El Lenguaje Intermedio Común (CIL) para la asignatura de Procesadores de Lenguaje (v.0.1)*

Jorge Calera Rubio
Alicia Garrido Alenda
Juan Antonio Pérez Ortiz
Pedro Ponce de León Amador
David Rizo Valero

9 de diciembre de 2009

Introducción

El CIL[1] es el lenguaje de más bajo nivel en la *Infraestructura para el Lenguaje Común* (CLI[2]) y en la plataforma .NET.[3] Los compiladores para cualquier lenguaje fuente de dicha plataforma generan este lenguaje intermedio que es ensamblado en *bytecodes*. El CIL es ejecutado por una máquina virtual basada en pila y, por lo tanto, es independiente de la plataforma. También se pueden crear aplicaciones utilizando distintos lenguajes fuente, siempre que todos ellos generen CIL. Es un lenguaje ensamblador orientado a objetos que adicionalmente dispone de recolector de basura, manejo de excepciones y otras características de los lenguajes modernos, por lo que no es necesario que el diseñador del programa gestione explícitamente la memoria.

El CIL fue originalmente conocido como MSIL (*Microsoft Intermediate Language*), y desde la estandarización de éste por los organismos internacionales ECMA e ISO/IEC (a propuesta de Microsoft, HP e Intel) es oficialmente conocido por su denominación actual. La especificación de la CLI (y del CIL) se recoge en el *Standard ECMA-335*.[4] La aparición de este documento ha permitido que surjan varios proyectos de software libre para trabajar con la CLI que generan código portable a la plataforma .NET. Cabe destacar entre ellos el proyecto *Mono* (`http://www.mono-project.com/`), con más de diez lenguajes fuente compatibles, y el proyecto *DotGNU* (`http://dotgnu.org/`) con compiladores para C y C#. Las herramientas del proyecto DotGNU se distribuyen

[1] Del inglés *Common Intermediate Language*.

[2] Del inglés *Common Language Infraestructure*.

[3] Cita de la Wikipedia.

[4] `http://www.ecma-international.org/ publications/standards/Ecma-335.htm`.

en un paquete llamado `pnet`; las de Mono en los paquetes `mono`, `mono-utils`, `mono-runtime` y `mono-gmcs`.

Como ya se ha comentado más arriba, un programa en CIL debe ser ensamblado en *bytecodes* para luego ser ejecutado por la máquina virtual. Para esta tarea, en DotGNU se utilizan los comandos `ilasm.pnet` e `ilrun`. El compilador de C y C# se invoca con la orden `cscc`. Adicionalmente se pueden desensamblar los bytecodes (generados con cualquier lenguaje de programación compatible) en CIL mediante el comando `ildasm`. En Mono, por otra parte, se usan las órdenes `ilasm2`, `mono`, `gmcs` y `monodis`.

Es preciso mencionar que el presente documento no pretende ser una descripción exhaustiva del CIL (el lector interesado puede consultar para ello el Standard ECMA-335), sino dar, simplemente, una descripción de unas pocas características de éste suficientes para utilizarlo en la asignatura de Procesadores de Lenguaje de la Universidad de Alicante.

Información general

Todas las operaciones en CIL se realizan en la *pila de evaluación*. Cuando, por ejemplo, una función[5] es llamada, sus parámetros son colocados en la pila. El código de la función se inicia con este estado de la pila y puede poner algunos valores más en ella, haciendo operaciones con ellos y/o desapilándolos.

La ejecución de las instrucciones y funciones en CIL se realiza en tres pasos:

1. Apilar los operandos de una instrucción o los parámetros de una función.
2. Ejecutar la instrucción CIL o la llamada a la función. La instrucción o función desapila los operandos (parámetros) y apila el resultado (valor devuelto).
3. Leer el resultado de la pila (desapilarlo).

Los pasos 1 y 3 son opcionales. Por ejemplo, una función `void` no apila ningún valor devuelto.

La pila puede contener objetos de *tipo valor* y referencias a objetos de *tipo referencia*. Los objetos de *tipo referencia* se guardan en el *heap*.

Los comandos CIL utilizados para apilar valores comienzan con las letras `ld...` (**load**). Los utilizados para desapilar y almacenar los valores en variables por `st...` (**store**).

La librería estándar del CIL se llama `mscorlib`. En ella están definidas muchas clases y métodos que son utilizados en los programas. Para poder acceder a estas clases y métodos es necesario incluirla con la directiva

```
.assembly extern mscorlib{}
```

Para evitar colisiones entre las palabras reservadas del CIL y los símbolos definidos por el usuario se recomienda que todos éstos vayan encerrados entre apóstrofos (p.ej. `'miVariable'`).

Todo el texto encerrado entre `/*` y `*/` se considera comentario, al igual que todo el texto que aparece a continuación de `//` hasta el final de la línea.

[5] A lo largo de este documento utilizaremos, indistintamente, los términos método y función.

Directivas

De entre las numerosas directivas para el ensamblador con las que cuenta el CIL destacaremos las siguientes:

- `.assembly`: esta directiva especifica a qué unidad de ejecución pertenece el módulo en el que se declara. Cuando va acompañada del atributo `extern` referencia a una unidad de ejecución externa que contiene la definición de algún método/clase que se utiliza en el módulo actual (el equivalente a `#include` en C o C++). Ejemplos:

  ```
  .assembly 'MiModulo' {}
  .assembly extern mscorlib {}
  ```

- `.class`: se utiliza para definir una clase. La sintaxis que usaremos en la asignatura es

  `.class` <nombre> `extends [mscorlib]System.Object {...}`

  Entre las llaves se incorpora la definición de los campos y métodos de la clase.

- `.field`: define un campo de una clase. La sintaxis que se utilizará en la asignatura es

  `.field` <atrib> <tipo> <nombre>

  <atrib> suele ser `private`, aunque también se puede utilizar, según las características del lenguaje fuente, el atributo `public` y muchos otros atributos disponibles en CIL (como `static`). <tipo> puede ser cualquiera de los numerosos tipos del Sistema Común de Tipos (CTS) disponible en CIL, si bien en la asignatura sólo se utilizarán los tipos básicos `int32` y `float64`, los arrays de dichos tipos `int32 []` y `float64 []`, y los objetos de tipo *clase* definidos por el usuario. En la librería `mscorlib` se encuentran definidas las clases `System.Int32` y `System.Double` que empaquetan los tipos básicos anteriores y permiten acceder a numerosos métodos para gestionarlos.

- `.method`: define un método o función. Su sintaxis será

  `.method` <atrib> <tipo> <nombre> `(` <args> `) cil managed {...}`

  <atrib> es una lista de cero o más atributos entre los que destacaremos:

  - `public | private`: para métodos públicos o privados respectivamente (en las prácticas de la asignatura todos son públicos)
  - `static` cuando no es necesaria una instancia de una clase para invocar el método. Esto puede ocurrir por dos motivos: porque el método sea global (en este caso para invocar el método únicamente se requiere el nombre); o porque, aunque sea un método de clase, se haya declarado explícitamente como estático (en este caso, para invocarlo, es necesario anteponer al nombre del método el nombre de la clase a la que pertenece).

    Aunque por defecto es necesaria una instancia de una clase para invocar un método no declarado como estático, se puede declarar explícitamente esto mediante el atributo `instance`.

<tipo> es el tipo del valor devuelto por la función (léase el apartado correspondiente a `.field`).

<args> es una lista con los tipos de los argumentos del método separados por comas. Junto al tipo de cada argumento se puede, opcionalmente, añadir un nombre. Los dos siguientes ejemplos son equivalentes, en el primero se deberá referenciar a los argumentos por su posición, en el segundo se podrá, además, referenciarlos por su nombre:

```
.method int32 'miMetodo' ( int32, float64)
.method int32 'miMetodo' ( int32 'a', float64 'b')
```

Entre las llaves se incorpora el código CIL del método.

Los métodos se pueden declarar dentro de una clase cuando son locales a ella y fuera de cualquier clase cuando son métodos globales (en la asignatura el único método global será el programa principal).

Hay un método especial llamado `.ctor` que se utiliza como constructor de la clase que lo contiene. En este método debe ir el código necesario para inicializar los miembros de tipo array y debe contener, como mínimo, el siguiente código:

```
.method public specialname rtspecialname instance void .ctor()
 cil managed
{
     .maxstack 1
     ldarg 0
     call instance void [mscorlib]System.Object::.ctor()
     ret
}
```

Donde "`call instance void [mscorlib]System.Object::.ctor()`" es la llamada al constructor de la clase `System.Object`, de la cual derivan todas las clases (ver más abajo).

- `.entrypoint`: todas las aplicaciones en CIL deben tener un *punto de entrada*. Cualquier método (no necesariamente el llamado `main`) puede servir de punto de entrada mientras contenga esta directiva y se haya declarado con los atributos `static` y `public`. Sólo puede haber un punto de entrada por aplicación.
- `.maxstack` <valor>: define la máxima profundidad de la pila utilizada por el código de un método. Es obligatorio incluir esta directiva, aunque no necesariamente al inicio del método. Si se especifica un valor inferior al realmente necesario, la máquina virtual, `ilrun`, producirá un error.
- `.locals(`<tipos>`)`: declara las variables locales de un método. Para ello, entre los paréntesis, y separados por coma, deben ir las descripciones del tipo de las variables (léase el apartado correspondiente a `.field`). Opcionalmente también se puede añadir a esta descripción un nombre para las variables. Las variables locales pueden referenciarse por la posición que ocupan en esta lista (comenzando por 0) o por el nombre. Ejemplos:
  `.locals(int32, float64[], class 'MiClase')`
  `.locals(int32 'i', float64[] 'MiArr', class 'MiClase' 'MiObj')`

Instrucciones

La máquina virtual conoce, en cualquier momento, el tipo de los datos en la pila, motivo por el cual hay un único operador para cada operación, independientemente del tipo de los operandos. Es por otra parte responsabilidad del programador que los operandos sean de tipos compatibles con la operación que se va a realizar, efectuando, si fuera necesario, las conversiones de tipo oportunas. Así, por ejemplo, los operadores aritméticos `add`, `sub`, `div` o `mul` pueden operar con enteros o reales, pero no con ambos a la vez. Las conversiones de tipo dependerán de las especificaciones del lenguaje fuente.

Se pueden utilizar etiquetas para marcar posiciones concretas en el código CIL. Estas etiquetas deben ir al comienzo de la línea y seguidas por dos puntos (`MiEtiq:`).

Se describe a continuación el subconjunto de instrucciones del CIL que se utilizará en la asignatura. Esta descripción viene acompañada de un diagrama de transición que muestra el estado de la pila de evaluación antes y después de que se ejecute la instrucción. Un diagrama típico tiene la forma
..., *valor1*, *valor2* ⟶ ..., *resultado*
indicando que la pila debe tener al menos dos elementos, siendo *valor2* el tope (la pila crece hacia la derecha). La instrucción desapila los operandos (en este caso *valor1* y *valor2* y apila *resultado*.

`add` : suma dos valores, devuelve el resultado
..., *valor1*, *valor2* ⟶ ..., *resultado*

`div` : devuelve el resultado de dividir *valor1* por *valor2*
..., *valor1*, *valor2* ⟶ ..., *resultado*

`mul` : multiplica dos valores, devuelve el resultado
..., *valor1*, *valor2* ⟶ ..., *resultado*

`sub` : devuelve el resultado de restar *valor2* a *valor1*
..., *valor1*, *valor2* ⟶ ..., *resultado*

`neg` : devuelve el valor negado (cambio de signo)
..., *valor* ⟶ ..., *resultado*

`conv.i4` : convierte a entero de 4 bytes (`int32`)
..., *valor* ⟶ ..., *resultado*

`conv.r8` : convierte a coma flotante de 8 bytes (`float64`)
..., *valor* ⟶ ..., *resultado*

`beq` ***etiq*** : salta a la posición de código etiquetada con *etiq* si *valor1* es igual a *valor2*
..., *valor1*, *valor2* ⟶ ...

`bge` ***etiq*** : salta a la posición de código etiquetada con *etiq* si *valor1* es mayor o igual que *valor2*
..., *valor1*, *valor2* ⟶ ...

`bgt` ***etiq*** : salta a la posición de código etiquetada con *etiq* si *valor1* es mayor que *valor2*
..., *valor1*, *valor2* ⟶ ...

`ble` ***etiq*** **:** salta a la posición de código etiquetada con *etiq* si *valor1* es menor o igual que *valor2*
..., *valor1*, *valor2* ⟶ ...

`blt` ***etiq*** **:** salta a la posición de código etiquetada con *etiq* si *valor1* es menor que *valor2*
..., *valor1*, *valor2* ⟶ ...

`bne.un` ***etiq*** **:** salta a la posición de código etiquetada con *etiq* si *valor1* es distinto de *valor2*
..., *valor1*, *valor2* ⟶ ...

`br` ***etiq*** **:** salta a la posición de código etiquetado con *etiq*
..., ⟶ ...

`call` **:** llama a un método. Si el método es de instancia (no ha sido declarado con el atributo `static`) debe ser invocado con

`call instance` <tipo> <nomClase>`::`<nomMet> `(`<tipoParam>`)`

donde <tipo> es el tipo del valor devuelto por el método, <nomClase> es el nombre de la clase en la que se definió el método, <nomMet> es el nombre del método y <tipoParam> es una lista compuesta por los tipos de los parámetros separados por comas. Antes de los parámetros se debe apilar el objeto sobre el que se llama al método,

..., *obj*, *arg1*, *arg2*, ..., *argN* ⟶ ..., *valDev*

valDev puede no existir si el método es de tipo `void`. Es importante señalar que un método no debe 'dejar restos' en la pila. Por ejemplo, no se puede apilar algo y no desapilarlo (excepto el valor de retorno).

Si el método es estático no se debe poner el atributo `instance` (en este caso el atributo `static` es opcional), y no se debe apilar el objeto sobre el que se llama al método (que además, en general, no existirá).

`ret` **:** vuelve de un método
..., *valDev* (en el método) ⟶ ..., *valDev* (en el llamador)

Es obligatorio que toda secuencia de ejecución de instrucciones posible dentro de un método acabe con `ret`.

`ldarg` ***pos*** **:** apila el argumento que ocupa la posición *pos* en la lista de argumentos (comenzando por 0). Si el método actual es estático, el primer argumento de la lista (el 0) se corresponde con el primer argumento de la definición, pero si el método es de instancia, el argumento 0 es el puntero `this` del objeto actual, viniendo a continuación el resto de argumentos.
..., ⟶ ..., *valor*

`ldc.i4` ***num*** **:** apila *num* como `int32`
..., ⟶ ..., *num*

`ldc.r8` ***num*** **:** apila *num* como `float64`
..., ⟶ ..., *num*

`ldloc` ***pos*** **:** apila la variable local que ocupa la posición *pos* (comenzando por 0)
..., $\longrightarrow$ ..., *valor*

`ldelem.i4` **:** apila el elemento de tipo `int32` que ocupa la posición *pos* (comenzando por 0) de *array* (realmente, un vector)
..., *array*, *pos* $\longrightarrow$ ..., *valor*

El operando *array* es la dirección del array, presumiblemente almacenada previamente en un campo, variable local o argumento.

`ldelem.ref` **:** apila la referencia al objeto que ocupa la posición *pos* (comenzando por 0) de *array* (realmente, un vector)
..., *array*, *pos* $\longrightarrow$ ..., *referencia*

`ldfld` ***campo*** **:** apila el miembro *campo* del objeto *obj*
..., *obj* $\longrightarrow$ ..., *valor*

La especificación completa de *campo* tiene la forma <tipo> <nomClase>::<nomCampo>.

`starg` ***pos*** **:** almacena *valor* en el argumento que ocupa la posición *pos* (comenzando por 0)
..., *valor* $\longrightarrow$ ...,

`stloc` ***pos*** **:** almacena *valor* en la variable local que ocupa la posición *pos* (comenzando por 0)
..., *valor* $\longrightarrow$ ...,

`stelem.i4` **:** almacena *valor* de tipo `int32` en la posición *pos* de *array* (realmente, un vector)
..., *array*, *pos*, *valor* $\longrightarrow$ ...,

`stelem.r8` **:** almacena *valor* de tipo `float64` en la posición *pos* de *array*
..., *array*, *pos*, *valor* $\longrightarrow$ ...,

`stfld` ***campo*** **:** almacena *valor* en el miembro *campo* del objeto *obj*
..., *obj*, *valor* $\longrightarrow$ ...,

La especificación completa de *campo* tiene la forma <tipo> <nomClase>::<nomCampo>.

`box` ***tipoClase*** **:** convierte *valor* en el objeto *obj* de tipo *tipoClase*, permitiendo así acceder a los métodos de dicha clase, lo que es útil, por ejemplo, en operaciones de entrada/salida (ver más abajo)
..., *valor* $\longrightarrow$ ..., *obj*

`newarr` ***defTipo*** **:** crea (en el *heap*) un nuevo *array* de tamaño *numElem* del tipo definido por *defTipo* (en nuestro caso puede ser `[mscorlib]System.Int32` para arrays de `int32` o `[mscorlib]System.Double` para arrays de `float64`). Todos los elementos del array son inicializados a 0 (del tipo correspondiente)
..., *numElem* $\longrightarrow$ ..., *array*

`newobj` ***ctor*** **:** crea un objeto sin inicializar y llama al constructor *ctor*
..., *arg1*, *arg2*, ..., *argN* $\longrightarrow$ ..., *obj*

La especificación completa de la llamada a *ctor* tiene la forma (ver `call`)

`instance void` <nomClase>`::.ctor(`<tipoParam>`)`.

`pop :` desapila *valor*
..., *valor* $\longrightarrow$ ...,

Entrada/salida

Ejemplos de código CIL para algunas operaciones de salida útiles para la asignatura son los siguientes:[6]

- **Imprimir un entero sin cambio de línea**:
  `call void [mscorlib]System.Console::Write(int32)`
- **Imprimir un entero con cambio de línea**:
  `call void [mscorlib]System.Console::WriteLine(int32)`
- **Imprimir un real sin formato sin cambio de línea**:
  `call void [mscorlib]System.Console::Write(double)`
- **Imprimir un real sin formato con cambio de línea**:
  `call void [mscorlib]System.Console::WriteLine(double)`
- **Imprimir un real con formato de coma fija (8 posiciones, 3 decimales) con cambio de línea**:
  `ldstr "{0,8:F3}"`
  carga del real a imprimir
  `box [mscorlib]System.Double`
  `call void [mscorlib]System.Console::WriteLine(string,object)`
- **Imprimir un carácter sin cambio de línea**:
  `call void [mscorlib]System.Console::Write(char)`
- **Imprimir en carácter con cambio de línea**:
  `call void [mscorlib]System.Console::WriteLine(char)`

Ejemplos de código CIL para algunas operaciones de entrada útiles para la asignatura son:[7]

- **Leer un número entero de la consola**:
  `call string [mscorlib]System.Console::ReadLine()`
  `call int32 [mscorlib]System.Int32::Parse(string)`
- **Leer un número real de la consola**:
  `call string [mscorlib]System.Console::ReadLine()`
  `call float64 [mscorlib]System.Double::Parse(string)`
- **Leer un carácter de la consola** (en la pila queda su valor ASCII como `int32`):
  `call int32 [mscorlib]System.Console::Read()`
  `conv.u2` La instrucción `conv.u2` convierte el valor que hay en el tope de la pila a un entero sin signo de dos bytes (aunque el valor apilado es de tipo `int32`).

[6] Asumimos que se ha utilizado la directiva `.assembly extern mscorlib{}` y que en el tope de la pila se encuentra el valor a imprimir.

[7] Asumimos que se ha utilizado la directiva `.assembly extern mscorlib{}`. El valor leído queda en la pila.

Ejemplo

Un programa válido en CIL (no necesariamente traducción de algún programa escrito en otro lenguaje fuente) es:

```
/* Hola.il */

.assembly extern mscorlib {}
.assembly 'Hola' {}

.class 'A' extends [mscorlib]System.Object  // Definición de la clase 'A'
{
.field private int32 [] 'mens'     // campo privado array de enteros

/* Definición del constructor */
.method public specialname rtspecialname instance void .ctor() cil managed
{
        .maxstack 2              // maxima profundidad de pila utilizada
        ldarg 0                         // puntero this
        ldc.i4 10                       // tamaño del array
        newarr [mscorlib]System.Int32   // crear array de enteros
        stfld int32[] 'A'::'mens'       // guardarlo en el campo 'mens'
        ldarg 0                         // puntero this
        call instance void [mscorlib]System.Object::.ctor()
                                // llamada a .ctor de la clase padre
        ret
}// fin .ctor

/* Definición del método público 'f' */
.method public int32 'f'() cil managed
{
        .locals (int32)           // una variable local de tipo int32
        .maxstack 3
        ldarg 0                   // puntero this
        ldfld int32[] 'A'::'mens' // campo 'mens' del objeto actual (this)
        ldc.i4 0                  // apila 0
        ldc.i4 72                 // carácter 'H'
        stelem.i4                 // almacena 'H' en la posición 0 de 'mens'
        ldarg 0
        ldfld int32[] 'A'::'mens'
        ldc.i4 1
        ldc.i4 111
        stelem.i4                 // almacena 'o' en la posición 1
        ldarg 0
        ldfld int32[] 'A'::'mens'
        ldc.i4 2
        ldc.i4 108
        stelem.i4                 // almacena 'l' en la 2
        ldarg 0
        ldfld int32[] 'A'::'mens'
```

```
        ldc.i4 3
        ldc.i4 97
        stelem.i4                   // almacena 'a' en la 3
                        // Inicializar la variable local
        ldc.i4 0                    // apila un 0
        stloc 0                     // lo almacena en la variable local 0
L3:                     // WHILE-DO
        ldloc 0                     // apila la variable local
        ldc.i4 3                    // apila un 3
        blt L1                      // compara ambos valores y salta a L1 si el
                                    // primero es menor
        ldc.i4 0                    // apila un 0
        br L2                       // salta a L2
L1:     ldc.i4 1                    // apila un 1
L2:     ldc.i4 0                    // apila un 0
        beq L4                      // comprueba si hay que ejecutar el cuerpo
                                    // del bucle, si no salta a L4
        ldarg 0
        ldfld int32[] 'A'::'mens'
        ldloc 0
        ldelem.i4
        call void [mscorlib]System.Console::Write(char)
                                    // imprime "Hol"
        ldloc 0
        ldc.i4 1
        add
        stloc 0                     // actualiza el contador (la variable local)
        br L3
L4:                     // fin bucle WHILE-DO
        ldarg 0
        ldfld int32[] 'A'::'mens'
        ldloc 0
        ldelem.i4
        call void [mscorlib]System.Console::WriteLine(char)
                        /* finalizar la escritura:
                           (imprimir 'a' y salto de línea)*/
        ldc.i4 0        // apilar valor devuelto por la función (0)
        ret             // devolver el control al llamador
}

} // fin clase 'A'

.method static public void main () cil managed
{
        .locals (class 'A')         // una variable local de tipo clase 'A'
        .entrypoint                 // punto de entrada
        .maxstack 1
        newobj instance void 'A'::.ctor() // crear el objeto
        stloc 0                           // almacenarlo en la local 0
        ldloc 0                           // cargar el objeto
```

```
        call instance int32 'A'::'f'()    // llamar al método 'f' del objeto
        pop                         /* desapilar el valor devuelto
                                    porque no queremos hacer nada con él */
        ret                         // fin del programa
}
```

Para compilar y ejecutar el programa con DotGNU:

```
ilasm.pnet -e Hola.il
ilrun Hola.exe
```

Para hacerlo con Mono:

```
ilasm2 /exe /output:Hola.exe Hola.il
mono Hola.exe
```