Ejemplos de Transformación de un Programa Dgcl a ARC

B. Gonzalez Kriegel - N. Wolovick

9 de mayo de 2000

1 Algoritmo de la División Entera

Queremos transformar el código escrito en Dgcl - el lenguaje de los comandos guardados de Dijkstra - obtenido en la página 134 del apunte "Cálculo de Programas", que implementa el algoritmo de la división entera, en un código ensamlador para la ISA de ARC.

```
[ var x,y,q,r:  Int
  {R : x ≥ 0 ∧ y ≥ 0}
  q,r := 0,x
  {Q : x = q × y + r ∧ 0 ≤ r}
  do
       r≥y → q,r := q+1,r-y
  od
  {Q : x = q × y + r ∧ 0 ≤ r ∧ r < y}
]
```

La primera línea de código declara 4 variables de tipo entero.

```
[ var x,y,q,r:  Int
```

Utilizamos 4 registros para almacenar el valor de cada variable. También indicamos al ensamblador el inicio de nuestro código, asi como la dirección de memoria que elejimos para que comience a ensamblar.

```
!  x <-> %r1
!  y <-> %r2
!  q <-> %r3
!  r <-> %r4
.begin
.org  2048
```

La siguiente sentencia es una multiasignación que hace verdadero el invariante en el inicio.

```
  q,r := 0,x
  {Q : x = q × y + r ∧ 0 ≤ r}
```

La multiasignación puede ser desacoplada sin problemas, pues los *rhs* no dependen de los *lhs*. Para hacer 0 a q y copiar x en r, podemos usar `sethi` y `add` de la siguiente forma: como `sethi` copia su operando inmediato de 22 bits en la parte superior del registro destino y pone a 0 los 10 bits bajos, con asignar 0 a la parte alta alcanza para asignar 0 a `%r3` (q)[1]. Para la segunda asignación efectuamos una suma entre `%r1` (x) y `%r0` y ponemos el resultado en `%r4` (r).

```
sethi 0, %r3          !  q := 0
add   %r1, %r0, %r4   !  r := x
```

[1] A esto mismo lo podemos lograr de muchas formas distintas, utilizando `%r0` y las instrucciones aritmético-lógicas.

El inicio de la repetición nos induce a introducir una etiqueta `loop`, para poder saltar aquí cuando el lazo deba repetirse.

do
    r≥y $\rightarrow \cdots$

La comparación de la guarda la obtenemos con un `subcc` que modifica el registro de estado y descarta el resultado. En el caso que la guarda no sea verdadera saltamos a `endloop`, etiqueta de la primera instrucción fuera del ciclo.

```
loop:   subcc %r4, %r2, %r0
        bneg  endloop         !  do r>=y
```

La multiasignación que le sigue a la guarda del **do...od**

$\cdots \rightarrow$ q,r := q+1,r-y

se logra fácilmente con instrucciones de suma y resta.

```
        add   %r3, 1, %r3     !  q := q+1
        sub   %r4, %r2, %r4   !  r := r-y
```

El final de la repetición la logramos con un salto incondicional al inicio del lazo, marcado por la etiqueta `loop`

```
        ba    loop            !  od
```

Finalmente ponemos la etiqueta de salida del lazo junto con la instrucción `halt` que detiene la ejecución de la máquina, e indicamos al ensamblador que terminamos el programa.

```
endloop:  halt
          .end
```

El código ARC completo se muestra a continuación, con el agregado de un par de instrucciones que sirven para cargar `%r1` y `%r2` a fin de probar el programa.

```
          !  x <-> %r1
          !  y <-> %r2
          !  q <-> %r3
          !  r <-> %r4
          .begin
          .org  2048
          ld    [x], %r1        !  cargo x
          ld    [y], %r2        !  cargo y
          sethi 0, %r3          !  q := 0
          add   %r1, %r0, %r4   !  r := x
loop:     subcc %r4, %r2, %r0
          bneg  endloop         !  do r>= y
          add   %r3, 1, %r3     !  q := q+1
          sub   %r4, %r2, %r4   !  r := r-y
          ba    loop            !  od
endloop:  halt
x:        25
y:        6
          .end
```

2 Suma de los Elementos de un Arreglo

En la página 142 del mismo apunte se deriva, a partir de la especificación Q, un algoritmo para obtener en la variable r la suma de los elementos de un arreglo A.

```
[ con N: Int, A: array [0,N) of Int; var n,r:  Int
  {R : N ≥ 0}
  n,r := 0,0
  {P : r = (∑.i : 0 ≤ i < n : A.i) ∧ 0 ≤ n ≤ N}
  do
       n≠N → n,r := n+1,r+A.n
  od
  {Q : r = (∑.i : 0 ≤ i < n : A.i)}
]
```

Vamos a obtener paso a paso el programa ARC a partir del código anterior. La primera línea contiene las declaraciones.

```
[ con N: Int, A: array [0,N) of Int; var n,r:  Int
```

El primer problema se presenta en como representar el tipo de datos `array` convenientemente. La implementación de bajo nivel más inmediata y también la más usada, es pensar un pedazo de la memoria como un arreglo de enteros, donde en `M[A]` tenemos el elemento cero, en `M[A+4]` el uno y asi (recordar que cada entero ocupa 4 bytes).

La constante `N` la podemos mapear a un símbolo definido con el pseudo-op `.equ`. Para las variables enteras n y r elegimos mantenerlas en registros del microprocesador.

```
          !  n <-> %r1
          !  r <-> %r2
          .begin
N         .equ  5              !  el arreglo tendra 5 elementos
          .org  2048
```

El arreglo no lo definimos aún, pues queremos que esté al final del programa.

La multiasignación siguiente, puede ser desacoplada en dos de manera directa.

```
  n,r := 0,0
```

Existen muchas formas de poner en 0 los registros `%r1` y `%r2`. Elegimos una de cada tipo.

```
          and   %r0, %r0, %r1   !  n := 0
          sethi 0, %r2          !  r := 0
```

El inicio del constructor de repetición lo mapeamos de la misma forma que en la sección anterior.

```
  do
       n≠N → ···
```

Con una resta entre n y N que modifique el `%psr` y un salto que salga del lazo en caso de igualdad nos alcanza. Seguimos la estructura `loop-endloop` de las etiquetas que marcan el inicio y fin del bucle.

```
loop:     subcc %r1, N, %r0
          be    endloop         !  do n!=N
```

La multiasignación siguiente si requiere cuidado en su desacople ya que un orden (n, x) nos da un resultado que no es equivalente a la asignación múltiple.

```
       ··· → n,r := n+1,r+A.n
```

El incremento de n es inmediato, no así la suma en r, pues tenemos que obtener por un lado el índice corregido a bytes en la memoria, y por el otro el valor de esa memoria para recién poder sumarlo con `%r2` (r).

```
          sll   %r1, 2, %r3     !  %r3 := %r1*4
          ld    A+%r3, %r4      !  %r4 := A.%r1
          add   %r2, %r4, %r2   !  r := r+A.n
          add   %r1, 1, %r1     !  n := n+1
```

El final del lazo se obtiene con un salto al inicio del mismo.

```
          ba    loop
```

El final del programa ARC está dado por la etiqueta `endloop` de finalización de lazo junto a la instrucción de parada del programa, la definición del arreglo `A` y la pseudo-op que indica el final del ensamblado.

```
endloop:  halt
A:        5
          -2
          3
          -5
          7
          .end
```

Veamos el código completo:

```
          !  n <-> %r1
          !  r <-> %r2
          .begin
N         .equ  5               !  el arreglo tendra 5 elementos
          .org  2048
          and   %r0, %r0, %r1   !  n := 0
          sethi 0, %r2          !  r := 0
loop:     subcc %r1, N, %r0
          be    endloop         !  do n!=N
          sll   %r1, 2, %r3     !  %r3 := %r1*4
          ld    A+%r3, %r4      !  %r4 := A.%r1
          add   %r2, %r4, %r2   !  r := r+A.n
          add   %r1, 1, %r1     !  n := n+1
          ba    loop
endloop:  halt
A:        5
          -2
          3
          -5
          7
          .end
```