Análisis de Flujo

CI4722 – Lenguajes de Programación III

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Copyright ©2012-2016

UNIVERSIDAD SIMÓN BOLÍVAR

Eliminación de redundancia

- Minimizar el número de evaluaciones de las expresiones.
  - Considerar *todas* las secuencias posibles.
  - Observar las evaluaciones de `x + y`.
  - Dejar la cantidad mínima de evaluaciones, aprovechando una variable temporal que la contenga.

Eliminación de redundancia

- Minimizar el número de evaluaciones de las expresiones.
  - Considerar *todas* las secuencias posibles.
  - Observar las evaluaciones de `x + y`.
  - Dejar la cantidad mínima de evaluaciones, aprovechando una variable temporal que la contenga.
- Puede que aparezcan más *usos* de la expresión a través de la variable temporal, pero las *evaluaciones* se reducen al mínimo posible.
- Mejora el desempeño del código final –
  las expresiones se evalúan sólo si no hay alternativa.

UNIVERSIDAD SIMÓN BOLÍVAR

Eliminación de redundancia

- Encontrar los lugares adecuados en el grafo de flujo para evaluar la expresión requiere combinar varios análisis de flujo de datos.
- Redunda la redundancia
  - Sub-expresiones comunes.
  - Expresiones invariantes en ciclos.
  - Expresiones parcialmente redundantes – sólo por un camino.

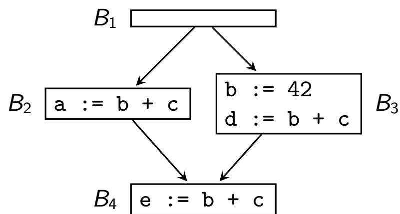
Eliminación de redundancia

- Encontrar los lugares adecuados en el grafo de flujo para evaluar la expresión requiere combinar varios análisis de flujo de datos.
- Redunda la redundancia
  - Sub-expresiones comunes.
  - Expresiones invariantes en ciclos.
  - Expresiones parcialmente redundantes – sólo por un camino.

Gotta catch'em all!

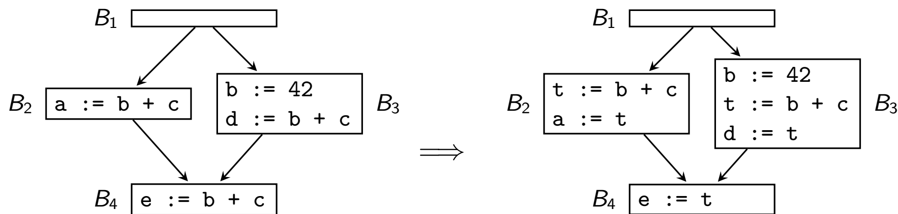
UNIVERSIDAD SIMÓN BOLÍVAR

Subexpresiones comunes globales



- `b + c` en B_4 es redundante – ha sido evaluada previamente, sin importar el flujo de control.

Subexpresiones comunes globales



- `b + c` en B_4 es redundante – ha sido evaluada previamente, sin importar el flujo de control.
- Calcular a la temporal `t` permite eliminar el cálculo en B_4

Subexpresiones comunes globales

- La expresión E es *completamente redundante* en un punto p, si
  - Ha sido calculada en *todos* los caminos que alcanzan p.
  - Las variables que participan en la expresión no han sido redefinidas después de cada evaluación.

Subexpresiones comunes globales

- La expresión E es *completamente redundante* en un punto p, si
  - Ha sido calculada en *todos* los caminos que alcanzan p.
  - Las variables que participan en la expresión no han sido redefinidas después de cada evaluación.
- El método sólo encuentra expresiones textualmente idénticas
  - Detectará que `t1` y `t2` tienen mismo valor en

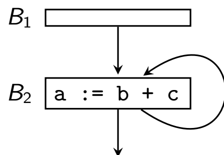
```
t1 := b  + c            t2 := b  + c
a  := t1 + d            e  := t2 + d
```

  - Pero no puede notar que `a` y `e` son la misma.
  - Siempre se puede aplicar la eliminación global repetidas veces hasta que no se encuentre nada que eliminar.

El problema de flujo de datos asociado se denomina
Análisis de Disponibilidad

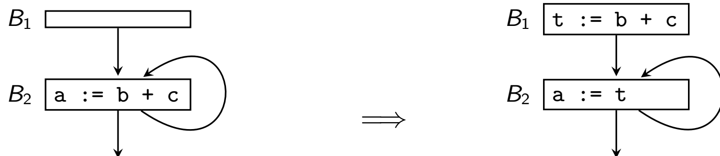
UNIVERSIDAD SIMÓN BOLÍVAR

Expresiones invariantes en ciclos



- `b + c` en B_2 es invariante – no cambia con las repeticiones.

Expresiones invariantes en ciclos



- `b + c` en B_2 es invariante – no cambia con las repeticiones.
- Calcular a la temporal `t` permite eliminar el cálculo en B_2

UNIVERSIDAD SIMÓN BOLÍVAR

Expresiones invariantes en ciclos

- La mejora está basada en "mover código" de un lugar a otro.
- El movimiento debe asegurar que **no** se ejecuta ninguna instrucción que no se hubiera ejecutado antes de hacer el movimiento.
- Supongamos que es posible salir del lazo sin ejecutar la instrucción invariante – entonces no podemos moverla.
  - Si la instrucción genera una excepción, al moverla se generaría en un sitio diferente al programa original...
  - ...y para los casos en que el ciclo terminaba antes, el programa "mejorado" ahora tarda más.

Expresiones invariantes en ciclos

- ¿Cómo hacerla posible *siempre*? – Reescribir los ciclos

```
while c {
  S
}
```

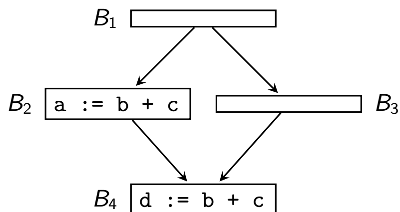
$\Longrightarrow$

```
if c {
  repeat {
    S;
  } until (not c)
}
```

permitiendo mover las invariantes justo antes del `repeat`, sin riesgo.

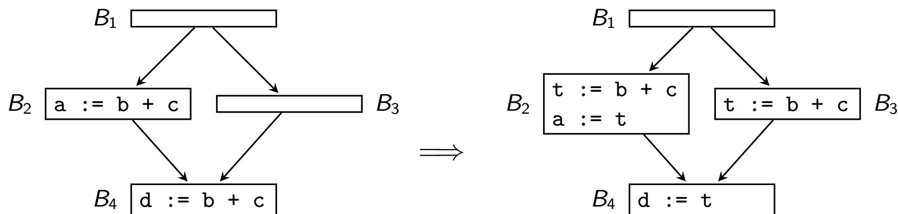
- La mejora se aplica varias veces, desde los ciclos más internos hacia los ciclos más externos.
  - Si una variable es invariante en un ciclo, las expresiones que le usan, posiblemente lo sean y se muevan con ella.
  - La máxima mejora ocurre cuando las expresiones se mueven tan "afuera" como sea posible.

Expresiones parcialmente redundantes



- `b + c` en B_4
  - Es redundante para el camino $B_1 \rightarrow B_2 \rightarrow B_4$
  - No es redundante para el camino $B_1 \rightarrow B_3 \rightarrow B_4$

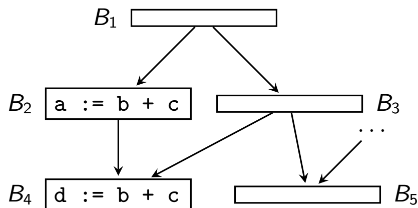
Expresiones parcialmente redundantes



- `b + c` en B_4
  - Es redundante para el camino $B_1 \rightarrow B_2 \rightarrow B_4$
  - No es redundante para el camino $B_1 \rightarrow B_3 \rightarrow B_4$
- Calcular a la temporal `t` permite eliminar el cálculo en B_4
- Es necesario incorporar el cálculo de `t` en B_3.

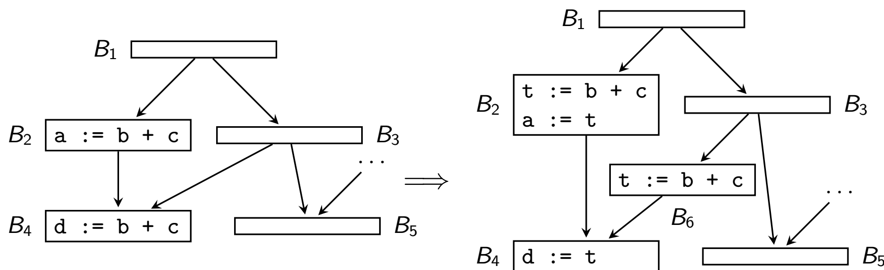
UNIVERSIDAD SIMÓN BOLÍVAR

¿Siempre se puede eliminar la redundancia?



- `b + c` en B_4 es redundante para el camino $B_1 \to B_2 \to B_4$
- No podemos agregarla en B_3 – cálculo adicional sobre $B_1 \to B_3 \to B_5$

¿Siempre se puede eliminar la redundancia?



- `b + c` en B_4 es redundante para el camino $B_1 \to B_2 \to B_4$
- No podemos agregarla en B_3 – cálculo adicional sobre $B_1 \to B_3 \to B_5$
- Agregamos el bloque B_6 *exclusivamente* para el cómputo, de manera que no afecte al camino $B_1 \to B_3 \to B_5$.

UNIVERSIDAD SIMÓN BOLÍVAR

Pudimos, reemplazando aristas críticas

- Tenemos una **arista crítica** en un grafo de flujo cuando:
  - Tiene origen en un nodo con más de un sucesor.
  - Tiene destino en un nodo con más de un predecesor.
- La arista entre B_3 y B_4 es crítica – también lo es entre B_3 y B_5.
- Agregar bloques *nuevos* sobre aristas críticas permitió ubicar la expresión en la forma deseada sin influir sobre otros caminos.

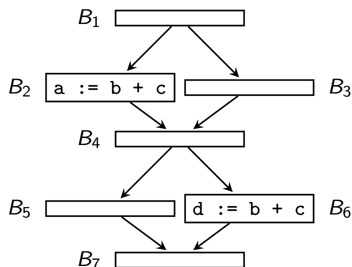
UNIVERSIDAD SIMÓN BOLÍVAR

Pudimos, reemplazando aristas críticas

- Tenemos una **arista crítica** en un grafo de flujo cuando:
  - Tiene origen en un nodo con más de un sucesor.
  - Tiene destino en un nodo con más de un predecesor.
- La arista entre B_3 y B_4 es crítica – también lo es entre B_3 y B_5.
- Agregar bloques *nuevos* sobre aristas críticas permitió ubicar la expresión en la forma deseada sin influir sobre otros caminos.

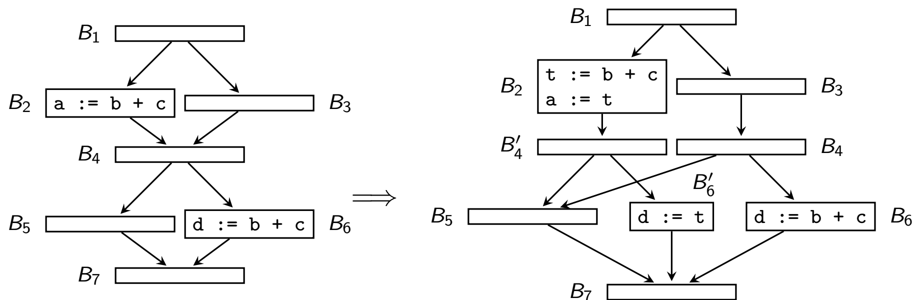
¿Será suficiente?

What if ifs become iffy?



- `b + c` en B_6 es redundante para el camino $B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_6$
- ¡No hay ningún punto ni arista *única* para $B_1 \rightarrow B_3 \rightarrow B_4 \rightarrow B_6$!

What if ifs become iffy?



- `b + c` en B_6 es redundante para el camino $B_1 \to B_2 \to B_4 \to B_6$
- ¡No hay ningún punto ni arista *única* para $B_1 \to B_3 \to B_4 \to B_6$!
- Tenemos que duplicar B_4 y B_6 – ¿Mejora?

No podemos eliminar toda la redundancia

- Ese "caso patólogico" es muy frecuente – cadena de condicionales.
- El número de caminos posibles en un programa es exponencial en la cantidad de condicionales.
- Eliminar la redundancia de *todas* las expresiones requeriría duplicar muchísimo código.

Sólo agregaremos bloques,
nunca los duplicaremos.

UNIVERSIDAD SIMÓN BOLÍVAR

Objetivos de la transformación

- Los programas resultantes de aplicar las mejoras que eliminan parcialmente la redundancia deben ser tales que:
    1. Se hayan eliminado *todas* las operaciones redundantes que sea posible eliminar *sin* duplicar código.
    2. No realicen ningún cómputo que no esté en el programa original.
    3. Las expresiones sean calculadas en el *último* momento posible.

Objetivos de la transformación

- Los programas resultantes de aplicar las mejoras que eliminan parcialmente la redundancia deben ser tales que:
  1. Se hayan eliminado *todas* las operaciones redundantes que sea posible eliminar *sin* duplicar código.
  2. No realicen ningún cómputo que no esté en el programa original.
  3. Las expresiones sean calculadas en el *último* momento posible.
- Esto va a mejorar notablemente la asignación de registros.
  - Asignar a registros los resultados de expresiones redundantes.
  - Calcularlos en el último momento reduce su tiempo de vida – mínima ocupación del registro asignado.

UNIVERSIDAD SIMÓN BOLÍVAR

Objetivos de la transformación

- Los programas resultantes de aplicar las mejoras que eliminan parcialmente la redundancia deben ser tales que:
  1. Se hayan eliminado *todas* las operaciones redundantes que sea posible eliminar *sin* duplicar código.
  2. No realicen ningún cómputo que no esté en el programa original.
  3. Las expresiones sean calculadas en el *último* momento posible.
- Esto va a mejorar notablemente la asignación de registros.
  - Asignar a registros los resultados de expresiones redundantes.
  - Calcularlos en el último momento reduce su tiempo de vida – mínima ocupación del registro asignado.

Mejorar el código con éstos objetivos se conoce como *Lazy Code Motion*

Un caso trivial para motivar el complejo

- ¿Qué quiere decir que e sea completamente redundante?
  - Una expresión e en el bloque B es redundante si para todo camino hasta B, e ha sido evaluada y sus operandos no han sido redefinidos.
  - Sea S el conjunto de bloques que contienen e, tales que hacen que e sea redundante en B.
  - Si removemos las aristas que *salen* de todos los bloques en S, entonces B quedará desconectado del inicio del programa – se denomina *cutset*.

Un caso trivial para motivar el complejo

- ¿Qué quiere decir que e sea completamente redundante?
  - Una expresión e en el bloque B es redundante si para todo camino hasta B, e ha sido evaluada y sus operandos no han sido redefinidos.
  - Sea S el conjunto de bloques que contienen e, tales que hacen que e sea redundante en B.
  - Si removemos las aristas que *salen* de todos los bloques en S, entonces B quedará desconectado del inicio del programa – se denomina *cutset*.
- ¿Qué quiere decir que e sea parcialmente redundante?
  - Sea S el conjunto de bloques que contienen e, tales que hacen que e sea redundante en B.
  - Si removemos las aristas que *salen* de todos los bloques en S, entonces B aún es alcanzable desde el inicio del programa.
  - Si logramos incoporar código adicional con copias de la expresión sobre las aristas restantes, se reduce al problema de redundancia completa.

Una restricción adicional. . .

- No queremos operaciones de más – el código adicional debe incorporarse solamente cuando se **anticipe** la expresión.
- La expresión `b + c` se **anticipa** en un punto p si todos los caminos *a partir* de p calculan `b + c` con los valores de `b` y `c` disponibles en p.
- Sea un camino $B_1 \rightarrow \ldots \rightarrow B_n$ sin ciclos.
  - Supongamos que e se evalúa en B_1 y B_n.
  - Los operandos de e no se redefinen en todo el camino.
  - e **no** es anticipada al entrar en B_i si y sólo si existe una arista desde B_j (con $i \leq j < n$) que lleva a un camino que **no** usa el valor de e.

La anticipación de una expresión
limita cuan temprano incorporarla.

. . . y sobre el mismo camino

- Si e es anticipada o está disponible en la entrada de B_i ya tenemos la arista para el *cutset* – precisamente $B_{i-1} \rightarrow B_i$
- Si e es anticipada pero no está disponible en la entrada de B_i, pondremos una copia de e en la arista $B_{i-1} \rightarrow B_i$ para incorporarla al *cutset*.
- ¡Estamos insertando copias de la expresión tan cerca del uso como sea posible y sin agregar redundancia!
- Mientras más temprano en el flujo se inserte la copia de la expresión, más redundancia podremos eliminar – el método la inserta lo más temprano posible, pero no demasiado temprano.

Algoritmo para *Lazy Code Motion*

Cuatro pasos

1. ¿Dónde incorporar expresiones? – Usar anticipación.

Algoritmo para *Lazy Code Motion*

Cuatro pasos

1. ¿Dónde incorporar expresiones? – Usar anticipación.
2. ¿Cuándo es lo más temprano posible?
   - Calcular *earliest cutset*.
   - Aquel que elimina la mayor cantidad de redundancia – sin copiar código y sin cambiar la semántica.
   - Incorporar las copias en el punto de primera anticipación.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para *Lazy Code Motion*

Cuatro pasos

1. ¿Dónde incorporar expresiones? – Usar anticipación.
2. ¿Cuándo es lo más temprano posible?
   - Calcular *earliest cutset*.
   - Aquel que elimina la mayor cantidad de redundancia – sin copiar código y sin cambiar la semántica.
   - Incorporar las copias en el punto de primera anticipación.
3. ¿Cuánto podemos refinar el *cutset*?
   - Se intenta "bajar" el *cutset* – diferir la evaluación.
   - Pero no tanto que cambie la semántica.

Algoritmo para *Lazy Code Motion*

Cuatro pasos

1. ¿Dónde incorporar expresiones? – Usar anticipación.
2. ¿Cuándo es lo más temprano posible?
   - Calcular *earliest cutset*.
   - Aquel que elimina la mayor cantidad de redundancia – sin copiar código y sin cambiar la semántica.
   - Incorporar las copias en el punto de primera anticipación.
3. ¿Cuánto podemos refinar el *cutset*?
   - Se intenta "bajar" el *cutset* – diferir la evaluación.
   - Pero no tanto que cambie la semántica.
4. ¿Dejamos asignaciones a temporales de un solo uso?

Algoritmo para *Lazy Code Motion*

Cuatro pasos

1. ¿Dónde incorporar expresiones? – Usar anticipación.
2. ¿Cuándo es lo más temprano posible?
   - Calcular *earliest cutset*.
   - Aquel que elimina la mayor cantidad de redundancia – sin copiar código y sin cambiar la semántica.
   - Incorporar las copias en el punto de primera anticipación.
3. ¿Cuánto podemos refinar el *cutset*?
   - Se intenta "bajar" el *cutset* – diferir la evaluación.
   - Pero no tanto que cambie la semántica.
4. ¿Dejamos asignaciones a temporales de un solo uso?

1 y 4 son *backward flow*,
2 y 3 son *forward flow*.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para *Lazy Code Motion*

Pre-procesamiento

- Supondremos que cada instrucción está en su propio bloque básico.
- Sólo se agregan cálculos de expresiones al inicio de los bloques – usando nuevos bloques si el bloque tiene más de un predecesor.
- Para cada bloque B definiremos los conjuntos
  - e_use_B – expresiones calculadas en B.
  - e_kill_B – expresiones muertas (operandos redefinidos) en B.

Algoritmo para *Lazy Code Motion*

Expresiones Anticipadas

- Una expresión e es anticipada al entrar si es usada en B.
- Una expresión e es anticipada al entrar a B sólo si
  - e es anticipada al salir de B
  - e no es "matada" (sus operandos no son redefinidos) en B.
- Ninguna expresión es anticipada al terminar el programa.
- En todos los puntos interiores, inicialmente *todas* las expresiones son anticipadas – refinamos por intersección.

Algoritmo para *Lazy Code Motion*

Expresiones Anticipadas

- Dominio – conjuntos de expresiones.
- Dirección – *backward flow*.
- Función de Transferencia

$$f_B(x) = e_use_B \cup (x - e_kill_B)$$

- Base – $IN[EXIT] = \emptyset$
- Inicialización – $IN[B] = U$ (todas las expresiones)
- Ecuaciones

$$IN[B] = f_B(OUT[B])$$
$$OUT[B] = \bigcap_{S \text{ sucesor de } B} IN[S]$$

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para *Lazy Code Motion*

Expresiones Disponibles

- Una expresión está **disponible** en el punto p si es anticipada en todos los caminos que llegan a p – usamos los resultados de la fase anterior para calcular en esta.
- Una expresión e está disponible al salir de B si
  - Está disponible al entrar, o
  - Es anticipada al entrar – podríamos calcularla aquí.
- Una expresión e está disponible al salir si no es "matada" en B.
- Combinamos los resultados de las fases 1 y 2 para determinar las expresiones cuya aparición más temprana es B

$$earliest[B] = anticipated[B].in - available[B].in$$

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para *Lazy Code Motion*

Expresiones Disponibles

- Dominio – conjuntos de expresiones.
- Dirección – *forward flow*.
- Función de Transferencia

$$f_B(x) = (anticipated[B].in \cup x) - e_kill_B$$

- Base – $OUT[ENTRY] = \emptyset$
- Inicialización – $OUT[B] = U$ (todas las expresiones)
- Ecuaciones

$$OUT[B] = f_B(IN[B])$$
$$IN[B] = \bigcap_{P \text{ predecesor de } B} OUT[P]$$

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para *Lazy Code Motion*

Expresiones Diferibles

- Una expresión es **diferible** hasta el punto p si aparece en *todos* los caminos desde el inicio hasta p y no hay ningún uso de la expresión entre su última aparición y p.
- Una expresión e se puede diferir a la salida si no se usa en B.
- Una expresión e se puede diferir a la salida de B si
  - Se puede diferir a la entrada de B, o
  - B es el punto más temprano de evaluación ($earliest[B]$).
- En los puntos interiores, una expresión puede diferirse a la entrada de B sólo si todos sus precedesores la difirieron a la salida – refinamos con la intersección.

Algoritmo para *Lazy Code Motion*

Expresiones Diferibles

- La intención es encontrar el punto en el cual la expresión pasa de ser diferible a ser necesaria inmediatamente.
- Es obligatorio evaluar e al principio de un bloque si
  - Está en el punto más temprano de evaluación ($earliest[B]$) o llega diferida a B, y
  - Se usa en B o hay algún sucesor al cual no puede diferirse.
- Combinamos los resultados de las fases 2 y 3 para determinar cuales expresiones *deben* evaluarse al iniciar cada B

$$latest[B] = (earliest[B] \cup postponable[B]) \cap \\ (e_use_B \cup \neg(\bigcap_{S,succ(B)} (earliest[S] \cup postponable[S])))$$

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para *Lazy Code Motion*

Expresiones Diferibles

- Dominio – conjuntos de expresiones.
- Dirección – *forward flow*.
- Función de Transferencia

$$f_B(x) = (earliest[B] \cup x) - e_use_B$$

- Base – $OUT[ENTRY] = \emptyset$
- Inicialización – $OUT[B] = U$ (todas las expresiones)
- Ecuaciones

$$OUT[B] = f_B(IN[B])$$
$$IN[B] = \bigcap_{P \text{ predecesor de } B} OUT[P]$$

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para *Lazy Code Motion*

Expresiones Usadas

- El algoritmo concreto utiliza variables temporales para contener las expresiones calculadas – es posible generar asignaciones que luego no son utilizadas.
- Equivalente al análisis de variables vivas – en contexto de expresiones.
- Una expresión usada al salir de un bloque B se mantiene usada al entrar si no fue calculada en B.
- Una expresión usada por B, se indica como usada al entrar.
- En los puntos interiores, nos interesa determinar si una expresión es usada en cualquiera de los sucesores – refinamos con la unión.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para *Lazy Code Motion*

Expresiones Usadas

- Dominio – conjuntos de expresiones.
- Dirección – *backward flow*.
- Función de Transferencia

$$f_B(x) = (e_use_B \cup x) - latest[B]$$

- Base – $IN[EXIT] = \emptyset$
- Inicialización – $IN[B] = \emptyset$
- Ecuaciones

$$IN[B] = f_B(OUT[B])$$
$$OUT[B] = \bigcup_{S \text{ sucesor de } B} IN[S]$$

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para *Lazy Code Motion*

El algoritmo concreto – los cálculos en orden

1. El Algoritmo trabaja sobre un grafo de flujo sobre el cual se ha calculado e_use_B y e_kill_B para cada bloque B.
2. Se insertan bloques *vacíos* en todas las aristas que ingresan a bloques con más de un predecesor.
3. Calcular $anticipated[B]$ para todos los bloques.
4. Calcular $available[B]$ para todos los bloques.
5. Calcular $earliest[B]$ para todos los bloques.
6. Calcular $postponable[B]$ para todos los bloques.
7. Calcular $latest[B]$ para todos los bloques.
8. Calcular $used[B]$ para todos los bloques.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para *Lazy Code Motion*

El algoritmo concreto – transformar código

9. Para cada expresión `x + y` del programa
   - **(a)** Definir una nueva variable temporal `t`.
   - **(b)** $\forall B$ tal que $x + y \in \mathit{latest}[B] \cap \mathit{used}[B]$, agregar `t := x + y` al comienzo de B.
   - **(c)** $\forall B$ tal que $x + y \in \mathit{e_use}_B \cap (\neg \mathit{latest}[B] \cup \mathit{used}[B])$, reemplazar `x + y` por `t`.

¿Por qué funciona?

- Las restricciones de ubicación vienen dadas por la anticipación de expresiones – para cada punto determina si la expresión se usa para todos los caminos salientes.
- La ubicación más temprana viene dada por la combinación de anticipar una expresión pero no tenerla disponible – para cada punto determina si la expresión fue anticipada en todos los caminos entrantes.
- La ubicación más posterior viene dada por las ubicación en las cuales ya no se puede posponer el cálculo – para cada punto determina si la expresión va a ser usada justo después de haber sido pospuesta en todos los caminos entrantes.
- Las asignaciones temporal persisten solamente si son empleadas por *algún* camino a partir de su aparición.

Bibliografía

- [*Aho*]
  - Sección 9.5
  - Ejercicios 9.5.1 a 9.5.3

UNIVERSIDAD SIMÓN BOLÍVAR