

Weakest-Precondition of Unstructured Programs

Autores del paper:

Mike Barnett y K. Rustan M. Leino (Microsoft Research)

Presentado por:

Pablo Barenbaum, Pablo del Sel y Diego Dobniewski

Repaso de *verificación*

- El objetivo es analizar la *verification condition*: una fórmula lógica cuya validez significa que el programa es válido respecto de su especificación
- La fórmula será input del demostrador automático
- Es importante mantener baja la complejidad de la fórmula

De qué vamos a hablar

- Unstructured Programs
- Cálculo de Weakest Preconditions
- Evitar redundancia en la *Verification Condition*
- Programas *Pasivos* y *Single Assignment*
- Aplicación en ciclos *while*

Unstructured Programs

- Son programas que utilizan el GOTO como estructura de control
- Pueden generar flujos más variados que con *while*

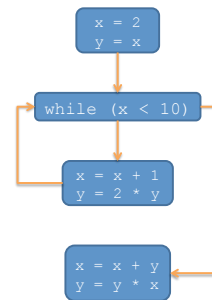
Unstructured Programs

- Ejemplo de un programa estructurado:

```
x = 2
y = x

while (x < 10) {
  x = x + 1
  y = 2 * y
}

x = x + y
y = y * x
```



Unstructured Programs

- Ejemplo de un programa **no** estructurado:

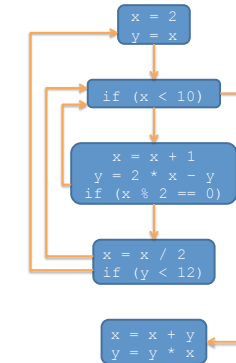
```
1: x = 2
2: y = x

3: if(x < 10) goto 4
   else goto 9

4: x = x + 1
5: y = 2 * x - y
6: if (x % 2 == 0) goto 7
   else goto 3

7: x = x / 2
8: if (y < 12) goto 1
   else goto 3

9: x = x + y
10: y = y * x
```



Los *GOTOs* no existen, pero que los hay, los hay

- Por qué queremos verificar código con *GOTOs*?
- Muchos lenguajes (sobre todo los que heredan de C) mantienen la instrucción *GOTO*.
- Los lenguajes de bajo nivel y los lenguajes *intermedios* de las máquinas virtuales también lo utilizan para representar:
 - Ciclos, *break*, *continue*
 - Excepciones

El lenguaje

- Basado en el lenguaje *intermedio* usado por el verificador de *Spec#* (*BoogiePL*).
- Utiliza variables (sin anotaciones de tipo)
- Las operaciones básicas son:
 - `varld := expresión` (asignación)
 - `havoc varld` (refresco del valor de una variable)
 - `assert expresión` (condición)
 - `assume expresión` (afirmación)
 - `skip` (no operación)

Ejemplos de las operaciones

- Asignación

```
// estadopre: x = 0, y = 3, ...
x := 5
```

Ejemplos de las operaciones

- Asignación

```
// estadopre: x = 0, y = 3, ...
x := 5
// estadopost: x = 5, y = 3, ...
```

Ejemplos de las operaciones

- Havoc
 - x toma un valor **arbitrario**

```
// estadopre: x = 0, y = 3, ...
havoc x
```

Ejemplos de las operaciones

- Havoc
 - x toma un valor **arbitrario**

```
// estadopre: x = 0, y = 3, ...
havoc x
// estadopost: x = -1, y = 3, ...
```

Ejemplos de las operaciones

- Assert
 - Puede dar origen a un estado de **no verificación**

```
// estadopre: x = 0, y = 3, ...
assert x >= 0
```

Ejemplos de las operaciones

- Assert
 - Puede dar origen a un estado de **no verificación**

```
// estadopre: x = 0, y = 3, ...
assert x >= 0
// estadopost: x = 0, y = 3, ...
```

Ejemplos de las operaciones

- Assert
 - Puede dar origen a un estado de **no verificación**

```
// estadopre: x = 0, y = 3, ...
assert x > 0
```

Ejemplos de las operaciones

- Assert
 - Puede dar origen a un estado de **no verificación**

```
// estadopre: x = 0, y = 3, ...
assert x > 0
```

no se pudo verificar la correctitud del programa

Ejemplos de las operaciones

- Assume
 - Puede dar origen a estados insatisfacibles, que no perjudican la verificación

```
// estadopre: x = 0, y = 3, ...
assume x >= 0
```

Ejemplos de las operaciones

- Assume
 - Puede dar origen a estados insatisfacibles, que no perjudican la verificación

```
// estadopre: x = 0, y = 3, ...
assume x >= 0
// estadopost: x = 0, y = 3, ...
```

Ejemplos de las operaciones

- Assume
 - Puede dar origen a estados insatisfacibles, que no perjudican la verificación

```
// estadopre: x = 0, y = 3, ...
assume x > 0
```

Ejemplos de las operaciones

- Assume
 - Puede dar origen a estados insatisfacibles, que no perjudican la verificación

```
// estadopre: x = 0, y = 3, ...
assume x > 0
estado insatisfacible, el programa todavía se
puede verificar
```

Bloques

- Un bloque tiene un **label**, una lista de **operaciones**, y una lista de **sucesores**
 - El **label** es un identificador unívoco del bloque
 - La secuencia de operaciones $o_1; \dots; o_n$ que serán ejecutadas en orden
 - La lista de bloques sucesores **goto** b_1, \dots, b_m puede ser vacía

b_0 : $x := 1; \text{assert } x > 0; \text{goto } b_5, b_9$

Flujo del programa

b_i : $o_1; \dots; o_n; \text{goto } b_1, \dots, b_m$

- Permite el flujo de control desestructurado
- El primer bloque del programa lleva el label **Start**
- El programa ejecutará las operaciones o_1 a o_n en orden y **bifurcará** la ejecución continuando en cada uno de los bloques b_1, \dots, b_m

Flujo del programa

- Codificación del **if**:
if (**Cond**) {**S**} **else** {**T**}

Start: **skip; goto** Then, Else

Then: **assume** **Cond**; **S**; **goto** Fin

Else: **assume** **-Cond**; **T**; **goto** Fin

Fin: ...

Correctitud

- Un programa será considerado **correcto** siempre que a partir del bloque *Start* no se llegue nunca a un estado de **no verificación**.
- Los programas **correctos** pueden alcanzar estados insatisfacibles
 - Por ej: la rama falsa del if

Proceso de verificación

- Se utiliza la estrategia *backward*
- Se calcula la *weakest precondition (wp)* sobre programas *pasivos* (libres de asignaciones)

Weakest Precondition

- **Assert**

```
// estadopre: vale WP(assert P, Q): P ^ Q
assert P
// estadopost: vale Q
```

Weakest Precondition

- **Assume**

```
// estadopre: vale WP(assume P, Q): P => Q
assume P
// estadopost: vale Q
```

Weakest Precondition

- **Bloque**

```
// estadopre: vale WP(S;T, Q): WP(S, WP(T, Q))
S;
// estadointer: vale WP(T, Q)
T;
// estadopost: vale Q
```

Weakest Precondition

- **Goto**

// estado_{pre0}: vale **WP(S, Q)**

b₀: S; goto b₂

// estado_{pre1}: vale **WP(T, Q)**

b₁: T; goto b₂

// estado_{pre2}: vale Q

b₂: ...

Weakest Precondition

- La *verification condition* será:

... **WP(S, Q) ^ WP(T, Q)** ...

- Al ser redundante, el verificador debe calcular **Q** más de una vez

Weakest Precondition

- Veamos el caso del **if**

Start: **skip**; goto Then, Else

Then: **assume** Cond; S; goto **Fin**

Else: **assume** ¬Cond; T; goto **Fin**

Fin: ...

- Se define una variable auxiliar para cada bloque

Start_{ok} = WP(**skip**, **Then**_{ok} ^ **Else**_{ok})

Then_{ok} = WP(**assume** Cond; S, **Fin**_{ok})

Else_{ok} = WP(**assume** ¬ Cond; S, **Fin**_{ok})

Fin_{ok} = ...

- Si calculamos una vez cada una, eliminamos la redundancia.

Weakest Precondition

- Se reescribe la *verification condition*

Start_{ok} = WP(**skip**, **Then**_{ok} ^ **Else**_{ok}) ^

Then_{ok} = WP(**assume** Cond; S, **Fin**_{ok}) ^

Else_{ok} = WP(**assume** ¬ Cond; S, **Fin**_{ok}) ^

Fin_{ok} = ...

=> **Start**_{ok}

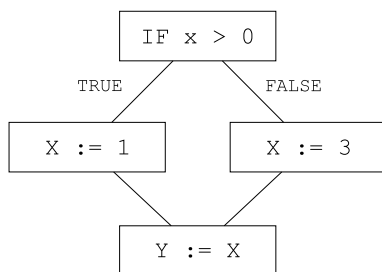
Single Assignment

- Se modifica el programa para lograr que variable sea asignada solamente una vez.
- Para cada variable X se crean nuevas variables X1, X2, etc...
- Cada aparición de X es remplazada por una de esas nuevas variables.
- Ej
 - $X := X + 1$
 - Se convierte en
 - $X_0 := X_1 + 1$

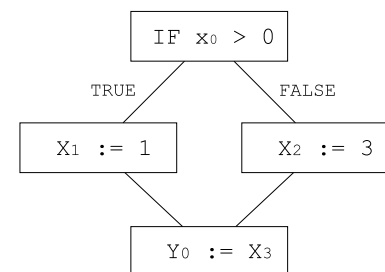
Passification

- Se reemplazan todas las asignaciones por assumes.
- Antes: $X_0 := Y_0$
- Después: $\text{assume } X_0 = Y_0$

SSA - IF



SSA - IF



$x_0 > 0 \Rightarrow X_3 = x_1$ AND
 $x_0 \leq 0 \Rightarrow X_3 = x_2$

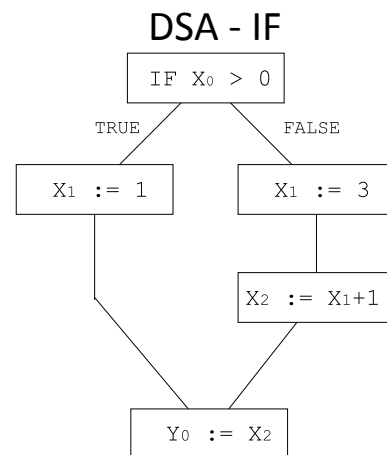
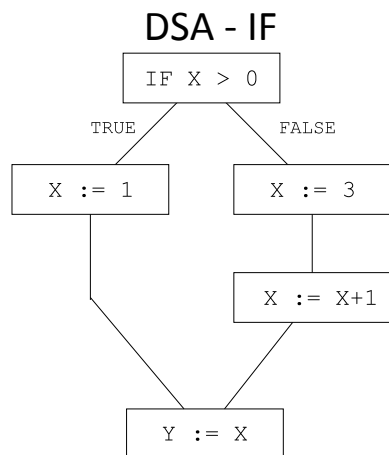
Dynamic single-assignment

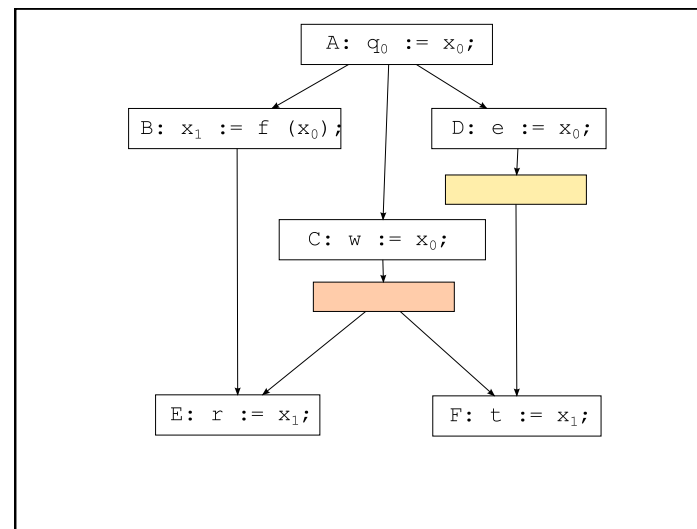
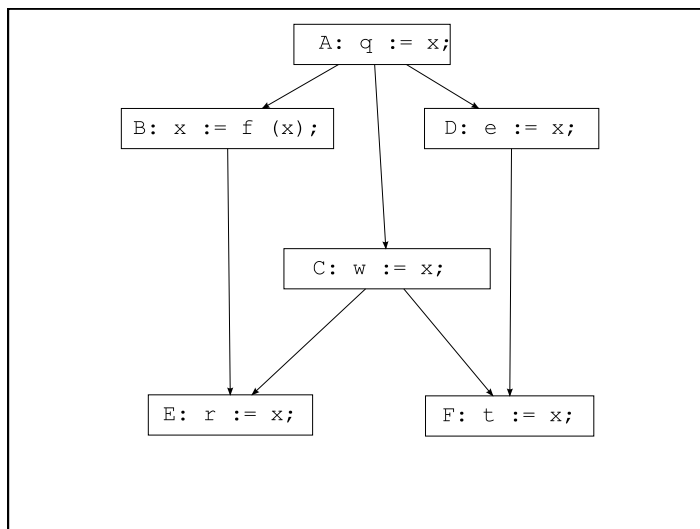
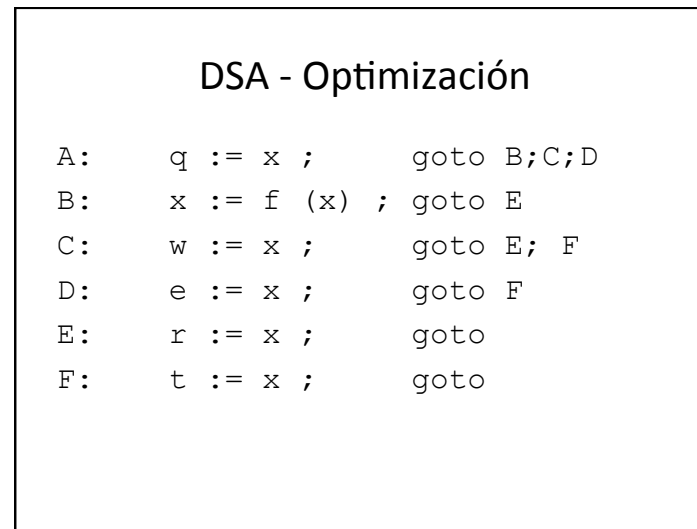
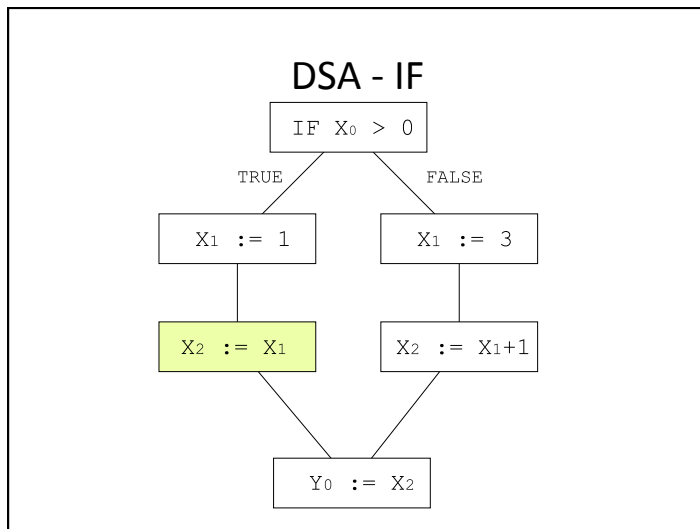
- Similar a SSA.
- Una variable puede tener mas de una definición, pero a lo sumo una puede ser usada en un programa.
- Evita redundancia, logrando una VC mas pequeña.

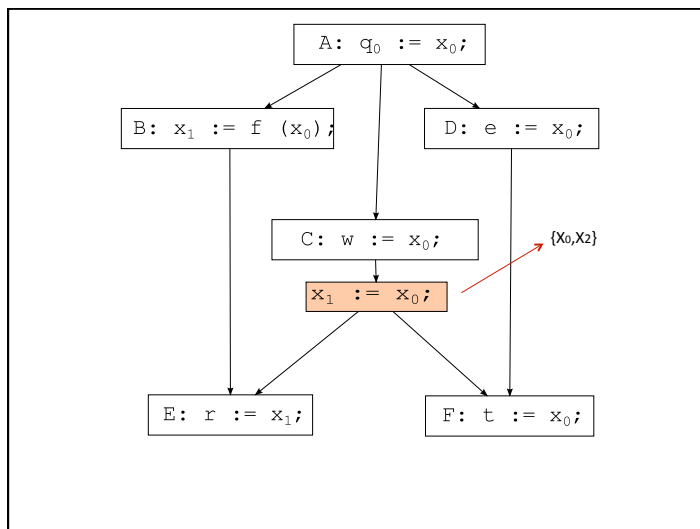
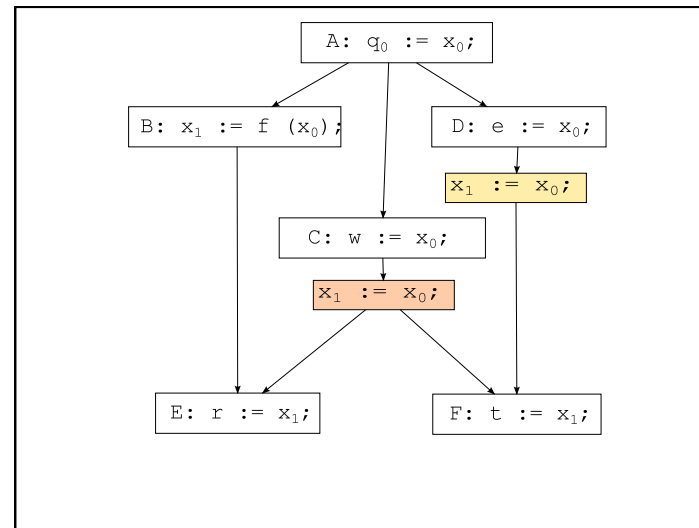
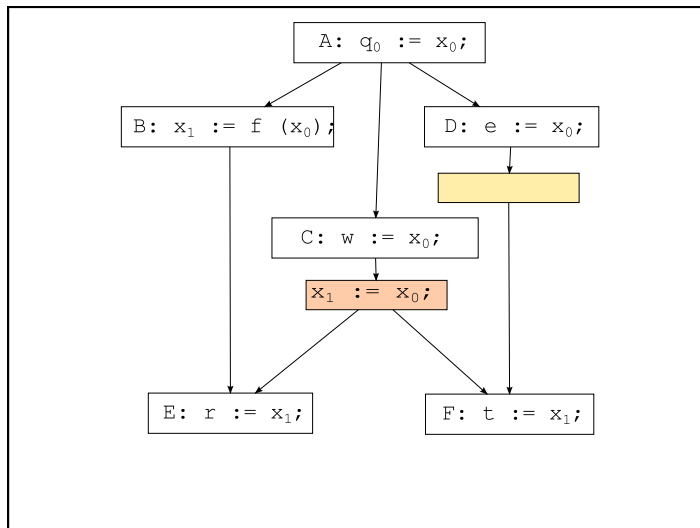
DSA - IF

```

A      : skip;      goto TRUE;FALSE
TRUE  : X := 1;    goto B
FALSE: X := 3 ;
        X := X+1;  goto B
B      : Y := X;
  
```







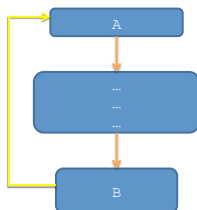
Ciclos - definiciones

- Un nodo A **domina** a otro B cuando todos los caminos que van a B pasan por A.

A diagram showing a path from node A to node B. The path consists of A, followed by three blue boxes representing intermediate nodes, and finally B. A curved arrow loops back from the path between the second and third blue boxes to node A, indicating a cycle.

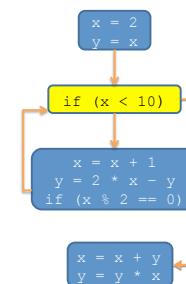
Ciclos - definiciones

- Back edge: es un eje en el control-flow graph tal que el nodo destino del eje domina el nodo de origen.



Ciclos - definiciones

- Loop head: Es el nodo destino de un back edge.



Ciclos - Identificación

- Se encuentran los back edges.
- Un back edge identifica inequívocamente a un ciclo.
- Cada natural loop es identificado por el par (L,B)

Ciclos - Transformacion

- La idea es llevar el CFG en uno acíclico.
- Se remueven todos los back edges.
- Verificamos el ciclo remplazándolo por una iteración genérica.

Ejemplo

```

int M(int x)
requires 100 <= x; // precondition
ensures result == 0; // postcondition
{
  while (0 < x)
    invariant 0 <= x; // loop invariant
    {
      x = x - 1;
    }
  return x;
}

```

Ejemplo

```

Start : assume 100 <= x ; // precondition
goto LoopHead ;
LoopHead : assert 0 <= x ; // loop invariant
goto Body;After ;
Body : assume 0 < x ; // loop guard
x := x - 1 ;
goto LoopHead ;
After : assume !(0 < x) ; // negation of guard
r := x ; // return statement
assert r = 0 ; // postcondition
goto ;

```

Ejemplo

```

Start : assume 100 <= x ;
assert 0 <= x ; // check inv.
goto LoopHead ;
LoopHead : havoc x ; // havoc loop targets
assume 0 <= x ; // assume inv.
goto Body;After ;
Body : assume 0 < x ;
x := x - 1 ;
assert 0 <= x ; // check inv.
goto ; // removed back edge
After : assume !(0 < x) ;
r := x ;
assert r = 0 ;
goto ;

```

FIN

¿Preguntas?