Generación de Código

CI4722 – Lenguajes de Programación III

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Generación de Código

- Fase final del compilador
  - Transforma el código de tres direcciones.
  - Se apoya en la Tabla de Símbolos.
  - Produce código final para la plataforma destino.
- Requerimientos estrictos sobre el código generado
  - Debe preservar la semántica del programa original.
  - Debe hacer uso efectivo de los recursos de la máquina destino.
  - Debe ser eficiente en la transformación.

UNIVERSIDAD SIMÓN BOLÍVAR

Generación de Código

Algunos problemas

- El problema de generación de código óptimo no es decidible.
  - Cualquiera genera código inocente – pocos generan código eficiente.
  - Aplicaremos heurísticas que generan buen código, aunque no necesariamente óptimo.
- Muchos de los problemas asociados con generar código son *intratables* (NP-Completos o NP-Duros).
  - Técnicas de Programación Dinámica.
  - Heurísticas algorítmicas.

Challenge accepted!

UNIVERSIDAD SIMÓN BOLÍVAR

Tareas del Generador de Código

- Seleccionar instrucciones – ¿cuáles instrucciones de la plataforma destino representan mejor a las instrucciones intermedias?
- Asignar registros – ¿cuáles valores se mantendrán en cuáles registros de la plataforma destino?
- Ordenar instrucciones – ¿cómo reorganizar la ejecución de las instrucciones para mejorar el desempeño?

UNIVERSIDAD SIMÓN BOLÍVAR

Preocupaciones del generador de código

¿Cuál es el insumo?

- Supondremos que el Generador de Código trabaja sobre la representación intermedia
  - DAG de código de tres direcciones.
  - Libre de errores de tipos – posiblemente incluye conversiones.
- Información relevante en la Tabla de Símbolos
  - Tamaño – reservar espacio en memoria.
  - Desplazamientos – ubicación absoluta o relativa.

La representación intermedia influye dramáticamente en el diseño del generador – escojan cuidadosamente.

UNIVERSIDAD SIMÓN BOLÍVAR

Preocupaciones del generador de código

¿Cuál es el producto?

- Depende de la arquitectura – RISC vs. CISC
  - Cantidad de registros.
  - Métodos de direccionamiento.
  - Variedad de instrucciones.

UNIVERSIDAD SIMÓN BOLÍVAR

Preocupaciones del generador de código

¿Cuál es el producto?

- Depende de la arquitectura – RISC vs. CISC
  - Cantidad de registros.
  - Métodos de direccionamiento.
  - Variedad de instrucciones.
- Código absoluto – ejecutable directo.

UNIVERSIDAD SIMÓN BOLÍVAR

Preocupaciones del generador de código

¿Cuál es el producto?

- Depende de la arquitectura – RISC vs. CISC
  - Cantidad de registros.
  - Métodos de direccionamiento.
  - Variedad de instrucciones.
- Código absoluto – ejecutable directo.
- Código reubicable (*relocatable object code*)
  - Módulos objeto compilables por separado.
  - Costo adicional de enlazado.
  - Eso es lo que se hace en la vida real.

UNIVERSIDAD SIMÓN BOLÍVAR

Preocupaciones del generador de código

¿Cuál es el producto?

- Depende de la arquitectura – RISC vs. CISC
  - Cantidad de registros.
  - Métodos de direccionamiento.
  - Variedad de instrucciones.
- Código absoluto – ejecutable directo.
- Código reubicable (*relocatable object code*)
  - Módulos objeto compilables por separado.
  - Costo adicional de enlazado.
  - Eso es lo que se hace en la vida real.
- Código ensamblable.
  - Más fácil de depurar.
  - Costo adicional de ensamblado.

UNIVERSIDAD SIMÓN BOLÍVAR

Preocupaciones del generador de código

¿Cuál es el producto?

- Depende de la arquitectura – RISC vs. CISC
  - Cantidad de registros.
  - Métodos de direccionamiento.
  - Variedad de instrucciones.
- Código absoluto – ejecutable directo.
- Código reubicable (*relocatable object code*)
  - Módulos objeto compilables por separado.
  - Costo adicional de enlazado.
  - Eso es lo que se hace en la vida real.
- Código ensamblable.
  - Más fácil de depurar.
  - Costo adicional de ensamblado.
- *Bytecode* para máquinas virtuales.

UNIVERSIDAD SIMÓN BOLÍVAR

La máquina destino hipotética

Ni CISC, ni RISC sino todo lo contrario

- Capaz de acceder a *cualquier* byte de memoria – palabras de 4 bytes.
- Tiene n registros de propósito general `R0`, `R1`, ..., `R`$n-1$.
- Conjunto limitado de instrucciones – focus!
  - *Cargar* (memoria a registro, o registro a registro) – `LD` *dst*, *addr*
  - *Almacenar* (sólo hacia memoria) – `ST` *dst*, *reg*
  - *Calcular* – `OP` *dst*, *src1*, *src2*
    - `OP` puede ser `ADD`, `SUB`, `MUL`, ...
    - Operar sobre *src1* y *src2* dejando el resultado en *dst*.
    - *src1*, *src2* y *dst* podrían ser el mismo.
  - *Salto* – `BR` *label*
  - *Salto condicional* – `B`*cond r*, *label*
    - *cond* puede ser `LTZ` (*less than zero*), `Z` (*zero*) ...
    - El salto se ejecuta si *cond* es cierta para *r*.

UNIVERSIDAD SIMÓN BOLÍVAR

La máquina destino hipotética

Ni CISC, ni RISC sino todo lo contrario

Modo	Forma	Dirección Accedida
Registro	Ri	Ri
Absoluto	c	c
Indizado	c(R)	$c + contents$(R)
Registro Indirecto	*R	$contents$(R)
Indizado Indirecto	*c(R)	$contents(c + contents$(R))

- R es un registro.
- c es una dirección de memoria – etiqueta o valor.
- Usaremos #c para referirnos al valor c como constante.

UNIVERSIDAD SIMÓN BOLÍVAR

La máquina destino hipotética

Algunos ejemplos – aritmética

```
x := y - z
```

$\Longrightarrow$

```
LD   R1, y
LD   R2, z
SUB  R1, R1, R2
ST   x,  R1
```

- El flujo de datos es de derecha a izquierda.
- En las operaciones:
  1. Se obtienen los valores de los operandos.
  2. Se calcula el resultado.
  3. Se almacena el resultado en el destino.

UNIVERSIDAD SIMÓN BOLÍVAR

La máquina destino hipotética

Algunos ejemplos – acceso a arreglos

```
b := a[i]
```

$\Longrightarrow$

```
LD   R1, i
MUL  R1, R1, 8
LD   R2, a(R1)
ST   b,  R2
```

```
a[j] := c
```

$\Longrightarrow$

```
LD   R1, c
LD   R2, j
MUL  R2, R2, 8
ST   a(R2), R1
```

- a es un arreglo de elementos de 8 bytes indizado desde cero.

UNIVERSIDAD SIMÓN BOLÍVAR

La máquina destino hipotética

Algunos ejemplos – apuntadores

`x := *p` $\Longrightarrow$

```
LD   R1, p
LD   R2, 0(R1)
ST   x,  R2
```

`*p := y` $\Longrightarrow$

```
LD   R1, p
LD   R2, y
ST   0(R1), R2
```

- Alternativa – usar el modo registro indirecto

UNIVERSIDAD SIMÓN BOLÍVAR

La máquina destino hipotética

Algunos ejemplos – saltos

```
if x < y goto L
```

$\Longrightarrow$

```
LD   R1, x
LD   R2, y
SUB  R1, R1, R2
BLTZ R1, L'
```

- L' es la etiqueta *real* para la instrucción L.

UNIVERSIDAD SIMÓN BOLÍVAR

El costo de las instrucciones

Porque "mejor" requiere un punto de referencia

- Para poder comparar dos fragmentos de código generado hará falta atribuir un costo de ejecución a cada uno.
- Cada instrucción de máquina
  - Requiere *espacio* para ser almacenada.
  - Requiere *tiempo* para ser ejecutada.
- Método simple – costo de modos de direccionamiento más uno.
- Aumenta la complejidad en máquinas reales.
  - Considerar pipelines, caché combinado o separado, ...
  - Operaciones sobre tipos de datos diferentes.

UNIVERSIDAD SIMÓN BOLÍVAR

El costo de las instrucciones

Ejemplos

Instrucción	Operaciones	Costo
`LD R0, R1`	$R0 \leftarrow contents(R1)$	1
`LD R0, M`	$R0 \leftarrow contents(M)$	2
`ST M, R0`	$M \leftarrow contents(R0)$	2
`LD R1, *4(R0)`	$R1 \leftarrow contents(contents(4 + contents(R0)))$	3
`ST M, *4(R0)`	$M \leftarrow contents(contents(4 + contents(R0)))$	4
`LD R0, #42`	$R0 \leftarrow 42$	1

- Acceso a un registro tiene costo cero.
- Accesos a memoria o indirectos tienen costo uno – la dirección o el *offset* ocupan espacio.
- Accesos a constantes tienen costo uno – el valor ocupa espacio *casi* siempre.

UNIVERSIDAD SIMÓN BOLÍVAR

Selección de instrucciones

Generación inocente por *plantillas*

- Cada instrucción intermedia tiene una **plantilla de generación**.
- Los símbolos particulares que ocurren en una instancia de la instrucción son sustituidos en la plantilla.

Supongamos que la instrucción `t0 := t1 + t2` tiene plantilla

```
LD   R0, t1
ADD  R0, R0, t2
ST   t0, R0
```

UNIVERSIDAD SIMÓN BOLÍVAR

Selección de instrucciones

Las plantillas pueden ocasionar redundancia

- Para cada instrucción intermedia, sustituir por su plantilla con los cambios de variable necesarios.
- No prestar atención al contexto – mera sustitución de símbolos.

Selección de instrucciones

Las plantillas pueden ocasionar redundancia

- Para cada instrucción intermedia, sustituir por su plantilla con los cambios de variable necesarios.
- No prestar atención al contexto – mera sustitución de símbolos.

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

$\Longrightarrow$

```
LD   R0, b
ADD  R0, R0, c
ST   a,  R0
LD   R0, a
ADD  R0, R0, e
ST   d,  R0
```

UNIVERSIDAD SIMÓN BOLÍVAR

Selección de instrucciones

Las plantillas pueden ocasionar redundancia

- Para cada instrucción intermedia, sustituir por su plantilla con los cambios de variable necesarios.
- No prestar atención al contexto – mera sustitución de símbolos.

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

$\Longrightarrow$

```
LD    R0, b
ADD   R0, R0, c
ST    a,  R0
LD    R0, a
ADD   R0, R0, e
ST    d,  R0
```

Las optimizaciones globales se encargarán de esto.

UNIVERSIDAD SIMÓN BOLÍVAR

Selección de instrucciones

Las plantillas pueden ocasionar redundancia

```
a := a + 1
```

$\Longrightarrow$

```
LD    R0, a
ADD   R0, R0, #1
ST    a,  R0
```

Selección de instrucciones

Las plantillas pueden ocasionar redundancia

```
a := a + 1
```

$\Longrightarrow$

```
LD    R0, a
ADD   R0, R0, #1
ST    a,  R0
```

```
ADD   a, a, #1
```

```
INC   a
```

- El código generado tiene costo 5 o 6 – ¿#1 requiere espacio?
- Alternativa con una sola instrucción tiene costo 3 o 4.
- Ganamos si la plataforma provee `INC`.

UNIVERSIDAD SIMÓN BOLÍVAR

Selección de instrucciones

Las plantillas pueden ocasionar redundancia

```
a := a + 1
```

$\Longrightarrow$

```
LD    R0, a
ADD   R0, R0, #1
ST    a,  R0
```

```
ADD   a, a, #1
```

```
INC   a
```

- El código generado tiene costo 5 o 6 – ¿#1 requiere espacio?
- Alternativa con una sola instrucción tiene costo 3 o 4.
- Ganamos si la plataforma provee `INC`.

Las optimizaciones locales se encargarán de esto.

UNIVERSIDAD SIMÓN BOLÍVAR

Selección de instrucciones

Las plantillas pueden ocasionar redundancia

```
a := b + c
```

$\Longrightarrow$

```
LD    R0, b
ADD   R0, R0, c
ST    a,  R0
```

```
LD    R0, #a
LD    R1, #b
LD    R2, #c
...
LD   *R0, *R1
ADD  *R0, *R0, *R1
```

```
LD    R1, b
LD    R2, c
...
ADD   R1, R1, R2
LD    a,  R1
```

- Direccionamiento indirecto vía registros ahorra instrucciones.
- Conocer la configuración previa puede mejorar la generación.

UNIVERSIDAD SIMÓN BOLÍVAR

Selección de instrucciones

Las plantillas pueden ocasionar redundancia

```
a := b + c
```

$\Longrightarrow$

```
LD    R0, b
ADD   R0, R0, c
ST    a,  R0
```

```
LD    R0, #a
LD    R1, #b
LD    R2, #c
...
LD   *R0, *R1
ADD  *R0, *R0, *R1
```

```
LD    R1, b
LD    R2, c
...
ADD   R1, R1, R2
LD    a,  R1
```

- Direccionamiento indirecto vía registros ahorra instrucciones.
- Conocer la configuración previa puede mejorar la generación.

Las optimizaciones globales se encargarán de esto.

Preocupaciones del generador de código

¿Cómo seleccionar instrucciones?

- La brecha entre el nivel de la representación intermedia y el lenguaje destino sugiere el empleo de plantillas.
- Código subóptimo, pero fácil de generar – dejamos el problema de optimización para más adelante.
- Es necesario conocimiento profundo de **todo** el juego de instrucciones de la máquina destino
  - Para aprovechar modos de direccionamiento.
  - Para aprovechar "jerga" del procesador.
- Es necesario estudiar contextos locales y globales del programa intermedio para tomar buenas decisiones.

UNIVERSIDAD SIMÓN BOLÍVAR

Preocupaciones del generador de código

¿Cómo asignar registros?

- Un registro es la unidad de cómputo más rápida – por eso hay pocos.
- **Register Allocation** – Para un punto particular del programa, decidir cuáles variables estarán en registros.
- **Register Assignment** – Llegado el momento, decidir cuál registro contendrá a cada variable particular.
- Encontrar una asignación óptima es NP-completo – incluso si solamente hay **un** registro.
- El problema se complica en plataformas con registros especializados – aritmética, direccionamiento, acceso indizado (¡hola Intel!).

UNIVERSIDAD SIMÓN BOLÍVAR

Asignación de Registros

No es un problema de cantidad

Asignación

```
t := a * b
t := t + a
t := t / d
```

R1←t

```
LD   R1, a
MUL  R1, R1, b
ADD  R1, R1, a
DIV  R1, R1, d
ST   t,  R1
```

```
t := a * b
t := t + a
t := t / d
```

R0←a
R1←t

```
LD   R0, a
LD   R1, R0
MUL  R1, R1, b
ADD  R1, R1, R0
DIV  R1, R1, d
ST   t,  R1
```

UNIVERSIDAD SIMÓN BOLÍVAR

Preocupaciones del generador de código

¿Cómo evaluar las expresiones?

- Un orden de cómputo puede...
  - ...requerir menos registros.
  - ...requerir menos instrucciones.
  - ...requerir menos transferencias a memoria.
- Encontrar un ordenamiento óptimo para cualquiera de esos criterios es NP-completo.
- La mayoría de los lenguajes aplica heurísticas para casos borde o simplemente genera en el mismo orden que el intermedio.
- Aprovechar arquitecturas superescalares o pipelines es un problema para el optimizador especializado.

UNIVERSIDAD SIMÓN BOLÍVAR

Reordenamiento de expresiones

Con el orden emitido por el generador intermedio

```
a+b-(c+d)*e
```

```
t1 := a + b
t2 := c + d
t3 := e * t2
t4 := t1 - t3
```

```
LD    R0, a
ADD   R0, R0, b
ST    t1, R0
LD    R1, c
ADD   R1, R1, d
LD    R0, e
MUL   R0, R0, R1
LD    R1, t1
SUB   R1, R1, R0
ST    t4, R1
```

UNIVERSIDAD SIMÓN BOLÍVAR

Reordenamiento de expresiones

Reordenando para evaluar primero el sustraendo

```
a+b-(c+d)*e
```

```
t2 := c + d
t3 := e * t2
t1 := a + b
t4 := t1 - t3
```

```
LD    R0, c
ADD   R0, R0, d
LD    R1, e
MUL   R1, R1, R0
LD    R0, a
ADD   R0, R0, b
SUB   R0, R0, R1
ST    t4, R0
```

UNIVERSIDAD SIMÓN BOLÍVAR

Bibliografía

- [*Aho*]
  - Secciones 8.1 y 8.2.
  - Ejercicios 8.2.1 a 8.2.6.

UNIVERSIDAD SIMÓN BOLÍVAR