Generación de Código Intermedio (CIL)

MSc. Alejandro Piad Morffis

El lenguaje CIL

Compilar de COOL a MIPS directamente es muy difícil :(

Necesitamos un lenguaje intermedio...

Class Intermediate Language (CIL)

- Lenguaje de máquina con instrucciones de 3 direcciones
- Todas las variables son enteros de 32 bits
- Tiene soporte para operaciones orientadas a objetos

Un programa en CIL

Primero, declaraciones de "tipos" (aguanten por ahora):

```
.TYPES

type A {
    attribute x ;
    method f : f1;
}

type B {
    attribute x ;    # Los atributos heredados con
    attribute y ;    # el mismo nombre y orden
    method f : f2 ; # Método sobre-escrito
    method g : f3 ; # Método nuevo
}
```

Un programa en CIL

Luego tenemos una parte para las constantes `string`.

```
.TYPES

...

.DATA

msg = "Hello World";
```

Un programa en CIL

Al final los métodos en sí (no tienen tipos asociados):

```
.CODE

function f1 {
    ...
}

function f2 {
    ...
}
```

Anatomía de un método

```
function f1 {
    PARAM x;   # argumentos (que vienen de afuera)
    PARAM y;

    LOCAL a; # variables locales (memoria propia)
    LOCAL b;

    <body>   # instrucciones
}
```

Algunos tipos de instrucciones

- Asignación:

```
x = y ;
```

- Operaciones aritméticas:

```
x = y + z ;
```

- Acceso a atributos (`x = y.b`):

```
x = GETATTR y b ;   # x = y.b
SETATTR y b x ;     # y.b = x
```

- Arrays y `strings`:

```
x = GETINDEX a i ; # x = a[i]
SETINDEX a i x ;   # a[i] = x
```

Algunos tipos de instrucciones

Manipulación de memoria:

- Creación:

  ```
  x = ALLOCATE T ;
  ```

- Creación de un array:

  ```
  x = ARRAY y ;
  ```

- ¿De qué tipo es una variable?

  ```
  t = TYPEOF x ;
  ```

Algunas cuestiones importantes hasta ahora

¡ Todos los valores son enteros de 32 bits !

Esto significa que la interpretación de un valor depende de cómo se use.

En `x = y + z` todo se interpreta como un valor entero.

En `x = GETATTR y z` pasan varias cosas interesantes:

- `y` es un entero que se interpreta como un tipo.
- `z` es un entero que se interpreta como un atributo (realmente como el **offset** del atributo).
- `x` realmente no importa, guardará lo que sea que haya en `y.z`, ya sea referencia o valor, cuando se use más adelante será interpretado.

Instrucciones de Saltos

- Etiquetas obligatorias:

  ```
  LABEL label ;
  ```

- Salto incondicional (ya son adultos para esto):

  ```
  GOTO label ;
  ```

- Salto condicional:

  ```
  IF x GOTO label ;
  ```

Las expresiones *booleanas* no existen:

```
x = y != z;
IF x GOTO label ;
```

```
x = y - z ;
IF x GOTO label ;
```

Invocación de métodos

- Invocación estática:

  ```
  x = CALL f ;
  ```

- Invocación dinámica:

  ```
  x = VCALL T f ;
  ```

- Paso de parámetros:

  ```
  ARG a ;
  ```

- Retorno (obligatorio):

  ```
  RETURN x ;     RETURN 0 ;     RETURN ;
  ```

  Es nuestra responsabilidad ser cuidadosos con los parámetros.

Funciones de cadena e IO

- Cargar dirección:

```
x = LOAD msg ;
```

- Algunos operadores:

```
y = LENGTH x ;
y = CONCAT z w ;
y = PREFIX x n ;
y = SUBSTRING x n ;
```

- Convertir entero a cadena:

```
z = STR y ;
```

- Entrada salida:

```
x = READ ;        PRINT x ;
```

Hola Mundo

```
.DATA

msg = "Hello World!\n" ;

.CODE

function main {
    LOCAL x ;

    x = LOAD msg ;
    PRINT x ;
    RETURN 0 ;
}
```

Hola Mundo (desde COOL)

Este es el código en COOL:

```
class Main: IO {
    msg : string = "Hello World!\n";

    function main() : IO {
        self.print(self.msg);
    }
}
```

Hola Mundo (desde COOL)

Primero los tipos:

```
.TYPES

type Main {
    attribute Main_msg ;    # fíjense en los nombres
    method Main_main: f1 ; # de los métodos y atributos
}

.DATA

s1 = "Hello World!\n";
```

Hola Mundo (desde COOL)

Esta es la función main

```
.CODE
# <...> aquí falta algo

function f1 {
    PARAM self ;

    LOCAL lmsg ;

    lmsg = GETATTR self Main_msg ;
    PRINT lmsg ;

    RETURN self ;
}
```

Hola Mundo (desde COOL)

```
.CODE

function entry {
    LOCAL lmsg ;
    LOCAL instance ;
    LOCAL result ;

    lmsg = LOAD s1 ;
    instance = ALLOCATE Main ;          # ...
    SETATTR instance Main_msg lmsg ; # estos nombres
                                       # realmente son
    ARG instance ;                   # solo números !
    result = VCALL Main Main_main ;  # ...

    RETURN 0 ;
}
```

Convenios para generación de código

```
let x : Integer = 5 in    # esto es parte de un
    x + 1                 # programa de COOL más
end                       # largo


function f {
    ...
    LOCAL x ;
    LOCAL <value> ;    # <value> es la variable
                       # donde se guarda el resultado
    ...                # intermedio
    x = 5 ;
    <value> = x + 1 ;
    ...
}
```

Convenios para generación de código

```
let x : Integer = 5 in
    x + <expr>        # expresión arbitraria
end


function f {
    ...
    <expr.locals>    # recursivamente, generar
    LOCAL x ;        # las variables locales
    LOCAL <value> ; # necesarias

    ...
    x = 5 ;
    <expr.code>      # generar el código
    <value> = x + <expr.value> ;
}
```

Generación de Let-In

```
let <var> : <type> = <init> in  # expresión let-in
    <body>                      # arbitraria
end

function f {
    ...
    <init.locals>    # recursivamente generar
    LOCAL <var> ;    # todas las variables
    <body.locals>    # locales en el orden
    LOCAL <value> ;  # que hace falta

    ...
    <init.code>                  # queda expresado
    <var> = <init.value> ;       # el orden de
    <body.code>                  # generación de
    <value> = <body.value> ;     # código explícitamente
}
```

Algunas cosas a tener en cuenta

De nuevo, todo son números!

- No hay ningún tipo de chequeo ni ayuda por el runtime
- El significado de una variable depende 100% de su uso
- Es imprescindible tener cuidado con el orden de las definiciones
- En CIL las variables no se ocultan, no hay ámbito ni visibilidad
- Toda la consistencia se garantiza porque hicimos el chequeo semántico

Se ve bastante claro como llegar desde COOL hasta CIL, ¿verdad?

Al menos, bastante más claro que intentar llegar directo a un lenguaje ensamblador *de verdad*.