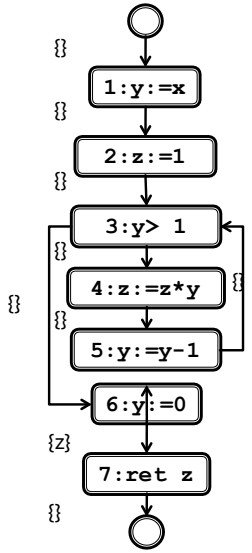
Reticulado : P({x, y, z})
Función de transferencia:
- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x:=e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, otro caso) = σ

out[*n*] := ∪{ ***in***[*m*] | *m* ∈***succ***(*n*) }
in[n] := ***F***(***out***[*n*],*stm*[*n*])

Calcula in[*n*] *para cada n* ∈*N:*
out[*n*] := ⊥
work.add= {exit}
Mientras *work* no vacio. Hacer:
n:= *work*.pop();
out'[*n*] := ∪{ ***in***[*m*] | *m* ∈***succ***(*n*) }
in'[*n*] := ***transfer***[*n*](***out'***[*n*])
si !(***in'***[*n*] ⊆***in***[*n*])
para todo m ∈***pred***(*n*) *work.add(m);*
out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {exit}
n = exit

out'[7] = {}
in' [7]= {z}
work' = {6}



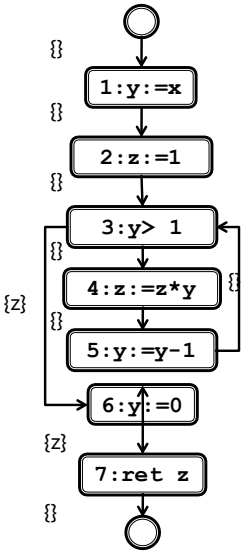
Live variable analysis

Reticulado : P({x, y, z})
Función de transferencia:
- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x:=e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, otro caso) = σ

out[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
in[n] := ***F***(***out***[*n*],*stm*[*n*])

Calcula in[*n*] *para cada n ∈ N:*
out[*n*] := ⊥
work.add= {exit}
Mientras *work* no vacio. Hacer:
  n:= *work*.pop();
  out'[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
  in'[*n*] := ***transfer***[*n*](***out'***[*n*])
  si !(***in'***[*n*] ⊆ ***in***[*n*])
    para todo m ∈ ***pred***(*n*) *work.add(m);*
  out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {6}
n = 6
out'[6] = {z}
in' [6]= {z}
work' = {3}



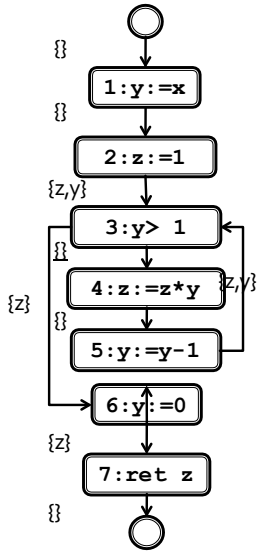
Live variable analysis

Reticulado : P({x, y, z})
Función de transferencia:
- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x:=e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, otro caso) = σ

out[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
in[n] := ***F***(***out***[*n*],*stm*[*n*])

Calcula in[*n*] *para cada n ∈ N:*
out[*n*] := ⊥
work.add= {exit}
Mientras *work* no vacio. Hacer:
  n:= *work*.pop();
  out'[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
  in'[*n*] := ***transfer***[*n*](***out'***[*n*])
  si !(***in'***[*n*] ⊆ ***in***[*n*])
    para todo m ∈ ***pred***(*n*) *work.add(m);*
  out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {3}
n = 3
out'[3] = {z} (de 6 y 4)
in'[3] = {z,y}
work' = {2,5}



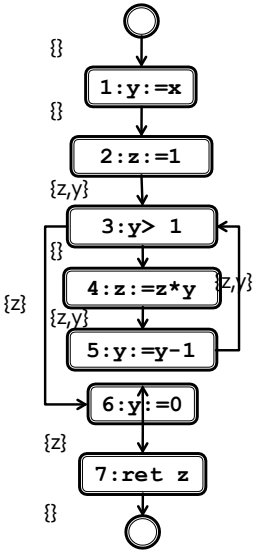
Live variable analysis

Reticulado : P({x, y, z})
Función de transferencia:
- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x:=e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, otro caso) = σ

out[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
in[n] := ***F***(***out***[*n*],*stm*[*n*])

Calcula in[*n*] *para cada n ∈ N:*
out[*n*] := ⊥
work.add= {exit}
Mientras *work* no vacio. Hacer:
  n:= *work*.pop();
  out'[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
  in'[*n*] := ***transfer***[*n*](***out'***[*n*])
  si !(***in'***[*n*] ⊆ ***in***[*n*])
    para todo m ∈ ***pred***(*n*) *work.add(m);*
  out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {5}
n = 5
out'[5] = {z,y}
in' [5]= {z,y}
work' = {2,4}



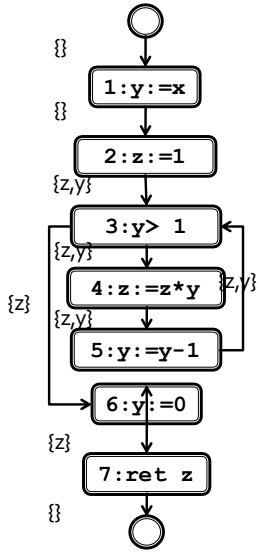
Live variable analysis

Reticulado : P({x, y, z})
Función de transferencia:
- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x:=e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, otro caso) = σ

out[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
in[n] := ***F***(***out***[*n*],*stm*[*n*])

Calcula in[*n*] *para cada n ∈ N:*
out[*n*] := ⊥
work.add= {exit}
Mientras *work* no vacio. Hacer:
  n:= *work*.pop();
  out'[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
  in'[*n*] := ***transfer***[*n*](***out'***[*n*])
  si !(***in'***[*n*] ⊆ ***in***[*n*])
    para todo m ∈ ***pred***(*n*) *work.add(m);*
  out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {2,4}
n = 4
out'[4] = {z,y}
in' [4]= {z,y}
work' = {2,3}



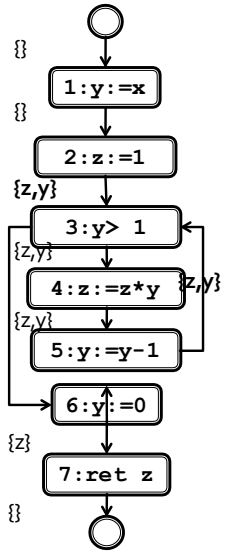
Live variable analysis

Reticulado : P({x, y, z})
Función de transferencia:
- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x:=e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, otro caso) = σ

out[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
in[n] := ***F(out***[*n*],*stm*[*n*])

Calcula in[*n*] *para cada n ∈N:*
out[*n*] := ⊥
work.add= {exit}
Mientras *work* no vacio. Hacer:
  n:= *work*.pop();
  out'[*n*] := ∪{ ***in***[*m*] | *m* ∈***succ***(*n*) }
  in'[*n*] := ***transfer***[*n*](***out'***[*n*])
  si !(***in'***[*n*] ⊆***in***[*n*])
    para todo m ∈***pred***(*n*) work.add(*m*);
  out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {2,3}
n = 3
out'[3] = {z,y} (de 6 y 4)
in' [3]= {z,y}
work' = {2}



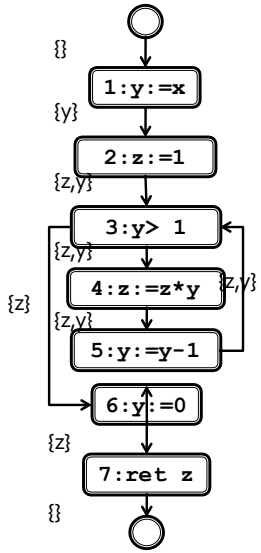
Live variable analysis

Reticulado : P({x, y, z})
Función de transferencia:
- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x:=e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, otro caso) = σ

out[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
in[n] := ***F(out***[*n*],*stm*[*n*])

Calcula in[*n*] *para cada n ∈N:*
out[*n*] := ⊥
work.add= {exit}
Mientras *work* no vacio. Hacer:
  n:= *work*.pop();
  out'[*n*] := ∪{ ***in***[*m*] | *m* ∈***succ***(*n*) }
  in'[*n*] := ***transfer***[*n*](***out'***[*n*])
  si !(***in'***[*n*] ⊆***in***[*n*])
    para todo m ∈***pred***(*n*) work.add(*m*);
  out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {2}
n = 2
out'[2] = {z,y}
in' [2]= {y}
work' = {1}



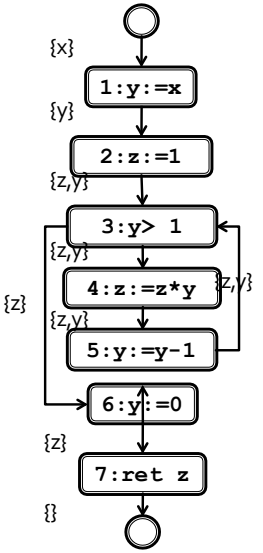
Live variable analysis

Reticulado : P({x, y, z})
Función de transferencia:
- F(σ, *e*) = σ ∪ vars(*e*)
- F(σ, *x:=e*) = (σ - { *x* }) ∪ vars(*e*)
- F(σ, otro caso) = σ

out[*n*] := ∪{ ***in***[*m*] | *m* ∈ ***succ***(*n*) }
in[n] := ***F(out***[*n*],*stm*[*n*])

Calcula in[*n*] *para cada n ∈N:*
out[*n*] := ⊥
work.add= {exit}
Mientras *work* no vacio. Hacer:
  n:= *work*.pop();
  out'[*n*] := ∪{ ***in***[*m*] | *m* ∈***succ***(*n*) }
  in'[*n*] := ***transfer***[*n*](***out'***[*n*])
  si !(***in'***[*n*] ⊆***in***[*n*])
    para todo m ∈***pred***(*n*) work.add(*m*);
  out[*n*] := ***out'***[*n*]; ***in***[*n*] := ***in'***[*n*];

work = {1}
n = 1
out'[1] = {y}
in' [1]= {x}
work' = {entry}



Terminación

- ¿Cómo sabemos que el algoritmo termina?
- Porque…
  - Las operaciones son monótonas
  - El dominio es finito
    - Live variables: conjunto de variables del programa
    - Available Expressions: conjunto de expresiones
    - Zero analysis: bot, Z,NZ, MZ
    - Etc.
  - Si el dominio no es finito aun se puede lograr terminación
    - Aceleración (widenning)

Sensitividad a flujo

- En un análisis sentitivo a flujo:
  - El órden de las instrucciones importa
    - Ejemplo: Zero analysis
- En un análisis no sensitivo
  - El órden de las instrucciones **no** importa
  - El resultado es el mismo sin importar cómo están ubicadas las instrucciones
    - Ej: inferencia de tipos en lenguajes tipados

Ejemplo de análisis no sensitivo

- Variables modificadas por un procedimiento
  - $M(x := e) = \{ x \}$
  - $M(S_1;S_2) = M(S_1) \cup M(S_2)$
- Notar que $M(S_1;S_2) = M(S_2;S_1)$

Comparando SF vs. no SF

- Los análisis sensitivos requieren un modelo del estado del programa
  - En cada punto del programa
  - Son más precisos, pero no escalan
- Los análisis insensitivos requieren **sólo** un estado global
  - Por ejemplo: El conjunto de variables modificadas
  - Son menos precisos, pero escalan muy bien

Sensibilidad

- Sensibilidad: que aspecto del código vamos a tomar en cuenta?
  - Orden de las instrucciones? Flow sensitive
  - Call Stack, parametros? Context sensitive
  - Saltos condicionales? Path sentivive

Análisis	**Sensibilidad**
Tipado	Insensible
Dataflow	Flow
Model Checking	Flow y Path
Points-to	C :Flow Java: Flow insensitive pero context sensitive

Lo que viene…

- **Análisis interprocedural y análisis basado en tipos**
  - Análisis dataflow interprocedural
  - Points-to analysis