Universidad Nacional de Córdoba

Facultad de Ciencias Exactas, Físicas y Naturales

Código Intermedio

Trabajo integrador final

Practica y construccion de compiladores

Autores:

Julián González

2021

Resumen

Este documento explicara los codigos intermedios usados por los compiladores GCC y CLANG/LLVM, como asi comparar sus principales caracteristicas viendo ventajas y desventajas entre ellos.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introduccion

El codigo intermedio es un codigo interno usado por el compilador para representar el codigo fuente. El codigo intermedio esta diseñado para llevar a cabo el procesamiento del codigo fuente, como es la optimizacion y la traduccion a codigo maquina.

Una de las caracteristicas mas esenciales del codigo intermedio es ser independiente del *hardware*. Por lo tanto, permite la portabilidad entre distintos sistemas.

Otra propiedad importante de todo codigo intermedio es su facil generacion a partir del codigo fuente, como asi tambien su facil traduccion al codigo maquina para la arquitectura deseada.

No existe un unico codigo intermedio, sino que hay distintos tipos y categorias, variando de compilador en compilador. Aunque un mismo compilador puede usar varios tipos de codigo intermedio en el proceso.

A continuacion, se presentan los codigos intermedios utilizados por los compiladores GCC y CLANG/LLVM, especificando las caracteristicas de cada uno y comparando sus prestaciones posteriormente.

Capítulo 2

Codigo intermedio de GCC

A continuacion se exponen las distintos codigos intermedios que GCC utiliza en la compilacion. Los distintos codigos intermedios estan relacionados en la forma que la salida de cada uno es la entrada del siguiente, avanzando desde una representacion general de alto nivel hacia una especifica de bajo nivel.

I Analizador lexico y sintactico

El analizador lexico lee la secuencia de caracteres desde la salida del preprocesador y agrupa los caracteres en secuencias llamadas lexemas. Por cada lexema, el analizador genera una token con la forma:

[*token name, attribute value*]

Donde *token name* son símbolos abstractos usados durante el análisis sintáctico, y *attribute value* es un puntero a una entrada en la tablas de símbolos. GCC no permite obtener los tokens válidos en forma de texto. Es un archivo interno. El analizador sintactico chequea si la gramática del lenguaje acepta la secuencia de tokens generados por el analizador lexico, sino reporta errores de sintaxis. Ademas, el analizador sintactico usa el primer componente de cada token para crear una representación en forma de árbol que muestre la estructura de los tokens, es decir, un árbol sintáctico. Con el *flag* -fdump-tree-original-raw se obtiene la representacion textual del arbol abstracto sintactico.

```
1    $ gcc -fdump-tree-original-raw codigo-ejemplo.c -o codigo-ejemplo
```

Listing 2.1: Comando de compilación del archivo codigo-ejemplo.c [1] para GCC.

II Analizador semantico

El analizador semantico utiliza el árbol sintáctico y la información de la tabla de símbolos para revisar la consistencia semántica del programa fuente con respecto a la definición del lenguaje.

III *Generic*

Generic es un codigo intermedio independiente del lenguaje con estructura de arbol que es generado por el *front end*. *Generic* es capaz de representar todos los lenguajes admitidos por GCC. *Generic* se produce eliminando construcciones especificas del lenguaje del arbol de parseo. GCC pasa del arbol abstracto sintactico a la representacion en *Gimple* en lenguajes como C.

IV *Gimple*

Gimple es un codigo intermedio de tres direcciones resultante de desglosar *Generic* en tuplas de no mas de tres operandos, a traves de la herramienta interna de GCC llamada *Gimplifier*. *Gimple* introduce variables temporales para poder computar expresiones complejas y permite supervisar el flujo de control a nivel inferior con sentencia secuenciales y saltos incondicionales. *Gimple* es el codigo intermedio principal de GCC (los lenguajes C y C++ se convierten a *Gimple* sin pasar por *Generic*), ademas de ser conveniente para optimizar.

Existen tres tipos de *Gimple*:

- *Gimple* de alto nivel que es lo que se obtiene despues de desglosar el *Generic*.
- *Gimple* de bajo nivel que se obtiene al linealizar todas las estructuras de flujo de control de del *Gimple* de alto nivel, incluidas las funciones anidadas, el manejo de excepciones y los bucles.
- *Gimple* SSA es el *Gimple* de bajo nivel reescrito en la forma SSA.

Con el *flag* -fdump-tree-gimple se obtiene la representacion en la forma de *Gimple*.

```
$ gcc -fdump-tree-gimple codigo-ejemplo.c -o codigo-ejemplo
```

Listing 2.2: Comando de compilación del archivo codigo-ejemplo.c [1] para GCC.

Ademas, con el *flag* -fdump-tree-all-graph GCC genera muchos archivos con la extension .cfg los cuales pueden visualizarse con una herramienta online Graphviz. Esta herramienta permite ver la evolucion del codigo en las distintas pasadas de una manera mucho mas conveniente para el usuario.

```
$ gcc -fdump-tree-all-graph codigo-ejemplo.c -o codigo-ejemplo
```

Listing 2.3: Comando de compilación del archivo codigo-ejemplo.c [1] para GCC.

V *Register Transfer Language*

Register Transfer Language es un codigo intermedio de bajo nivel semejante al lenguaje ensamblador. La mayor parte del trabajo del compilador se realiza en *Register Transfer Language*. Tiene una forma interna, formada por estructuras que apuntan a otras estructuras, y una forma textual que se utiliza en la descripción de la máquina y en los volcados de depuración impresos. El formulario textual usa paréntesis anidados para indicar los punteros en el formulario interno. Con el *flag* -fdump-rtl-final se obtiene la representacion en la forma de *Register Transfer Language* ya optimizado por el compilador.

```
$ gcc -fdump-rtl-final codigo-ejemplo.c -o codigo-ejemplo
```

Listing 2.4: Comando de compilación del archivo codigo-ejemplo.c [1] para GCC.

VI *Assembler*

Por ultimo, es posible obtener la salida en *Assembler* con el *flag* -S.

```
$ gcc -S codigo-ejemplo.c -o codigo-ejemplo.s
```

Listing 2.5: Comando de compilación del archivo codigo-ejemplo.c [1] para GCC.

Capítulo 3

Codigo intermedio de CLANG/LLVM

Clang es el *frontend* de LLVM para la familia de los lenguajes de C. Clang consiste en un preprocesador C, un analizador léxico, un analizador sintáctico, un analizador semántico y un generador IR. El Optimizador analiza la IR y la traduce a una forma más eficiente. *opt* es la herramienta de optimización de LLVM. El *Backend* genera código máquina al mapear la IR al conjunto de instrucciones del hardware de destino. La herramienta backend de LLVM es llc. Genera código máquina a partir del IR de LLVM en tres fases: selección de instrucciones, asignación de registros y programación de la instrucción.

I Analizador lexico y sintactico

El analizador lexico lee la salida del preprocesador y realiza su tokenización. El analizador sintactico chequea si la gramática del lenguaje acepta la secuencia de tokens generados por el analizador lexico y reporta errores de sintáxis. El analizador sintactico produce el *error recovery*, si es posible. La tokenización se organiza de la siguiente manera:

- La definición del tipo de token en cuestion.
- El *token*.
- *Flag* que describe un adicional de el *token*.
- La ubicacion del token en el codigo fuente.

Después de analizar la gramática de la secuencia de *tokens*, emite un árbol abstracto sintactico. Los nodos de un árbol abstracto sintactico representan declaraciones, sentencias y tipos.

```
1    $ clang -fsyntax-only -Xclang -dump-tokens codigo-ejemplo.c  2>&1 | tee tokens
```

Listing 3.1: Comando de compilación del archivo codigo-ejemplo.c [1] para CLANG/LLVM.

```
1    $ clang -fsyntax-only -Xclang -ast-dump codigo-ejemplo.c -fno-color-diagnostics > ast
```

Listing 3.2: Comando de compilación del archivo codigo-ejemplo.c [1] para CLANG/LLVM.

II Analizador semantico

El analizador semántico recorre el árbol abstracto sintactico, determinando si las sentencias del código tienen un significado válido. Esta fase comprueba los errores de tipo.

III LLVM IR

El generador IR traduce el árbol abstracto sintactico a codigo intermedio.

```
$ clang -S -emit-llvm codigo-ejemplo.c -o llvm-ir.ll
```

Listing 3.3: Comando de compilación del archivo codigo-ejemplo.c [1] para CLANG/LLVM.

En esta fase tambien trabaja el optimizador que mejora la eficiencia del código basándose en su comprensión del comportamiento en tiempo de ejecución del programa.

```
$ opt -O2 -S llvm-ir.ll -o llvm-ir-optimized.ll
```

Listing 3.4: Comando de compilación del archivo codigo-ejemplo.c [1] para CLANG/LLVM.

IV Selección de instrucciones

La seleccion de instrucciones es el mapeo de instrucciones IR al conjunto de instrucciones de la máquina de destino. Este paso utiliza un espacio de nombres infinito de registros virtuales.

V Asignación de registros

La asignacion de registros es el mapeo de registros virtuales a registros reales en la arquitectura de destino.

VI Programación de la instrucción

La programación de la instrucción es la reordenación de operaciones para reflejar las restricciones de funcionamiento de la máquina de destino.

```
$ llc llvm-ir.ll -print-machineinstrs 2>&1 tee machineinstrs
```

Listing 3.5: Comando de compilación del archivo codigo-ejemplo.c [1] para CLANG/LLVM.

VII *Assembler*

Por ultimo, es posible obtener la salida en lenguaje *assembler* del codigo fuente.

```
$ llc llvm-ir.ll -o assembly.s
```

Listing 3.6: Comando de compilación del archivo codigo-ejemplo.c [1] para CLANG/LLVM.

Capítulo 4

Comparacion entre GCC y CLANG/LLVM

A partir de un mismo código fuente en lenguaje C, se procederá a la compilación empleando tanto GCC como CLANG/LLVM. Luego, se llevará a cabo una comparación y breve descripción de lo observado. Se presentan los códigos relevantes que coinciden con las etapas nombradas anteriormente.

I Analizador lexico y sintactico

GCC no permite obtener la salida del analizador lexico, por lo que se analizara el arbol abstracto sintactico.

GCC describe el arbol relacionando sus nodos a partir de punteros con la concatenando "@z un numero. Entonces, es posible armar el arbol siguiendo estos punteros. En este fragmento se ve como la declaracion de la variable .ª", que es del tipo entera con un valor inicial de "20", es traducida al arbol.

```
1    int a = 20;
```

Listing 4.1: Fragmento del codigo fuente del archivo codigo-ejemplo.c [1] para GCC.

Se observa en @5 la palabra clave *var_decl*, la cual indica la declaracion de una variable. En este caso, el campo *name* apunta a @9, que es un nodo identificador con el nombre de la variable .ª". Tambien, se puede obtener el tipo de la variable refiriendose a @10, que muestra *integer_type*, y lo mismo para *init* se refiere al valor inicial apuntado por @12 que muestra que es el valor 20.

```
1    @5      var_decl         name: @9       type: @10      scpe: @11   srcp: codigo-ejemplo.c:5
     ↪       init: @12
2    @9      identifier_node  strg: a        lngt: 1
3    @10     integer_type     name: @24      size: @13      algn: 32      prec: 32       sign:
     ↪ signed   min : @25     max : @26
4    @11     function_decl    name: @27      type: @28      srcp: codigo-ejemplo.c:3
5    @12     integer_cst      type: @10      int: 20
```

Listing 4.2: Fragmento del arbol de GCC del archivo codigo-ejemplo.c.005t.original [1] para GCC.

El arbol construido por CLANG/LLVM se muestra de una forma mas grafica con barras e identacion. Se observa una *FunctionDecl*, la cual hace referencia al *main()* y adentro de esta una *DeclStmt*, que se encuentra en la linea 5 del codigo fuente. El nodo *VarDecl* define la existencia una variable llamada a de tipo *int*, con un valor de 20.

```
1    |-FunctionDecl 0x207e190 <line:3:1, line:17:1> line:3:5 main 'int ()'
2    | `-CompoundStmt 0x207ea60 <col:12, line:17:1>
3    |   |-DeclStmt 0x207e2d0 <line:5:5, col:15>
4    |   | `-VarDecl 0x207e248 <col:5, col:13> col:9 used a 'int' cinit
5    |   |   `-IntegerLiteral 0x207e2b0 <col:13> 'int' 20
```

Listing 4.3: Fragmento del arbol de CLANG/LLVM del archivo ast [1] para GCC.

II Gimple vs LLVM IR

En esta etapa se comparara como el bucle del codigo fuente se desarrolla en el codigo intermedio de ambos compiladores.

```
    for (int i = 0; i < c; i++)
    {
        arr[i] = suma(a, b);
    }
```

Listing 4.4: Fragmento del codigo fuente del archivo codigo-ejemplo.c [1] para GCC.

GCC emplea *Gimple* para esta etapa que tiene una familiaridad con el lenguaje C, en el cual se ve que se crean etiquetas para realizar el control del salto y el bucle se transforma en sentencias *if else*. Tambien, se crean variables temporales para alojar resultados, como por ejemplo "_1"para el resultado que devuelve la funcion suma.

```
    {
        int i;
        i = 0;
        goto <D.1955>;
        <D.1954>:
        _1 = suma (a, b);
        arr[i] = _1;
        i = i + 1;
        <D.1955>:
        if (i < c) goto <D.1954>; else goto <D.1952>;
        <D.1952>:
    }
```

Listing 4.5: Fragmento del *Gimple* de GCC del archivo codigo-ejemplo.c.006t.gimple [1] para GCC.

Por otro lado se tiene LLVM IR, el cual ya tiene un parecido con el lenguaje *assembler* mas que con el lenguaje C. Se utilizan variables temporales para los resultados de las instrucciones sin repetir concatenando " %z un numero. La etiqueta 10 hace referencia al control del salto que en la ultima instruccion decide si saltar a la etiqueta 14 o 24. En la etiqueta 14 se cargan las variables %2 y %3 que hacen referencia a las variables a y b para realizar la llamada a la funcino suma. Después, se guarda el resultado en el *array*.

```
10:                                               ; preds = %21, %0
  %11 = load i32, i32* %6, align 4
  %12 = load i32, i32* %4, align 4
  %13 = icmp slt i32 %11, %12
  br i1 %13, label %14, label %24

14:                                               ; preds = %10
  %15 = load i32, i32* %2, align 4
  %16 = load i32, i32* %3, align 4
  %17 = call i32 @suma(i32 %15, i32 %16)
  %18 = load i32, i32* %6, align 4
  %19 = sext i32 %18 to i64
  %20 = getelementptr inbounds [10 x i32], [10 x i32]* %5, i64 0, i64 %19
  store i32 %17, i32* %20, align 4
  br label %21

21:                                               ; preds = %14
  %22 = load i32, i32* %6, align 4
  %23 = add nsw i32 %22, 1
  store i32 %23, i32* %6, align 4
  br label %10
```

Listing 4.6: Fragmento del LLVM IR de CLANG/LLVM del archivo llvm-ir.ll [1] para GCC.

III RTL vs MachineInstr

En esta etapa ambos codigos son poco legibles para humanos por lo que su descripción no es sencilla. Se muestra como la declaracion de las variables a y b son expresadas por cada compilador.

```
    int a = 20;
    int b = 30;
```

Listing 4.7: Fragmento del codigo fuente del archivo codigo-ejemplo.c [1] para GCC.

```
(insn 6 3 7 2 (set (mem/c:SI (plus:DI (reg/f:DI 6 bp)
                (const_int -60 [0xffffffffffffffc4])) [2 a+0 S4 A32])
        (const_int 20 [0x14])) "codigo-ejemplo.c":5:9 75 {*movsi_internal}
     (nil))
(insn 7 6 8 2 (set (mem/c:SI (plus:DI (reg/f:DI 6 bp)
                (const_int -56 [0xffffffffffffffc8])) [2 b+0 S4 A64])
        (const_int 30 [0x1e])) "codigo-ejemplo.c":6:9 75 {*movsi_internal}
     (nil))
```

Listing 4.8: Fragmento de RTL de GCC del archivo codigo-ejemplo.c.330r.final [1] para GCC.

```
    MOV32mi %stack.1, 1, $noreg, 0, $noreg, 20 :: (store 4 into %ir.2)
    MOV32mi %stack.2, 1, $noreg, 0, $noreg, 30 :: (store 4 into %ir.3)
```

Listing 4.9: Fragmento de MachineInstr de CLANG/LLVM del archivo machineinstrs [1] para GCC.

IV Assembler

Por ultimo, la salida en codigo *assembler* de ambos compiladores es practicamente la misma. A continuacion, se expone la funcion suma expresada en codigo *assembler* destacando la similitud entre ambos compiladores aunque los codigos intermedios sean notoriamente diferentes en el proceso.

```
int suma (int a, int b) {
    return a + b;
}
```

Listing 4.10: Fragmento del codigo fuente del archivo codigo-ejemplo.c [1] para GCC.

```
    .globl  suma
    .type suma, @function
suma:
    .LFB1:
    .cfi_startproc
    endbr64
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
    .cfi_def_cfa_register 6
    movl  %edi, -4(%rbp)
    movl  %esi, -8(%rbp)
    movl  -4(%rbp), %edx
    movl  -8(%rbp), %eax
    addl  %edx, %eax
    popq  %rbp
    .cfi_def_cfa 7, 8
    ret
```

```
    .cfi_endproc
.LFE1:
    .size suma, .-suma
```

Listing 4.11: Fragmento del codigo *assembler* de GCC del archivo codigo-ejemplo.s [1] para GCC.

```
  .globl  suma                    # -- Begin function suma
  .p2align  4, 0x90
  .type suma,@function
suma:                                 # @suma
  .cfi_startproc
# %bb.0:
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset %rbp, -16
  movq  %rsp, %rbp
  .cfi_def_cfa_register %rbp
  movl  %edi, -8(%rbp)
  movl  %esi, -4(%rbp)
  movl  -8(%rbp), %eax
  addl  -4(%rbp), %eax
  popq  %rbp
  .cfi_def_cfa %rsp, 8
  retq
.Lfunc_end1:
  .size suma, .Lfunc_end1-suma
  .cfi_endproc
                                      # -- End function
```

Listing 4.12: Fragmento del codigo *assembler* de CLANG/LLVM del archivo assembly.s [1] para GCC.

Bibliografía

[1] J. Gonzalez, “Compiladores, codigo-ejemplo.”