Evaluación de Aprendizaje 3

Unidades Temáticas 11 a 13

Alumno: Facundo Mediotte

DNI: 39436162

Ejercicio II

Se sabe que la fórmula de acceso que el compilador agrega al código ejecutable para acceder a un array de dos dimensiones ordenado por filas es la siguiente:

*z(i,j):=dir[v(Fi,Ci)] + [((i - Fi) * (Cn - Ci + 1)) + (j - Ci)]*tamaño del componente (tipo)*

Suponga que el compilador solo soporta limites fijos para sus vectores. Es decir, el límite inferior y superior de las filas (Fi y Fn) y de las columnas (Ci y Cn) son conocidos en tiempo de compilación.

Por ejemplo: para acceder al componente z(i,j) del vector int z(10..18,20..30)

z (i,j) = dir **[** z (10,20) **]** + **[** ((i–10) * (30-20+1)) + (j-20) **]** * 2 bytes

También se sabe que es posible hacer una optimización por reducción simple en esta fórmula al generar polaca inversa. Indique dónde se podría hacer. Explique y ejemplifique como lo haría.

Respuesta II

Se puede optimizar el código por medio de reducción simple en un vector, en la sección de las operaciones aritméticas de constantes pudiendo reducir por constant folding en cada una de las porciones de acceso a ese vector.

z (i,j) = dir **[** z (10,20) **]** + **[** ((i–10) * (30-20+1)) + (j-20) **]** * 2 bytes

transformarla en: z (i,j) = dir **[** z (10,20) **]** + **[** ((i–10) * (11)) + (j-20) **]** * 2 bytes

Por ejemplo, en la zona marcada se puede aplicar una reducción, reemplazando dicha operación aritmética por el resultado de la misma. En el caso de que i y j tuviesen valores constantes se podría reducir mucho más la expresión.

Esta optimización se puede realizar también con polaca inversa a la entrada de la generación de código intermedio, es decir en la acción semántica antes de escribir el código intermedio.

También podemos hacerlo en la representación intermedia reescribiendo la misma, o a la salida de la generación de código intermedio y se optimiza en la salida es decir en la escritura del código assembler, que se escribe directamente optimizado.

Al generar la polaca inversa:

- Si lo que se escribe en la notación es un operador, se toma los dos operandos anteriores y si son constantes, se resolverá la operación y se reemplaza la operación con el resultado en la polaca.

Ejemplo:

z(i,j) = dir [z (10,20)] + [((i-10) * (30-20+1)) + (j-20)] * 2 bytes
dir [z(10,20)] -> cte_z10_20

cte_{z10_20} i 10 – 30 20 – 1 + * j 20 - + 2 * + z =
cte_{z10_20} i 10 – 11 * j 20 - + 2 * + z =

Con valores en i y j se podría reducir aún más:

z(11,22) = dir [z (10,20)] + [((11-10) * (30-20+1)) + (22-20)] * 2 bytes

dir [z(10,20)] -> cte_z10_20

cte_{z10_20} 11 10 – ⟶ cte_{z10_20} 1
cte_{z10_20} 1 30 20 – ⟶ cte_{z10_20} 1 10
cte_{z10_20} 1 10 1 + ⟶ cte_{z10_20} 1 11
cte_{z10_20} 1 11 * ⟶ cte_{z10_20} 11
cte_{z10_20} 11 22 20 - ⟶ cte_{z10_20} 11 2
cte_{z10_20}11 2 + ⟶ cte_{z10_20} 13
cte_{z10_20}13 2 * ⟶ cte_{z10_20} 26
cte_{z10_20} 26 + z_{11_22} = ⟶ así quedaría generada la polaca inversa si aplicamos optimización de código al generarla. (se podría reducir también la expresión cte_{z10_20} 26 + si supiésemos el resultado de cte_{z10_20}) luego en la generación de código assembler se realizarán menos instrucciones de ADD y MUL ya que las expresiones aritméticas se resolvieron en la generación del código intermedio.

En representación intermedia:

- Se recorrer la polaca inversa ya construida:
  - Si el elemento es un operando se apila
  - Si el elemento es un operador se desapila los operandos anteriores y si son constantes, resuelve la operación aritmética y apila. Se sustituye en la polaca inversa por el resultado.
- Se produce una baja lógica de la celda, debido a que si se pierde la referencia de estas no funcionarían correctamente los saltos condicionales.

La polaca se transformaría de la siguiente forma:

cte_{z10_20} 11 10 – 30 20 – 1 + * 22 20 - + 2 * + z_{11_22} =
cte_{z10_20} 26 x x x x x x x x x x x x x + z_{11_22} =

A la salida:

- Se recorre la polaca inversa construida para la traducción a assembler, el proceso es similar al de optimización en representación intermedia, pero se realiza antes de la traducción al assembler.

A la traducción del assembler quedaría algo asi:

MOV R1 cte_{z10_20}
ADD R1, 26
MOV z_{11_12}**, R1**

Ejercicio III

Suponga un lenguaje que tiene las siguientes reglas de promoción numérica para sus tipos de datos.

int → long int → double

float → double

Suponga también que en una versión del compilador se generó el siguiente conjunto de tercetos para la sentencia:

25 (*, b, c)
26 (*, [25], d)
27 (*, 2.0 , a)
28 (+, [26], [27])
29 (=, w, [28])

w= b*c*d+ (2.0*a) con: int d; double w; long a,c; float b

En otra versión del compilador se pide que el mismo genere los tercetos con conversiones. Escribir como quedaría el conjunto de tercetos de la sentencia anterior con conversiones de tipo según la promoción numérica establecida.

Respuesta III

Asumo que 2.0 es Float

25 (CFD, b, _)	**Double**
26 (CLD, c, _)	**Double**
27 (*, [25], [26])	**Double**
28 (CIL, d, _)	**Long**
29 (CLD, [28], _)	**Double**
30 (*, [27], [29])	**Double**
31 (CFD, 2.0, _)	**Double**
32 (CLD, a, _)	**Double**
33 (*, [31], [32])	**Double**
34 (+, [30], [33])	**Double**
35 (=, w, [34])	**Double**

Tabla de conversiones ampliada

f	Int			Long			Float			Double	
Int	**I**			**L**	**CIL**(*)		**F**	CIL CLD	CFD	**D**	CIL CLD
Long	**L**		CIL	**L**			**D**	CLD	CFD	**D**	CLD
Float	**D**	CFD	CIL CLD	**D**	CFD	CLD	**F**			**D**	CFD
Double	**D**		CIL CLD	**D**		CLD	**D**		CFD	**D**	