Seminararbeit aus dem Seminar “Inside .NET and Java”

The Common Intermediate Language (CIL)

Wolfgang Fischereder, 9801695

“All roads lead to Rome”
(proverb)

“All languages lead to Intermediate Language”
(.NET-Framework and CLR)

Vorbemerkung

Bei allen Beispielen in Hochsprache handelt es sich – wenn nicht ausdrücklich anders angegeben – um Programmcode in der Sprache C#.
Alle Codesequenzen (in Hochsprache und in CIL) sowie die Stack-Transition-Diagramme sind in der Schriftart Book Antiqua gehalten.
Zur einfacheren Lesbarkeit des CIL-Codebeispiele wurde der *Debugmodus* des Compilers verwendet. Im *Releasemodus* werden die Namen der Variablen und Methodenargumente im CIL-Code durch interne Bezeichner ersetzt, was die Lesbarkeit herabsetzt und das Verständnis erschwert.

Einführung – Was ist die Common Intermediate Language?

Die *Common Intermediate Language* (dt. *gemeinsame Zwischensprache*) ist als Teil des Standards ECMA-335 spezifiziert.

Der Standard ECMA-335 definiert die ...
Common Language Infrastructure (CLI) *in which applications written in multiple high-level languages may be executed in different system environments without the need to rewrite the applications to take into consideration the unique characteristics of those environments*

(ECMA-335, CLI Partition I: Concepts and Architecture, p. 1)

Die **Common Language Runtime** (CLR) ist die Implementierung der CLI von Microsoft.
Die **Common Intermediate Language** (Akronym CIL) ist die einheitliche (*gemeinsame*) Zwischensprache (*Intermediate Language*), die die Common Language Runtime versteht. Sie

wird manchmal auch als MSIL (*Microsoft* Intermediate Language) oder einfach nur als IL – Intermediate Language – bezeichnet.
Eine ausführliche Spezifikation der Common Intermediate Language ist in dem Spezifikationsdokument ECMA-335, CLI Partition III: CIL Instruction Set, vorzufinden. Dieses Dokument ist unter der Internetadresse http://www.ecma-international.org frei zugänglich [ECMA02].

Die Common Intermediate Language besteht aus einem Satz grundlegender Instruktionen (***Basisinstruktionssatz***) und Instruktionen, die Features der Objektorientiertheit von Programmiersprachen unterstützen (***Object Model*** Instructions).

CIL-Instruktionen sind maschinencodeähnliche Instruktionen für die im Standard ECMA-335 spezifizierte virtuelle Ausführungsumgebung der CLR.

Jede Programmiersprache, deren Programme in einer CLR – Umgebung ausgeführt werden sollen, muss daher in CIL-Code übersetzt werden können.

Einen wesentlichen Vorteil bringt diese Designentscheidung für eine einheitliche Zwischensprache mit sich: Für *n* Programmiersprachen muss es auch nur *n* Compiler geben, die jeweils eine Sprache in semantisch äquivalenten CIL-Code übersetzen.

Aus diesem Grund ist es für den Frontend-Compilerwriter auch wichtig, die semantischen Eigenschaften der CIL-Instruktionen zu kennen. Denn ein Hochsprachencompiler für eine CLI (wie die CLR) muss die Semantik der Programmkonstrukte der Hochsprache exakt in CIL-Code nachmodellieren.

Die Aufgabe für den JIT- (***J****ust*-***I****n*-***T****ime*) Compiler für eine konkrete Zielplattform ist es nunmehr, schon vorhandenen CIL-Code in semantisch äquivalenten native Code für die Zielmaschine zu transformieren (bei *m* Zielplattformen muss es auch nur *m* JIT-Compiler geben!), um ein Programm laufen zu lassen.

Um den Sachverhalt auf den Punkt zu bringen, kann das Sprichwort „Alle Wege führen nach Rom“ in „All languages lead to Intermediate Language“ (alle [Hochprogrammier]sprachen führen zur Zwischensprache) umgelegt werden.

Virtual Execution System (VES)

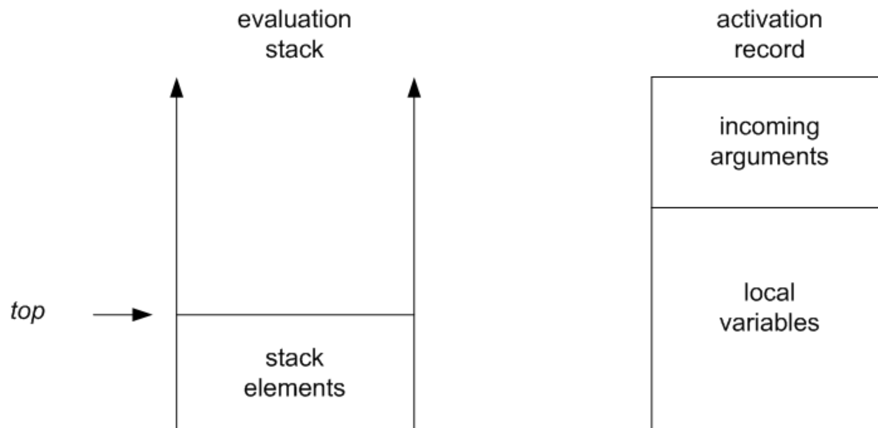
Um den Instruktionssatz der Common Intermediate Language und die Semantik der Instruktionen zu verstehen, ist es notwendig, die dem CIL-Instruktionssatz zugrunde gelegten Annahmen über die virtuelle Execution Engine, das im Standard ECMA-335 spezifizierte Virtual Execution System (VES), zu kennen.

Das Virtual Execution System ist eine Stackmaschine.

Der Zustand dieser *execution engine* bezüglich eines Methodenaufrufs ist durch zwei Teile gekennzeichnet:

1. Activation Record (*Aktivierungssatz*)
2. Evaluation Stack

Der Zustand des VES kann somit durch den aktuellen Zustand von Activation Record und Evaluation Stack beschrieben werden. Im Diagramm kann das ganze wie folgt dargestellt werden:

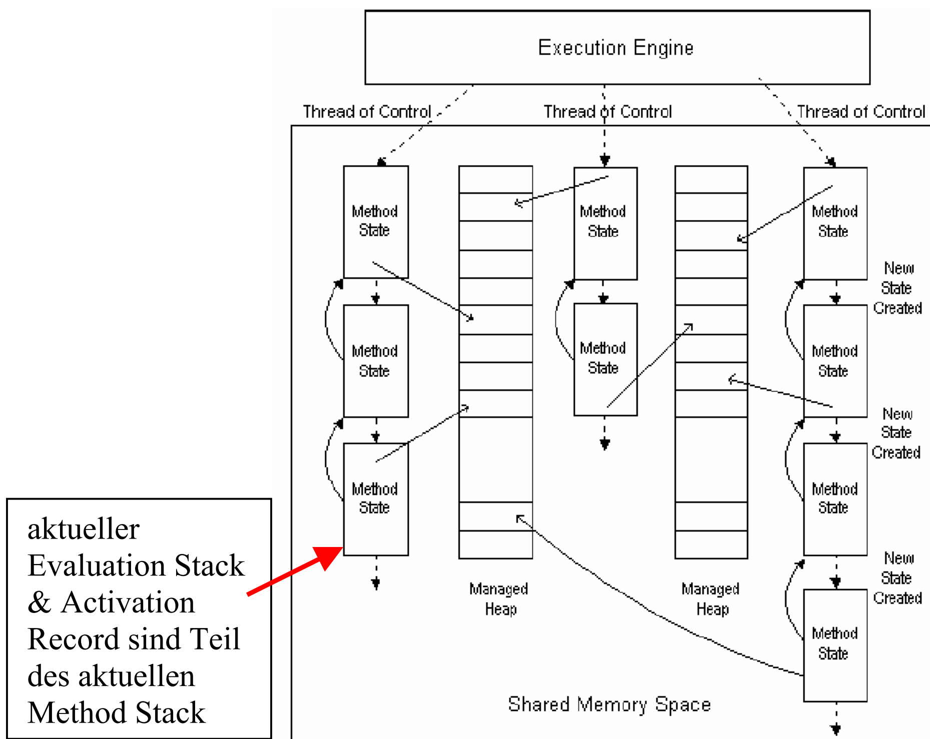


Der *Evaluation Stack* ist ein – theoretisch – unbegrenzter Stapelspeicher (Stack). Elemente können auf ihn *gepusht* oder von ihm *gepoppt* werden. Der Stackgröße nimmt bei einer Push-Operation zu und nimmt bei einer Pop-Operation ab. Es können auch Operationen auf Elemente am Stack durchgeführt werden.
Ein Zeiger *top* zeigt jeweils auf das oberste Element am Stack. *top* ist programmatisch (d. h. über CIL-Code) nicht zugreifbar. Der aktuelle Wert von *top* ist nur innerhalb des VES sichtbar und wird bei der Abarbeitung von CIL-Code implizit verwendet und verändert.

Ein *Activation Record* wird beim Aufruf einer Methode (mittels eines Befehls der Art `call`) allokiert. Ein Activation Record enthält jeweils die der Methode übergebenen Argumente, sowie die lokal in einer Methode sichtbaren Variablen. Argumente wie lokale Variable einer Methode sind von null aufwärts, jeweils getrennt nummeriert. Es kann in einer Methode (und somit in ihrem zugehörigen Aktivierungssatz) null oder mehrere Argumente sowie null oder mehrere lokale Variable geben.

Bei der Nummerierung der Argumente und lokalen Variablen handelt es sich um eine logische Nummerierung (beginnend bei 0). Es übt auf die Nummerierung der lokalen Variablen und der Argumente einer Methode keinen Einfluss aus, ob sich bei diesen ausschließlich um Werte vom Typ 32-Bit-Integer oder um Werttypen (siehe unten) von einer Größe von mehreren hundert Bytes handelt.

ECMA-335, CLI Partition I: Concepts and Architecture, p. 80:
... The CLI manages multiple concurrent threads of control (not necessarily the same as the threads provided by a host operating system), multiple managed heaps, and a shared memory address space.
A thread of control can be thought of, somewhat simplistically, as a singly linked list of *method states*, where a new state is created and linked back to the current state by a method call instruction – the traditional model of a stack-based calling sequence. ….)



Eine ausführliche Spezifikation, Darstellung und Diskussion des VES findet sich im Spezifikationsdokument ECMA-335, CLI Partition I, Kapitel 12 [ECMA02].

Basisinstruktionssatz 1

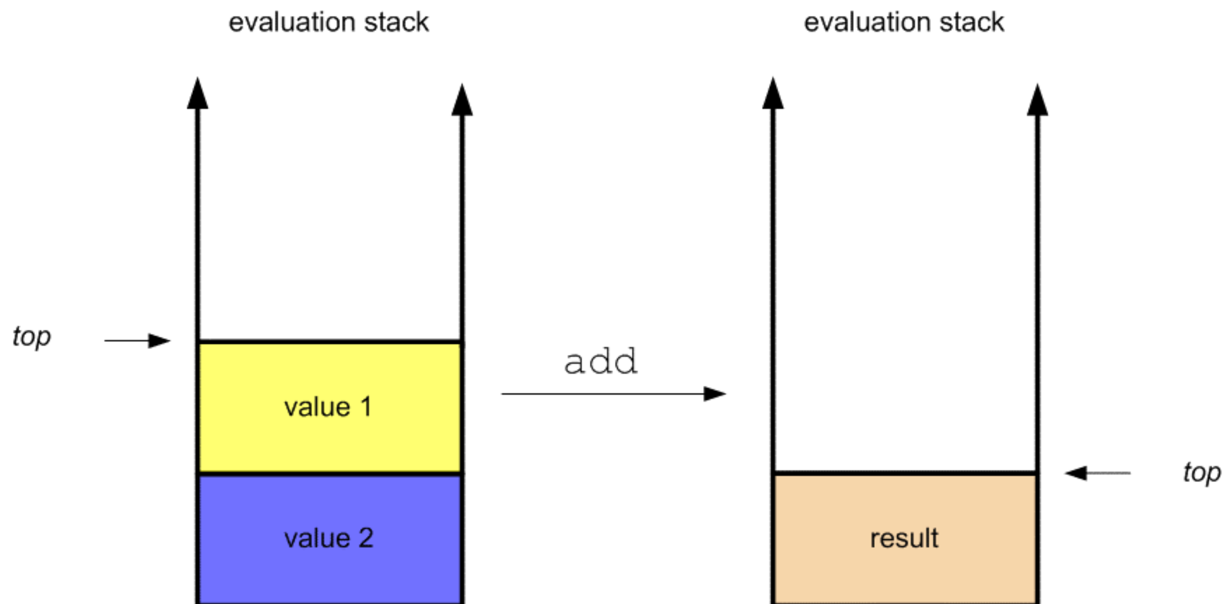
Die Instruktionen der Common Intermediate Language können nun danach eingeteilt werden, welche Wirkung die einzelnen Instruktionen auf den Stack zeigen.

Dieses Kriterium lässt eine Einteilung des CIL-Instruktionssatzes in drei Klassen zu:

Instruktionen, die ...

1. Operationen auf am Stack vorhandene Werte durchführen:

Zu dieser Kategorie gehört etwa der Befehl `add`. Dieser Befehl der CIL addiert die beiden obersten am Stack liegenden Werte und ersetzt sie durch das Ergebnis der Addition.



2. einen Wert auf den Stack *pushen* (`load`-Befehle)

3. einen Wert vom Stack *poppen* und an einer bestimmten Stelle speichern (`store`-Befehle)

Bestimmte Befehle lassen sich keiner dieser Kategorien zuordnen. Es sind dies spezielle Befehle wie **pop** (einfaches Entfernen vom Stack), **dup** (Duplizieren des obersten Elements am Stack), sowie **nop** (die *leere* Operation, bewirkt keine Veränderung am Stack).

Eine übliche Notation, die Befehle eines Instruktionssatzes einer Stackmaschine zu beschreiben, ist das *stack transition diagram*. Im folgenden ist das stack transition diagram für den `add` Befehl gegeben.

…, value 1, value 2 → …, result

Ein weiteres Charakteristikum, um eine Instruktion der CIL zu beschreiben, ist die Veränderung der Anzahl der Elemente am Stack, die aus der Ausführung einer Instruktion resultiert. Bei dieser Zahl handelt es sich um das sogenannte Stack-Δ („Stack-Delta").

Stack-Δ =

Differenz der Anzahl der Elemente
auf dem evaluation stack
nach Ausführung eines Befehls i und
vor Ausführung eines Befehls i.

Für den Additionsbefehl add ist der Wert des Stack-Δ gerade –1.

Beispiel einer einfachen Sequenz von Instruktionen

Gegeben sei eine kurze statische Methode TestMethod:

```
public static void TestMethod (int a, int b) // arguments
{
        int c; int d; int e; // locals
        c = a + b;
        d = 10;
        e = c + d;
}
```

Dieses Stück Hochsprache wird vom C#-Compiler in CIL-Code übersetzt:

```
.method public hidebysig static void  TestMethod(int32 a,int32 b) cil managed
{
  // Code size       12 (0xc)
  .maxstack  2
  .locals init ([0] int32 c,
           [1] int32 d,
           [2] int32 e)
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  add
  IL_0003:  stloc.0
  IL_0004:  ldc.i4.s   10
  IL_0006:  stloc.1
  IL_0007:  ldloc.0
  IL_0008:  ldloc.1
  IL_0009:  add
  IL_000a:  stloc.2
  IL_000b:  ret
} // end of method Class1::TestMethod
```

ldarg *num* — load argument no. *num* onto the stack
… → …, value

ldloc *indx* — load local variable no. indx onto the stack
… → …, value

ldc *num* — load numeric constant
… → …, num

stloc *indx* pop value from stack to local variable
…, value → …

starg *num* store a value in an argument slot
…, value → …

Die CIL-Codesequenz für die Methode TestMethod verwendet zumindest je einen Befehl aus jeder der oben genannten drei Kategorien und zeigt, wie auf die lokalen Variablen und Argumente der Methode schreibend und lesend zugegriffen wird.

An die Methode werden zwei Argumente vom Typ Integer (à 4 Byte / 32 Bit), a und b, übergeben. Dies spiegelt sich direkt im CIL-Code wider. Im Methodenkopf von TestMethod werden 2 Argumentslots vom Typ int32 in der entsprechenden Reihenfolge reserviert. Die Nummerierung der Argumente ist explizit nicht sichtbar, die Argumente sind von 0 bis n – 1 – bei n Methodenargumenten – durchnummeriert.
Ebenso werden für die drei lokalen Integervariablen c, d und e Slots angelegt, durchnummeriert in der Reihenfolge ihres Auftretens im Code der Methode.

Abb.: Die Slots für die Argumente und lokalen Variablen der Methode TestMethod im Activation Record.

Wird jetzt im CIL-Code auf ein Argument oder eine lokale Variable schreibend (store) oder lesend (load) zugegriffen, so wird im Befehl das entsprechende Methodenargument bzw. die lokale Variable immer mittels der zugehörigen Nummer angesprochen.

Um auf ein Argument einer Methode zuzugreifen, sind im CIL-Instruktionssatz die Befehle ldarg (lesend) und starg (schreibend) vorhanden. Mit dem Befehl ldarg.0 etwa wird der Wert von Argument 0 – a im Beispiel – auf den Stack geladen. Mit starg.0 könnte man einen Wert, der auf dem Stack an oberster Stelle liegt, in den Argumentslot von a speichern.

Der Zugriff auf lokale Variablen wird genauso gehandhabt wie der Zugriff auf die Methodenargumente. Mittels ldloc wird der Inhalt einer lokalen Variablen auf den Stack gepusht (lesender Zugriff), mittels stloc wird ein Wert vom Stack gepusht und einer lokalen Variablen *zugewiesen* (schreibender Zugriff).

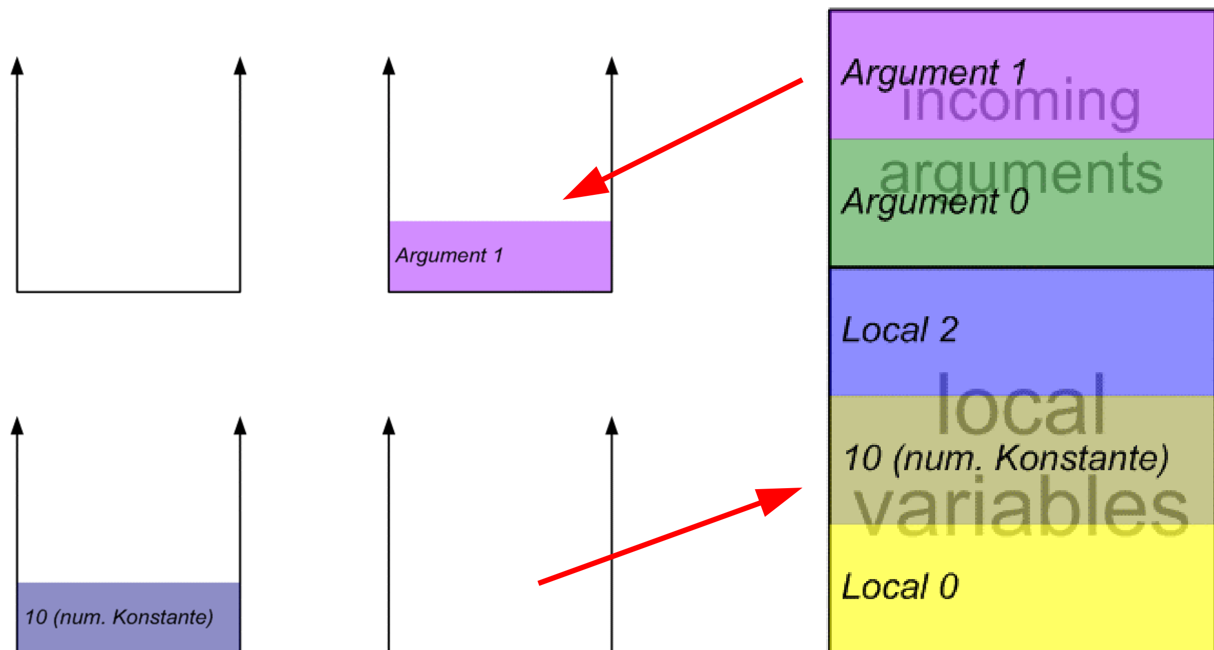


Abb.: Wirkung von Befehlen wie ldarg.1 und stloc.1 auf Evaluation Stack und Activation Record

In TestMethod wird etwa der lokalen Variable e der Wert des Ausdrucks c + d zugewiesen.

e = c + d;

Diese Zuweisungsoperation ist so implementiert:

```
IL_0007:  ldloc.0
IL_0008:  ldloc.1
IL_0009:  add
IL_000a:  stloc.2
```

Zuerst müssen die Werte von c und d auf den Stack geladen werden. Dies geschieht durch die beiden Anweisungen ldloc.0 und ldloc.1. ldloc.0 lädt die lokale Variable mit der Nummer 0, also c, und ldloc.1 lädt die lokale Variable Nummer 1, also d. Dann muss die Summe dieser beiden Werte gebildet werden. Dies wird durch den add Befehl realisiert. Die beiden

Argumente für die add-Operation liegen mittlerweile schon auf dem Stack, nach Ausführung von add, bleibt als Resultat die Summe von c und d auf dem Stack zurück. Diese Summe wird nun mittels stloc.2 in die lokale Variable e (e ist lokale Variable mit der Nummer 2) gespeichert, also e zugewiesen.

Neben dem Laden und Speichern von Argumenten und lokalen Variablen, muss der CIL-Instruktionssatz auch die Möglichkeit vorsehen, *festcodierte* Werte, also numerische Konstanten, auf den Stack zu laden.

Um eine numerische Konstante auf den Stack zu laden, gibt es in der IL die Instruktion ldc. Die Zuweisung einer Integerkonstante 10 auf d

```
d = 10;
```

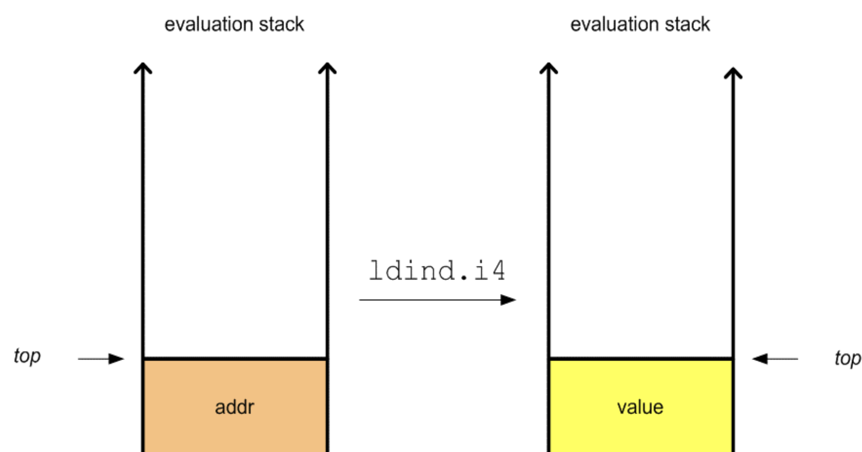
wird in CIL wie folgt realisiert:

```
IL_0004:  ldc.i4.s   10
IL_0006:  stloc.1
```

Indirektion bei Load und Store

Weiters sieht der CIL-Instruktionssatz auch die Möglichkeit der Indirektion beim Laden auf den und beim Speichern von Werten vom Stack vor.
Indirektion (oder *Dereferenzierung*) bedeutet, dass die Adresse der Variable, deren Inhalt geladen oder geschrieben werden soll, zur Verfügung steht. Im VES bedeutet das, dass eine Zieladresse für die Lade- bzw. Speicheroperation auf dem Stack liegt.

Zum Zugriff mittels Indirektion gibt es in der Common Intermediate Language zwei Instruktionen: ldind (lesend) und stind (schreibend):



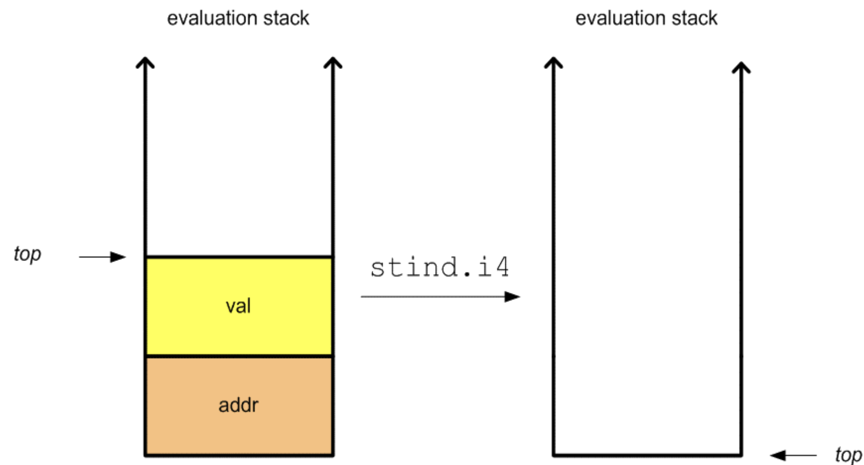


Abb.: Stack-Transition-Diagramme für ldind und stind: der Suffix (.i4) gibt den Typ von value an

Das „klassische" Beispiel der Indirektion findet sich in C/C++ (*Managed C++*)

```
int x = 99;
int* px = &x;

*px = 40;
```

der entsprechende CIL-Code:

```
.maxstack  2
  .locals ([0] int32* px,
           [1] int32 x)
  IL_0000:  ldc.i4.s   99
  IL_0002:  stloc.1
  IL_0003:  ldloca.s   x        // dieser Befehl ermittelt die Adresse
                                // der Variable x, = transient pointer
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  ldc.i4.s   40
  IL_0009:  stind.i4
  IL_000a:  ldc.i4.0
  IL_000b:  ret
```

In diesem kurzen Codestück wird der Zeigervariablen px die Adresse der Variable x (vom Typ Integer) zugewiesen. Adressen werden mittels spezieller Ladebefehle ermittelt: Die Adresse der lokalen Variable x wird mittels der IL-Anweisung ldloca ermittelt – man beachte die Schreibweise ldloc**a** statt ldloc! – und der Zeigervariable zugewiesen. ldloca hinterlässt die Adresse von x auf dem Stack. Eine solche Adresse wird als ein *transient pointer* bezeichnet, da sie nur vorübergehend am Stack liegt. Die Adresse wird in die Zeigervariable px gespeichert (stloc.0).
Nun soll x der Wert 40 über Dereferenzierung der Zeigervariablen px, die auf x zeigt, zugewiesen werden.

Dazu wird der transient pointer, der in px gespeichert ist, auf den Stack geladen, dann wird der zu speichernde Wert (40) auf den Stack geladen, und mittels stind wird der Wert 40 an der entsprechenden Adresse (in x) gespeichert. Der Suffix (.i4) gibt den Typ des Wertes an, der gespeichert wird (hier: ein int32 – 4 Byte Integer).

Weitere Befehle zur Adressberechnung:

ldarga wird zur Parameterübergabe an Methoden mit ref- oder out-Parametern verwendet (s. u.: Aufruf der Methode *DemoReference*).

ldflda / ldsflda / ldelema:
Diese drei Ladeanweisungen treten im Zusammenhang mit Werttypen (s. u.) auf (etwa wenn eine Struktur (aus C#) ein Feld eines strukturierten Datentyps ist oder bei Arrays von Strukturen).

Datentypen

Die CLR kennt mehrere Datentypen:

- bool
- char, string
- objectref, typedref
- int8, int16, int32, int64 (*unsigned*: unsigned int8, …)
- float32, float64
- native int, native unsigned int

Auf dem Stack des VES sind aber nur folgende Datentypen zulässig:

- int32, int64, native int (i – native size integers), ein interner Typ F (float, native size floating point numbers)
- Object references (o)
- pointer types (native unsigned integers, &)

Dies bedeutet, dass beim Laden / Speichern auf den / vom Stack – falls notwendig – entsprechende implizite Konvertierungen eines Typs in einen anderen Typ vorgenommen werden müssen (z. B.: int8 und bool sind int32 am Stack: *true* ist 1, *false* ist 0, s. Boolsche Ausdrücke)

Object Model

Die bis jetzt vorgestellten Instruktionen der CIL helfen einen Teil der grundlegenden Funktionen einer Programmiersprache zu modellieren. Da heutige High-Level Programmiersprachen aber meist mit Features der Objektorientiertheit versehen sind, bietet die CIL neben dem Basisinstruktionssatz auch noch einen Satz von Instruktionen an, die die Modellierung der Semantik von modernen High-Level Programmiersprachen wesentlich erleichtern.

Klassen und Objekte

Im folgenden ist eine Klasse *TestClass* mit überladenem Konstruktor in C# gegeben (der überladene Konstruktor benötigt zwei Integerwerte als Argumente):

```
public class TestClass
{
  int a, b;
  public TestClass(int c, int d)
  {
        a = c;
        b = d;
  }
}
```

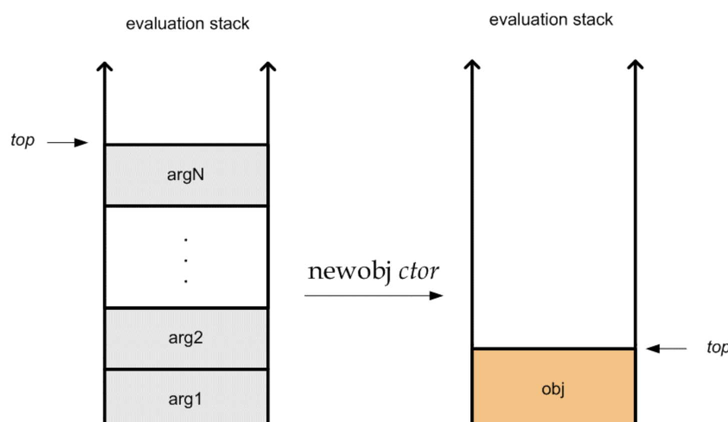
Im Hauptprogramm wird ein Objekt der Klasse *TestClass* angelegt:

```
TestClass TestObject = new TestClass(10, 40);
```

Der CIL-Code dieses kurzen Programms:

```
.maxstack  3
  .locals init ([0] class DemoApplication5.Class1/TestClass TestObject)
  IL_0000:  ldc.i4.s   10
  IL_0002:  ldc.i4.s   40
  IL_0004:  newobj     instance void DemoApplication5.Class1/TestClass::.ctor(int32,
                                                    int32)
  IL_0009:  stloc.0
  IL_000a:  ret
```

newobj *ctor*   create a new object
..., arg1, ... argN → ..., obj
Allokiert ein uninitialisiertes Objekt und ruft *ctor* (Konstruktor) auf. *ctor* ist ein Metadatatoken.



Die vom Konstruktor benötigten Argumente (arg1, ..., argN) werden auf den Stack geladen. Dann kann der Konstruktor aufgerufen werden (newobj *ctor*) und ein Objekt wird allokiert. Zurück bleibt die eine Referenz (obj) auf das soeben angelegte Objekt, das sich – wie alle Objekte – physisch in einem dafür vorgesehenen Speicherbereich, dem sogenannten *Heap* befindet. Die auf dem Stack zurückbleibende Referenz wird einer Referenzvariable vom Typ *TestClass* zugewiesen (stloc.0).

Definition der Klasse *DemoClass*:

```
public class DemoClass
{
        public static int StatDemoVariable;
        public int ObjDemoVariable;
}
```

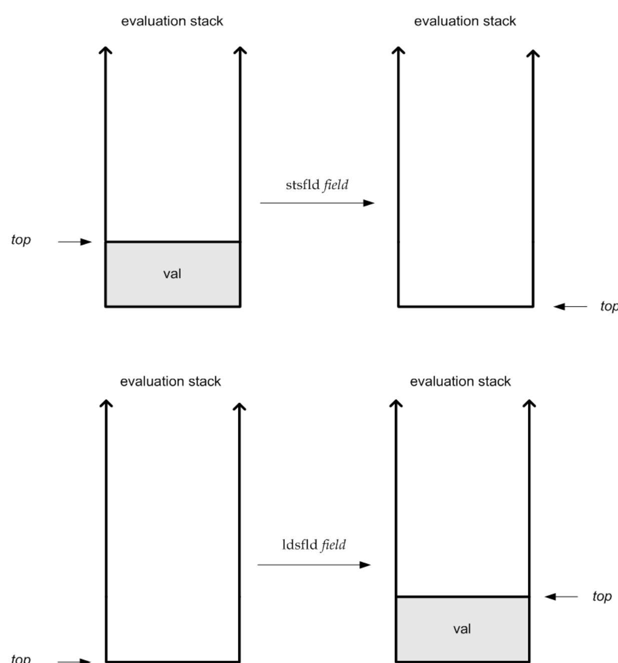
Member einer Klasse können statisch sein oder zu einer Instanz gehören. Man spricht von statischen Klassenfeldern und -methoden und von Objektfeldern und -methoden. Die Klasse DemoClass besitzt ein Klassenfeld StatDemoVariable und ein Objektfeld ObjDemoVariable.

Um auf eine statisches Feld zuzugreifen, gibt es in der CIL die folgenden Instruktionen:

stsfld *field*   store into a static field of an object
…, *value* → …,
Speichert den Wert *value* im statischen Feld *field* einer Klasse

ldsfld *field*   load static field of a class
..., → ..., *value*
Pusht den Wert eines statischen Felds *field* einer Klasse auf den Stack

field ist jeweils ein *Metadatatoken*, das sich auf ein statisches Feld einer Klasse bezieht

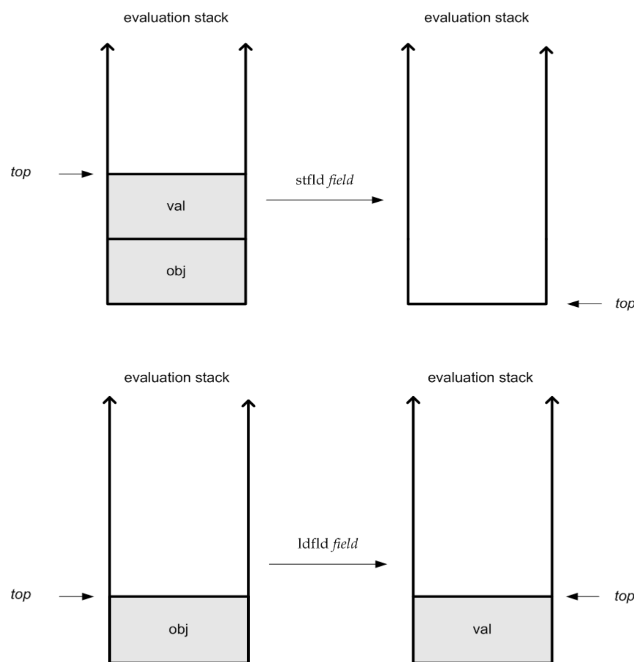


Für den Zugriff auf Objektfelder stehen folgende Instruktionen zur Verfügung:

ldfld *field* load field of an object
…, *obj* → …, *value*
Pusht den Wert von field von *obj*, oder des value type *obj*, auf den Stack

stfld *field* store into a field of an object
…, *obj*, *value* → …,
Speichert den Wert *value* im Feld *field* des Objektes *obj*

field ist jeweils ein *Metadatatoken*, das sich auf ein Feld eines Objekts bezieht



Beispiel:

Zuweisung (= schreibender Zugriff) auf Klassen- und Objektfelder

DemoClass DemoObject;
DemoObject = new DemoClass();

DemoClass.StatDemoVariable = 7;
DemoObject.ObjDemoVariable = 8;

CIL-Code:

```
.maxstack  2
  .locals init ([0] class DemoApplication2.Class1/DemoClass DemoObject)
  IL_0000:  newobj     instance void DemoApplication2.Class1/DemoClass::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldc.i4.7
```

```
 IL_0007:  stsfld     int32 DemoApplication2.Class1/DemoClass::StatDemoVariable
 IL_000c:  ldloc.0   // Lade Referenz auf DemoClass auf den Stack
 IL_000d:  ldc.i4.8
 IL_000e:  stfld      int32 DemoApplication2.Class1/DemoClass::ObjDemoVariable
 IL_0013:  ret
```

Lesender Zugriff auf Klassen- und Objektfelder:

```
int x;
int y;
DemoClass DemoObject;
DemoObject = new DemoClass();

x = 14 + DemoObject.ObjDemoVariable;
y = 15 + DemoClass.StatDemoVariable;
```

CIL-Code:

```
.maxstack  2
  .locals init ([0] int32 x,
           [1] int32 y,
           [2] class DemoApplication2.Class1/DemoClass DemoObject)
  IL_0000:  newobj     instance void DemoApplication2.Class1/DemoClass::.ctor()
  IL_0005:  stloc.2
  IL_0006:  ldc.i4.s   14
  IL_0008:  ldloc.2   // Lade Referenz auf DemoClass auf den Stack
  IL_0009:  ldfld      int32 DemoApplication2.Class1/DemoClass::ObjDemoVariable
  IL_000e:  add
  IL_000f:  stloc.0
  IL_0010:  ldc.i4.s   15
  IL_0012:  ldsfld     int32 DemoApplication2.Class1/DemoClass::StatDemoVariable
  IL_0017:  add
  IL_0018:  stloc.1
  IL_0019:  ret
```

Die Instruktionen zum Zugriff auf Klassen- und Objektfelder (ldsfld / stsfld / ldfld / stfld) verhalten sich grundsätzlich genauso wie die einfachen Load- und Store-Befehle. Im Unterschied zu diesen benötigen die Lade- und Speicheranweisungen für Felder aber ein *Metadatatoken* im Instruktionscode, das sich auf die Klassen- und Objektfelder, auf die zugegriffen werden soll, bezieht.

Der markanteste Unterschied zwischen Klassen- und Objektfeldern beim Zugriff ist, dass sich ein Zugriff auf ein Objektfeld auf eine bestimmte Instanz einer Klasse (Objekt) bezieht, daher muss auch die Referenz auf das Objekt (ein Verweis auf das Objekt am Heap) beim Zugriff auf ein Objektfeld zur Verfügung gestellt werden, was beim Zugriff auf die statischen Felder einer Klasse entfällt.

Methodenaufruf

Ein Methodenaufruf (*method call*) wird in CIL mittels einem der folgenden zwei Befehle realisiert:

1.
call *method* call a method
..., arg1, arg2, ..., argn → ..., retVal (not always returned)
Wird zum Aufruf von Klassenmethoden verwendet (genauer: bei früher Bindung – *early binding*). *method* ist ein Metadatatoken, das Name und Signatur der aufzurufenden Methode beschreibt.

2.
callvirt *method* call a method associated, at runtime, with an object
..., obj, arg1, arg2, ..., argN → ..., returnValue (not always returned)
Wird zum Aufruf einer Objektmethode verwendet (genauer: bei später Bindung – *late binding*). *method* ist ein Metadatatoken, das Name, Klasse und Signatur der aufzurufenden Methode beschreibt.

Weiters gibt es noch die Möglichkeit eine Methode via eines Methodenzeigers aufzurufen: Dazu wird die Instruktion calli benutzt. Diese Art von Methodenaufruf ist jedoch nicht typsicher, die typsichere Version eines Methodenaufrufs via Methodenzeiger in der CLR ist ein Delegate (s. u.).

calli *callsitedescr* indirect method call
..., arg1, arg2, ..., argn, ftn → ..., retVal (not always returned)

Beispiel:

Aufruf einer statischen Methode *DemoReference*

```
int x = 10;
int y = 30;

DemoReference(ref x, ref y);
```

CIL-Code:

```
.maxstack  2
  .locals init ([0] int32 x,
           [1] int32 y)
  IL_0000:  ldc.i4.s   10
  IL_0002:  stloc.0
  IL_0003:  ldc.i4.s   30
```

```
  IL_0005:  stloc.1
  IL_0006:  ldloca.s   x
  IL_0008:  ldloca.s   y
  IL_000a:  call       void DemoApplication2.Class1::DemoReference(int32&, int32&)
  IL_000f:  ret
```

Der statischen Methode *DemoReference* werden beim Aufruf die Referenzen (Adressen) zweier Integervariablen übergeben. Diese Argumente müssen wie alle Argumente vor dem Methodenaufruf der Reihe nach auf den Stack geladen werden. Die Adressen von x und y, lokalen Integervariablen, werden in den Zeilen IL_0006 und IL_0008 ermittelt (ldloca.s) und auf den Stack gelegt. Jetzt kann die Methode aufgerufen werden (Zeile IL_000a).

Bemerkung: Da die Adressen von x und y übergeben wurden, werden beim Speichern in die Argumentslots von *DemoReference* die Änderungen der Variableninhalte von x und y auch außerhalb der Methode sichtbar – etwa, wenn die Methode wie folgt aussieht:

```
public static void DemoReference(ref int a, ref int b)
{
        a = 90;
        b = 177;
}
```

CIL-Code von DemoReference:

```
.method public hidebysig static void  DemoReference(int32& a,
                                                    int32& b) cil managed
{
  // Code size       12 (0xc)
  .maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  ldc.i4.s   90
  IL_0003:  stind.i4
  IL_0004:  ldarg.1
  IL_0005:  ldc.i4     0xb1
  IL_000a:  stind.i4
  IL_000b:  ret
} // end of method Class1::DemoReference
```

Aufruf einer Objektmethode

Beim Aufruf einer nichtstatischen Methode einer Klasseninstanz, d. h. einer Objektmethode, muss zusätzlich zu den Argumenten der Methode, auch noch der Verweis auf das Objekt, dessen Methode aufgerufen werden soll, auf den Stack gepusht werden.

Beispiel:

Definition der Klasse *Time*

```
public class Time
{
	public int TimeInSeconds(int hours, int minutes, int seconds)
	{
		return (hours * 3600 + minutes * 60 + seconds);
	}
}
```

Hauptprogramm:

```
Time FirstTime = new Time();
int hours = 5;
int minutes = 47;
int seconds = 7;
FirstTime.TimeInSeconds(hours, minutes, seconds);
```

CIL-Code:

```
.maxstack  4
  .locals init ([0] class DemoApplication2.Class1/Time FirstTime,
           [1] int32 hours,
           [2] int32 minutes,
           [3] int32 seconds)
  IL_0000:  newobj     instance void DemoApplication2.Class1/Time::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldc.i4.5
  IL_0007:  stloc.1
  IL_0008:  ldc.i4.s   47
  IL_000a:  stloc.2
  IL_000b:  ldc.i4.7
  IL_000c:  stloc.3
  IL_000d:  ldloc.0
  IL_000e:  ldloc.1
  IL_000f:  ldloc.2
  IL_0010:  ldloc.3
  IL_0011:  callvirt   instance int32
DemoApplication2.Class1/Time::TimeInSeconds(int32,
                                                               int32,
                                                               int32)
  IL_0016:  pop
  IL_0017:  ret
```

Das kurze Hauptprogramm besitzt vier lokale Variablen. Die erste lokale Variable FirstTime ist vom Referenztyp Time. Ein Objekt der Klasse Time wird auf dem Heap angelegt (IL_0000: newobj instance void DemoApplication2.Class1/Time::.ctor()), die Objektreferenz wird FirstTime zugewiesen (IL_0005: stloc.0).

Beim Aufruf der Objektmethode TimeInSeconds der Klasse Time, wird nun als Argument 0 die Referenz auf das Objekt FirstTime übergeben (ldloc.0).

Das Argument 0 erfüllt in Instanzmethoden implizit die Funktion des *this*-Zeigers. Der *this*-Zeiger eines Objekts ist ein Zeiger, der auf das Objekt selbst verweist (Selbstreferenz).
In statischen Methoden ist kein this-Zeiger vorhanden, da statische Klassenmember nicht mit einem konkreten Objekt assoziiert sind, sondern auf Klassenebene gelten.

Die Instanzmethode *TimeInSeconds* der Klasse Time gibt die als Stunden, Minuten und Sekunden übergebene Zeitspanne in Sekunden zurück. Im CIL-Code von TimeInSeconds ist zu sehen, dass die Methodenargumente (int hours, int minutes, int seconds) im Methodenrumpf eine Nummerierung von 1 aufwärts besitzen.

CIL-Code *TimeInSeconds*

```
.method public hidebysig instance int32  TimeInSeconds(int32 hours,
                                        int32 minutes,
                                        int32 seconds) cil managed
{
  // Code size       19 (0x13)
  .maxstack  3
  .locals init ([0] int32 CS$00000003$00000000)
  IL_0000:  ldarg.1 // lädt hours auf den Stack (Argumentslot 1)
  IL_0001:  ldc.i4     0xe10
  IL_0006:  mul
  IL_0007:  ldarg.2 // lädt minutes auf den Stack (Argumentslot 2)
  IL_0008:  ldc.i4.s   60
  IL_000a:  mul
  IL_000b:  add
  IL_000c:  ldarg.3 // lädt seconds auf den Stack (Argumentslot 3)
  IL_000d:  add
  IL_000e:  stloc.0
  IL_000f:  br.s       IL_0011
  IL_0011:  ldloc.0
  IL_0012:  ret
} // end of method Time::TimeInSeconds
```

Über den Argumentslot 0 (ldarg.0) ließe sich das Objekt, auf das *FirstTime* verweist, ansprechen (*this*-Zeiger). In folgendem Codebeispiel ist dies sogar notwendig:

```
public class Time
{
        private int h;
```

```
        …
        …
        public int TimeInSeconds(int hours)
        {
                this.h = 33;
                return (hours * 3600);
        }
}
```

Der Klasse Time wird ein nichtstatisches Member private int h hinzugefügt, die Methode *TimeInSeconds* einmal überladen. In *TimeInSeconds* wird – explizit, obwohl nicht notwendig – über *this* auf h zugegriffen.

In der CIL äußerst sich das dann folgendermaßen:

```
.method public hidebysig instance int32  TimeInSeconds(int32 hours) cil managed
{
  // Code size       20 (0x14)
  .maxstack  2
  .locals init ([0] int32 CS$00000003$00000000)
  IL_0000:  ldarg.0 // Argument 0 = this-pointer
  IL_0001:  ldc.i4.s   33 // value to be assigned to field h of object
  IL_0003:  stfld      int32 DemoApplication2.Class1/Time::h
  IL_0008:  ldarg.1
  IL_0009:  ldc.i4     0xe10
  IL_000e:  mul
  IL_000f:  stloc.0
  IL_0010:  br.s       IL_0012
  IL_0012:  ldloc.0
  IL_0013:  ret
} // end of method Time::TimeInSeconds
```

Mit stfld *field* kann in das Feld eines Objektes gespeichert werden (Zuweisung). Dazu benötigt stfld eine Objektreferenz (und den zu speichernden Wert) am Stack. Der Verweis auf das entsprechende Objekt ist in der überladenen TimeInSeconds-Methode der *this*-Zeiger. Er wird mit ldarg.0 auf den Stack geladen (IL_0000).

Arrays

Arrays sind Objekte am *Heap*.
Ein eindimensionales Array wird mittels folgendem Befehl angelegt:

newarray *etype*   create a zero-based, one-dimensional array
..., numElements → ..., obj

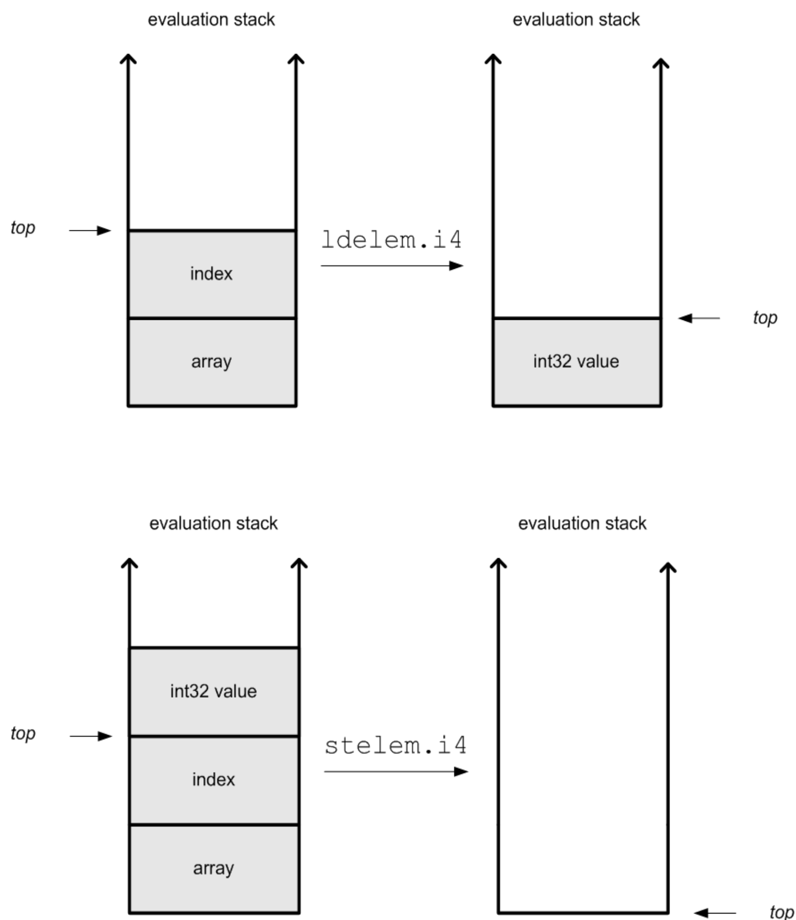
Create a new array with elements of type *etype* (Metadatatoken).

Die Anzahl der Elemente des Arrays (numElements) muss bei Aufruf von newarray bereits auf dem Stack liegen. Der Typ der Elemente des Array wird durch das Metadatentoken *etype* beschrieben.

Die Befehle ldelem und stelem dienen zum Zugriff auf Elemente im Array.

ldelem.* load an element of an array
..., array, index → ..., value
ldelem lädt das Element (*value*) an der Stelle *index* des Arrays *array* (Referenz auf das Objekt am Heapspeicher) auf den Stack.

stelem.* store an element of an array
..., array, index, value → ...
stelem speichert einen Wert *value* (vom Stack) an die Stelle *index* des Arrays *array.*



Die Wildcard (*) zeigt den Typ des Wertes an, der geladen oder gespeichert wird.

Beispiel:

```
int x;
int [] IntegerArray = new int [3];
IntegerArray[1] = 4;
x = 99 + IntegerArray[1];
```

CIL-Code:

```
.maxstack  3
  .locals init ([0] int32 x,
           [1] int32[] IntegerArray)
  IL_0000:  ldc.i4.3
  IL_0001:  newarr     [mscorlib]System.Int32
  IL_0006:  stloc.1
  IL_0007:  ldloc.1
  IL_0008:  ldc.i4.1
  IL_0009:  ldc.i4.4
  IL_000a:  stelem.i4
  IL_000b:  ldc.i4.s   99
  IL_000d:  ldloc.1
  IL_000e:  ldc.i4.2
  IL_000f:  ldelem.i4
  IL_0010:  add
  IL_0011:  stloc.0
  IL_0012:  ret
```

Die Wildcard (*) in ldelem.* und stelem.* ist durch *i4* ersetzt worden. Das bedeutet, dass ein *signed 4 Byte Integer (32 Bit)* geladen oder gespeichert wird.

Beispiel:

Definition der Klasse Haus:

```
public class Haus
{
    public int Bewohner;
}
```

Programmcode:

```
Haus [] HausArray = new Haus [3];
HausArray[2] = new Haus();
HausArray[2].Bewohner = 999;
```

CIL-Code:

```
// Code size       29 (0x1d)
  .maxstack  3
  .locals init ([0] class DemoApplication2.Class1/Haus[] HausArray)
  IL_0000:  ldc.i4.3
  IL_0001:  newarr     DemoApplication2.Class1/Haus
  IL_0006:  stloc.0
  IL_0007:  ldloc.0
  IL_0008:  ldc.i4.2
  IL_0009:  newobj     instance void DemoApplication2.Class1/Haus::.ctor()
  IL_000e:  stelem.ref
  IL_000f:  ldloc.0
  IL_0010:  ldc.i4.2
  IL_0011:  ldelem.ref
  IL_0012:  ldc.i4     0x3e7
  IL_0017:  stfld      int32 DemoApplication2.Class1/Haus::Bewohner
  IL_001c:  ret
```

Hier wird die Wildcard (*) in ldelem / stelem durch ein *ref* ersetzt. Aus Klassendefinition und Programmcode der Hochsprache geht hervor, dass der Typ des Wertes, der auf den Stack geladen wird, eine Referenz auf ein Objekt ist, da auf ein Objektfeld zugegriffen wird. Die Elemente des Arrays *HausArray* sind Verweise auf Objekte der Klasse *Haus*.

In der CLR – Haussprache C# ist sind eindimensionale Arrays nur eine Form von Arrays. C# kennt auch mehrdimensionale Arrays. Es gibt „ausgefranste" (*jagged*) mehrdimensionale Arrays und „rechteckige" mehrdimensionale Arrays. Aus entsprechendem CIL-Code geht hervor, dass der Zugriff auf die rechteckigen, kompakten Arrays effizienter ist.

Werttypen – strukturierte Datentypen

Strukturierte Datentypen sind auf der Java VM (Java Virtual Machine) ausschließlich Referenztypen. Das heißt, Werte dieser strukturierten Datentypen liegen als Objekte auf dem Heap. Die Variablen solcher Typen enthalten nur Zeiger (Verweise) auf Objekte am Heap.

In der CLR hingegen ist es möglich, dass ein Wert eines strukturierten Datentyps direkt auf dem Stack liegt. In C# wird diese Möglichkeit für das Sprachelement der Strukturen (*structures*) ausgenutzt, die Werttypen darstellen und als ganzes am Stack liegen (Bemerkung: Enumerationen in C# sind ebenfalls Werttypen).

Beispiel:

```
struct Point {
  public int x, y;
}
```

Hauptprogramm:

Point MyPoint = new Point();

CIL-Code:

```
.maxstack  1
  .locals init ([0] valuetype ConsoleApplication2.Point MyPoint)
  IL_0000:  ldloca.s   MyPoint
  IL_0002:  initobj    ConsoleApplication2.Point
  IL_0008:  ret
```

initobj *classTok* initialize a value type
..., addrOfValObj → ...,
Initialisiert alle Felder entsprechend auf *null* oder *0. classTok* repräsentiert einen Werttyp.

Darüber hinaus gibt es in der CLR sogar CIL-Befehle, die es ermöglichen, einen Werttypen in einen Referenztyp umzuwandeln.
Aus diesem Grund gibt es für die einfachen Datentypen (int, float, ...), die ja Werttypen sind, in C# keine Wrapper-Klassen (wie in Java). Ebenso ist es für die *structures* in C#, die ebenfalls Werttypen sind, nicht notwendig, auch noch für jede *structure* eine eigene Wrapper-Klasse zu definieren.
Umgekehrt können Referenztypen sogar wieder in Werttypen zurückverwandelt werden.

box *valTypeTok* convert value type to object reference
..., valueType → ..., obj

box macht aus jedem Werttyp einen äquivalenten Referenztyp.

unbox *valueType* convert boxed type into its raw form
..., obj → ..., valueTypePtr
ACHTUNG: *unbox* ist nicht die genaue Umkehrung zu *box*! (siehe unten)

valTypeTok & *valueType* sind Metadatatoken!

Beispiel:

```
object obj = 3; // Boxing
int x = (int) obj; // Unboxing
```

CIL-Code:

```
.maxstack  1
```

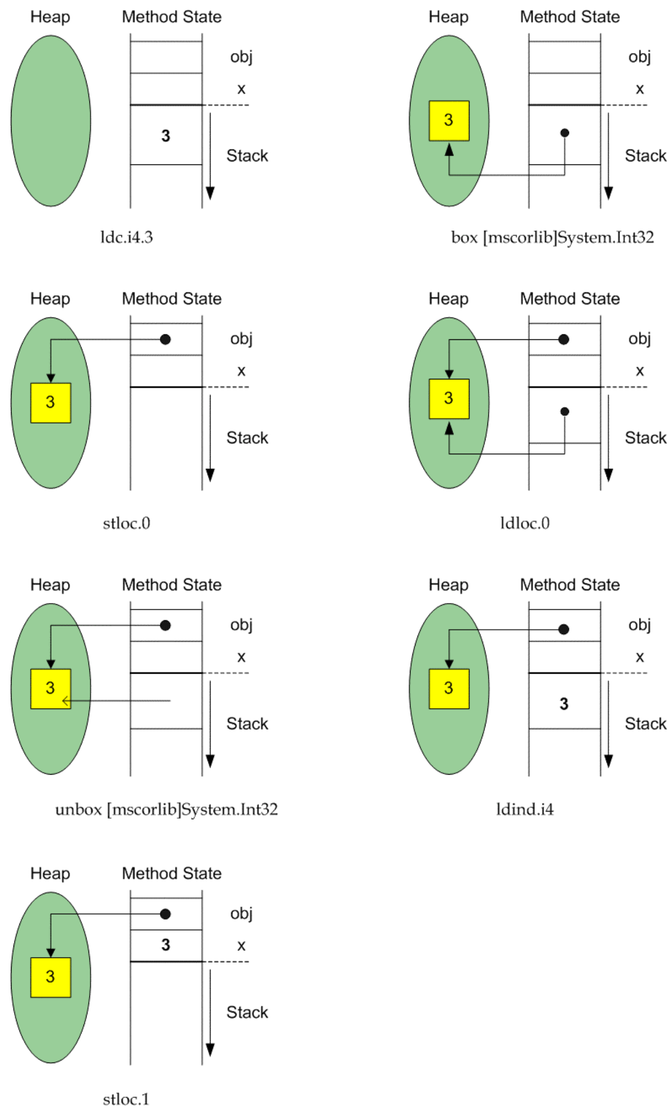
```
 .locals init ([0] object obj,
        [1] int32 x)
Boxing:
 IL_0000:  ldc.i4.3
 IL_0001:  box        [mscorlib]System.Int32
 IL_0006:  stloc.0

Unboxing:
 IL_0007:  ldloc.0
 IL_0008:  unbox      [mscorlib]System.Int32
 IL_000d:  ldind.i4
 IL_000e:  stloc.1
 IL_000f:  ret
```

Der Variable *obj* (vom Typ *object*) soll die Integerkonstante *3* zugewiesen werden. Dazu muss diese zuerst einmal auf den Stack geladen werden (ldc.i4.3). Nun liegt ein Werttyp (int32) am Stack. Jetzt kann auf diesen Werttyp der Befehl *box* angewandt werden (box [mscorlib]System.Int32). box reserviert am Heap eine entsprechende Menge Speicherplatz, in den der Wert des Werttypen (oder die Inhalte seiner Felder, wenn es sich bei dem Werttypen um eine Struktur handelt) kopiert werden. Im diesem Beispiel wird ein Int32-Objekt am Heap erzeugt. Zurück bleibt schließlich auf dem Stack ein Objektzeiger auf dieses Objekt, der dann noch in der lokalen Variablen *obj* gespeichert wird (stloc.0).

In der zweiten Anweisung wird die Objektvariable *obj*, die auf das Int32-Objekt zeigt, wieder einer Variablen x vom Werttyp Integer zugewiesen. Zuerst wird der Zeiger auf des Integerobjekt wieder auf den Stack geladen. (ldloc.0) Einen Zeiger auf das umzuwandelnde Objekt erwartet sich nun die CIL-Anweisung *unbox*. Aber: *unbox* ist nicht etwa invers zu *box*, sondern wandelt den Objektzeiger in einen verwalteten Zeiger (*managed pointer*) auf den im Objekt gespeicherten Wert um (unbox [mscorlib]System.Int32). Über diesen verwalteten Zeiger wird nun der Wert auf den Stack geladen (ldind.i4). Anschließend wird der Wert noch vom Stack in die lokale Variable *x* gespeichert (stloc.1).

Folgende Graphik veranschaulicht die Wirkung der CIL-Instruktionen:



In folgendem C#-Codestück wird Boxing und Unboxing – analog zu obigem Code – an einer C#-Struktur durchgeführt:

Beispiel:

Definition der Struktur Point

```
struct Point
{
        int x, y;
}
```

Hauptprogramm:

```
Point xy = new Point();

Object obj = xy;
Point ab = (Point) obj;
```

CIL-Code:

```
.maxstack  1
  .locals init ([0] valuetype ConsoleApplication1.Point xy,
           [1] object obj,
           [2] valuetype ConsoleApplication1.Point ab)
```

Initialisierung des Werttyp-Variable xy:

```
  IL_0000:  ldloca.s   xy
  IL_0002:  initobj    ConsoleApplication1.Point
```

Zuweisung von xy an obj – Boxing:

```
  IL_0008:  ldloc.0
  IL_0009:  box        ConsoleApplication1.Point
  IL_000e:  stloc.1
```

Zuweisung von obj an xy – Unboxing:

```
  IL_000f:  ldloc.1
  IL_0010:  unbox      ConsoleApplication1.Point
  IL_0015:  ldobj      ConsoleApplication1.Point
  IL_001a:  stloc.2
  IL_001b:  ret
```

Boxing und Unboxing einer Struktur-Werttyps (aus C#) läuft im wesentlichen genauso ab, wie bei einfachen Werttypen wie Integer oder Float. Der einzige Unterschied ist, dass beim Unboxing die Struktur, die sich auf dem Heap befindet, mittels des CIL-Befehls *ldobj* auf den Stack geladen wird (ldobj ConsoleApplication1.Point).

Delegates

Delegates (in C#) entsprechen in etwa den Funktionszeigern aus C/C++. Doch im Unterschied zu C/C++ handelt es sich bei den Delegates (in C#) um *typsichere* (und verifizierbare) Methodenzeiger.

```
public static void DemoReference(ref int a, ref int b)
{
        int x;
        a = 90;
        x = b;
```

```
}
```

Deklaration des Methodentyps = *Delegate*:

```
delegate void Reference(ref int a, ref int b);
```

Hauptprogramm:

```
Reference refer;
refer = new Reference(DemoReference);

int x = 20, y = 30;
refer(ref x, ref y);
```

Gegeben sei eine statische Methode *DemoReference*. Diese Methode benötigt als Argumente zwei Werte vom Typ *ref int* (a und b).
Nun wird ein Delegate deklariert, also der Typ eines Methodenzeigers (eine Variable dieses Typs kann allerdings – wie aus der Deklaration des Delegatetyps hervorgeht – nur auf Methoden zeigen, die als Methodenargument zwei Werte vom Typ *ref int* übergeben bekommen und *void* als Rückgabetyp haben – die Signaturen der zuweisbaren Methoden müssen gleich der Signatur des Delegatetyps sein).

Im Hauptprogramm wird nun eine Variable *refer* vom Delegatetyp *Reference* deklariert. Anschließend wird *refer* auch gleich eine Referenz auf die Methode *DemoReference*, also ein Methodenzeiger zugewiesen.

Der Aufruf der Methode DemoReference via Methodenzeiger durch refer(ref x, ref y) erfolgt zuletzt.

Ein Blick in den CIL-Code des Hauptprogramms zeigt nunmehr die Details:

```
.maxstack  3
  .locals init ([0] class DemoApplication2.Class1/Reference refer,
           [1] int32 x,
           [2] int32 y)
  IL_0000:  ldnull
  IL_0001:  ldftn      void DemoApplication2.Class1::DemoReference(int32&,
                                                                   int32&)
  IL_0007:  newobj     instance void DemoApplication2.Class1/Reference::.ctor(object,
                                                                   native int)
  IL_000c:  stloc.0
  IL_000d:  ldc.i4.s   20
  IL_000f:  stloc.1
  IL_0010:  ldc.i4.s   30
  IL_0012:  stloc.2
  IL_0013:  ldloc.0
  IL_0014:  ldloca.s   x
```

```
 IL_0016:  ldloca.s   y
 IL_0018:  callvirt   instance void
DemoApplication2.Class1/Reference::Invoke(int32&,
                                          int32&)
 IL_001d:  ret
```

Delegatetypen sind Referenztypen, daher referenziert eine Delegatevariable ein Objekt auf dem Heap.

Ein Delegatetyp besitzt immer einen Konstruktor (der mit new auf gerufen wird, siehe Hochsprachencode): newobj instance void DemoApplication2.Class1/Reference::.ctor(object, native int) (IL_0007).

Dem Konstruktor des Delegates wird ein Verweis auf das Empfängerobjekt, sowie ein Verweis auf die eigentliche Methode, auf die die Delegatevariable zeigt, übergeben.

Da es sich bei DemoReference um eine statische Methode handelt, wird für object, das Empfängerobjekt, *null* angenommen (ldnull). Der Zeiger (Adresse) auf die Methode DemoReference wird mittels ldftn auf den Stack geladen.

ldftn *method* load method pointer
… → …, ftn
pusht einen Zeiger auf eine Methode referenziert von *method* (Metadatatoken) auf den Stack. Wird zur Konstruktion eines Delegates verwendet.

Zuletzt wird noch der Zeiger auf das Delegateobjekt in die Variable refer gespeichert.

Der Aufruf einer Methode via Delegate verläuft ähnlich einem Objektmethodenaufruf:

Zuerst wird die Referenz auf das Delegateobjekt auf den Stack geladen, anschließend die notwendigen Parameter für den Delegateaufruf. Dann erfolgt der Aufruf der Methode *Invoke* des Delegates mittels callvirt. Die Signatur von Invoke entspricht der Signatur des Delegatetyps, beide Signaturen sind in Rückgabetyp, Zahl und Typ der Methodenparameter gleich. Mit *Invoke* wird die im Delegateobjekt gekapselte Methode aufgerufen.

Auch eine Objektmethode kann via Delegate gekapselt werden. In diesem Fall muss auch das entsprechende Empfängerobjekt berücksichtigt werden.

Beispiel

Klasse Time mit Methode SayTime:

```
public class Time {
  ...
  virtual public void SayTime(int hour)
  {
    Console.WriteLine("It's " + hour + " o'clock!");
  }
```

}

Deklaration des Methodentyps = *Delegate:*

```
delegate void Notifier(int hour);
```

Hauptprogramm:

```
Time FirstTime = new Time();
Notifier notify;

notify = new Notifier(FirstTime.SayTime);
notify (5);
```

CIL:

```
.maxstack  3
  .locals init ([0] class DemoApplication2.Class1/Time FirstTime,
           [1] class DemoApplication2.Class1/Notifier notify)
  IL_0000:  newobj     instance void DemoApplication2.Class1/Time::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  dup
  IL_0008:  ldvirtftn  instance void DemoApplication2.Class1/Time::SayTime(int32)
  IL_000e:  newobj     instance void DemoApplication2.Class1/Notifier::.ctor(object,
                                                         native int)
  IL_0013:  stloc.1
  IL_0014:  ldloc.1
  IL_0015:  ldc.i4.5
  IL_0016:  callvirt   instance void DemoApplication2.Class1/Notifier::Invoke(int32)
  IL_001b:  ret
```

ldvirtftn *mthd* load a virtual method pointer
..., object → ..., ftn
pusht Adresse einer virtuellen Methode *mthd* auf den Stack. Wird ebenfalls zur Konstruktion eines Delegates verwendet.

Ein Objekt der Klasse Time, FirstTime, wird angelegt.
Dann wird der Delegatevariablen notify (vom Typ Notifier), die virtuelle Objektmethode FirstTime.SayTime zugewiesen: Im CIL-Code wird zuerst die lokale Variable FirstTime auf den Stack geladen (ldloc.0), diese Objektreferenz auf dem Stack wird anschließend verdoppelt, da der Befehl ldvirtftn sowie der Konstruktor des Delegatetyps eine solche Objektreferenz erwarten.

Ldvirtftn *mthd* lässt die virtuelle Addresse der entsprechenden virtuellen Methode des Objekts, dessen Adresse am Stack liegt (*mthd* ist ein Metadatatoken), am Stack zurück.
Aus dieser Adresse (Adresse der virtuellen Objektmethode) und der Adresse des zugehörigen Objekts (Adresse des Empfängerobjekt, aus der Verdopplung) wird jetzt das zugehörige Delegateobjekt konstruiert (newobj).
Beim Aufruf werden die Referenz auf das Delegateobjekt (ldloc.1 = notify) und die benötigten Parameter auf den Stack geladen. Mit callvirt wird wieder die *Invoke*-Methode des Delegateobjekts aufgerufen, die den Aufruf der im Delegate gekapselten Methode veranlasst.

Basisinstruktionssatz 2

Operationen

Der Basisinstruktionssatz der Common Intermediate Language formt eine *turingvollständige* Menge an Grundoperationen. Ein wesentlicher Bestandteil sind die arithmetischen Grundoperationen:

add add numeric values
..., value 1, value 2 → ... result
addiert value 1 und value 2

sub subtract numeric values
..., value 1, value 2 → ... result
subtrahiert *value 2* von *value 1*

mul multiply values
..., value 1, value 2 → ... result
multipliziert *value 1* mit *value 2*

div divide values
..., value 1, value 2 → ... result
dividiert *value 1* durch *value 2*

neg negate
..., value → ..., result
Gibt das 2er Komplement für Ganzzahl- und Fliesskommatypen zurück.

rem compute remainder
..., value 1, value 2 → ... result
ergibt den Rest von *value 1* dividiert durch *value 2*

Die angeführten Befehle existieren darüber hinaus auch noch in einer Variante für *unsigned* (nicht vorzeichenbehaftet) Operanden. Sie tragen den Suffix (*.un*).

Weiters weisen die CLR (und damit die CIL Instruktionen) eine Besonderheit im Unterschied zur Java VM bei arithmetischen Operationen auf: Eine arithmetische Operation kann zur Laufzeit – falls ein Überlauf (Overflow) auftritt – eine Ausnahme (*OverflowException*) auslösen. Befehle für arithmetische Operationen, die eine OverflowException auslösen können, tragen ein (.ovf) als Suffix im CIL-Code.

Somit ergibt sich nochmals eine weitere Anzahl von CIL-Instruktionen für die arithmetischen Grundoperationen:

add.ovf, add.ovf.un add [un]signed integer values with overflow check
div.un divide interger values, unsigned
mul.ovf, mul.ovf.un multiply integer values with overflow check
rem.un compute integer remainder, unsigned
sub.ovf, sub.ovf.un subtract integer values, checking for overflow

Typkonvertierung

Eine weitere Reihe von CIL Befehlen dient dazu, am Stack eine Typkonvertierung der numerischen Werttypen (int32, int64, F, ...) durchführen zu können.

conv.<to type> data conversion
..., value à ... result
Konvertiert *value* in den (im Opcode) spezifizierten Typ **<to type>**.
<to type> ist i1, i2, i4, i8, r4, r8, u1, u2, u4, u8, i, u, r.un

Von conv.* gibt es eine Version (auch *unsigned*) mit Überlaufprüfung :

conv.ovf.<to type> data conversion with overflow detection
<to type> ist i1, i2, i4, i8, u1, u2, u4, u8, i, u

conv.ovf.<to type>.un unsigned data conversion with overflow detection
<to type> ist i1, i2, i4, i8, u1, u2, u4, u8, i, u

Beispiel:

```
float fl = 0.0F;
double dbl = 0.0;
int i = 30;
```

```
long lint = 40;

i = (int) lint;

lint = i;

dbl = fl;

fl = (float) dbl;
```

CIL:

```
.maxstack  1
  .locals init ([0] float32 fl,
           [1] float64 dbl,
           [2] int32 i,
           [3] int64 lint)
  IL_0000:  ldc.r4     0.0
  IL_0005:  stloc.0
  IL_0006:  ldc.r8     0.0
  IL_000f:  stloc.1
  IL_0010:  ldc.i4.s   30
  IL_0012:  stloc.2
  IL_0013:  ldc.i4.s   40
  IL_0015:  conv.i8
  IL_0016:  stloc.3

// i = (int) lint;
IL_0017:  ldloc.3
  IL_0018:  conv.i4
  IL_0019:  stloc.2

// lint = i;
  IL_001a:  ldloc.2
  IL_001b:  conv.i8
  IL_001c:  stloc.3

// dbl = fl;
  IL_001d:  ldloc.0
  IL_001e:  conv.r8
  IL_001f:  stloc.1

// fl = (float) dbl;
  IL_0020:  ldloc.1
  IL_0021:  conv.r4
  IL_0022:  stloc.0
```

IL_0023: ret

Bitweise Operationen

Diese Operatoren ermöglichen die bitweise Verknüpfung oder Behandlung von Werten am Stack.

and bitwise AND
..., value 1, value 2 → ... result
bitweises AND zweier ganzzahliger Werte, lässt ganzzahliges Resultat zurück

or bitwise OR
..., value 1, value 2 → ... result
bitweises OR zweier ganzzahliger Werte, lässt ganzzahliges Resultat zurück

xor bitwise XOR
..., value 1, value 2 → ... result
bitweises OR zweier ganzzahliger Werte, lässt ganzzahliges Resultat zurück

not bitwise complement
..., value → ... result
bitweises Komplement von *value*, *result* ist vom selben Typ wie *value*

Shift-Operationen

Shiftoperationen dienen zur Bitmanipulation von Werten am Stack. Mit ihnen kann zum Beispiel die Semantik bestimmter Operationen in Hochsprachen nachgebaut werden [Gou02].

shl shift integer left (*arithmetic* shift)
..., value, shiftAmount → ... result
shiftet *value* um *shiftAmount* Stellen nach links

shr shift integer right (*arithmetic* shift)
..., value, shiftAmount → ... result
shiftet *value* um *shiftAmount* Stellen nach rechts

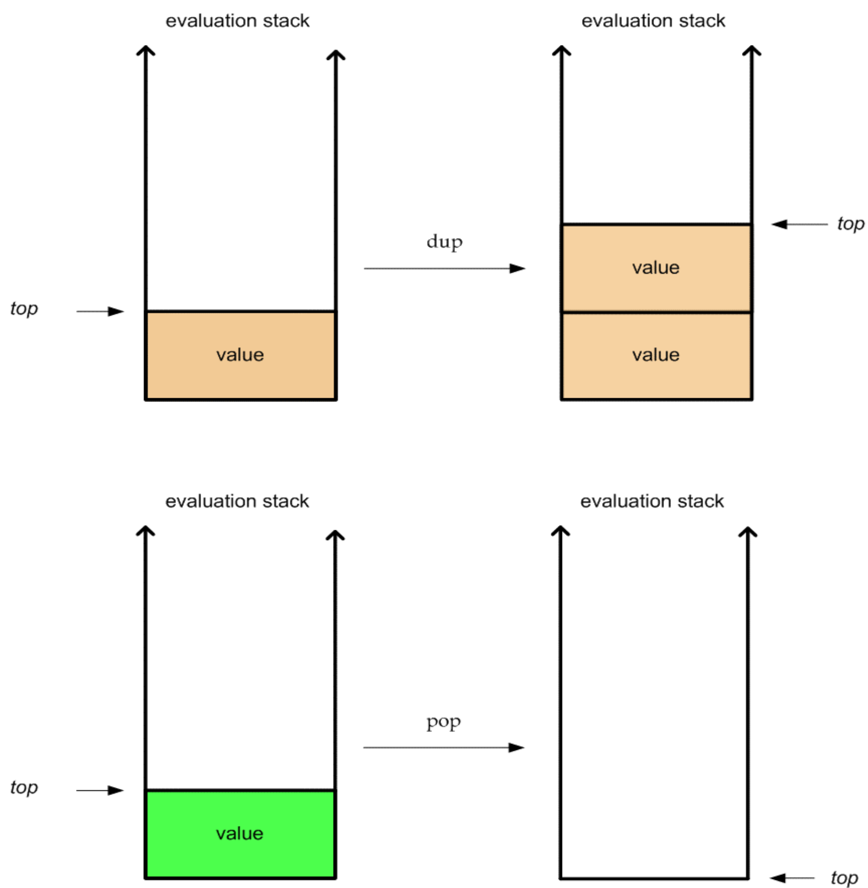
shr.un shift integer right, unsigned (*logical* shift)
..., value, shiftAmount → ... result
shiftet *value* um *shiftAmount* Stellen nach rechts

Spezielle Operationen

dup duplicate the top value of the stack
..., value → value, value
dupliziert das oberste Element am Stack

pop pop a value from the stack
..., value → ...
entfernt (*poppt*) das oberste Element vom Stack

nop Do nothing
... → ...
macht gar nichts: die *leere* Operation



Boolsche Operationen

ceq compare equal
..., value 1, value 2 → ... result
legt 1 auf den Stack, falls *value 1* gleich *value 2*, sonst 0

cgt compare greater than
..., value 1, value 2 → ... result
legt 1 auf den Stack, falls *value 1* größer *value 2*, 0 sonst

clt compare less than
..., value 1, value 2 → ... result
legt 1 auf den Stack, falls *value 1* kleiner *value 2*, sonst 0

unsigned / unordered Versionen:
cgt.un *target* compare greater than, unsigned or unordered
clt.un *target* compare less than, unsigned or unordered

(*unordered* bei real/floating point – Operanden: Test positiv, wenn mindestens einer der beiden Operanden NaNS ist = Not-a-Number-Symbol in IEEE 754)

Es fällt auf, dass es nur boolsche Operationen für die Überprüfung auf Gleichheit (**ceq**) und die Vergleiche auf *größer* (**cgt**, >) und *kleiner* (**clt**, <) gibt.
Compiler bedienen sich aus diesem Grund Identitäten wie der im folgenden angeführten, um auch die restlichen Vergleiche (ungleich ≠, größer gleich ≤, kleiner gleich ≥) durchführen zu können.

Identitäten:	Äquivalente Instruktionssequenz:
$\forall$ a, b: $(a \neq b) = \neg(a = b)$	**ceq; ldc.1; xor**
$\forall$ a, b: $(a \leq b) = \neg(a > b)$	**cgt; ldc.1; xor**
$\forall$ a, b: $(a \geq b) = \neg(a < b)$	**clt; ldc.1; xor**

Beispiel:

```
int a = 10;
int b = 20;

bool c = ( a != b );
```

CIL:

```
.maxstack  2
  .locals init ([0] int32 a,
```

```
        [1] int32 b,
        [2] bool c)
 IL_0000:  ldc.i4.s   10
 IL_0002:  stloc.0
 IL_0003:  ldc.i4.s   20
 IL_0005:  stloc.1

IL_0006:  ldloc.0
 IL_0007:  ldloc.1
 IL_0008:  ceq
 IL_000a:  ldc.i4.0
 IL_000b:  ceq

IL_000d:  stloc.2
 IL_000e:  ret
```

In diesem Beispiel soll der boolschen Variable c der Wahrheitswert des boolschen Ausdrucks ($a \neq b$), also ($10 \neq 20$), zugewiesen werden. Der C#-Compiler bedient sich hier allerdings nicht der oben angegebenen Identität für die Ungleichheitsoperation, sondern der Identität ceq; ldc.0; ceq, um den Test auf Ungleichheit durchzuführen.

Beispiel:

```
int a = 10;
int b = 20;
bool c = ( a >= b );
```

CIL:

```
.maxstack  2
 .locals init ([0] int32 a,
        [1] int32 b,
        [2] bool c)
 IL_0000:  ldc.i4.s   10
 IL_0002:  stloc.0
 IL_0003:  ldc.i4.s   20
 IL_0005:  stloc.1
 IL_0006:  ldloc.0
 IL_0007:  ldloc.1
 IL_0008:  clt
 IL_000a:  ldc.i4.0
 IL_000b:  ceq
 IL_000d:  stloc.2
 IL_000e:  ret
```

Für die Auswertung des Ausdrucks (a ≥ b) wird die Identität clt; ldc.0; ceq verwendet.

Flow Control – Verzweigen und Springen

Neben den Methodenaufrufen sind es die Sprungbefehle, die eine Abänderung des Kontrollflusses verursachen.
Die Möglichkeit zum Springen und Verzweigen im CIL-Code ist notwendig, um bedingte Statements (if-Schleifen) und andere Schleifen (while-, for-Schleifen) nachzubilden.
Unbedingte Sprünge ändern die Verlauf des Kontrollflusses auf jeden Fall. Bedingte Sprünge verzweigen entsprechend Wahrheitswerten, die entweder schon auf dem Stack obenauf liegen oder aus der Auswertung boolscher Ausdrücke resultieren.

Unbedingter Sprung:

br *target* unconditional branch
..., → ...
Kontrollfluss springt zu *target*

Bedingte Sprünge:

brfalse *target* branch on null, false or zero
..., value → ...
Kontrollfluss springt zu *target*, falls *value* false, 0 oder null ist.

brtrue *target* branch on non-false or non-null
..., value → ...
Kontrollfluss springt zu *target*, falls *value* ***nicht****false*, *ungleich 0* oder ***nicht****null* ist.

beq *target* branch on equal
..., value 1, value 2 → ...
Kontrollfluss springt zu *target*, falls *value 1* gleich *value 2* ist.

bge *target* branch on greater than or equal to
..., value 1, value 2 → ...
Kontrollfluss springt zu *target*, falls *value 1* größer oder gleich *value 2* ist.

bgt *target* branch on greater than
..., value 1, value 2 → ...
Kontrollfluss springt zu *target*, falls *value 1* größer *value 2* ist.

unsigned / unordered Versionen:
bge.un *target* branch on greater than or equal to, unsigned or unordered
bgt.un *target* branch on greater than, unsigned or unordered

ble *target* branch on less than or equal to
..., value 1, value 2 → ...
Kontrollfluss springt zu *target*, falls *value 1* kleiner oder gleich *value 2* ist.

blt *target* branch on less than
..., value 1, value 2 → ...
Kontrollfluss springt zu *target*, falls *value 1* kleiner *value 2* ist.

bne.un *target* branch on not equal or unordered
..., value 1, value 2 → ...
Kontrollfluss springt zu *target*, falls *value 1* ungleich *value 2* ist.

unsigned / unordered Versionen:
ble.un *target* branch on greater than or equal to, unsigned or unordered
blt.un *target* branch on greater than, unsigned or unordered

Beispiele:

Kurzschlussauswertung:

Beispiel 1.

```
bool eins = false;
bool zwei = true;

if (eins || zwei)
{
  Console.WriteLine("if-Schleife betreten");
}
```

CIL:

```
.maxstack  1
  .locals init ([0] bool eins,
           [1] bool zwei)
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.0
  IL_0002:  ldc.i4.1
```

```
  IL_0003:  stloc.1

IL_0004:  ldloc.0
  IL_0005:  brtrue.s   IL_000a
  IL_0007:  ldloc.1
  IL_0008:  brfalse.s  IL_0014
  IL_000a:  ldstr      "if-Schleife betreten"
  IL_000f:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0014:  ret
```

Beispiel 2.

```
bool eins = false;
bool zwei = true;

if (eins && zwei)
{
  Console.WriteLine("if-Schleife betreten");
}
```

CIL:

```
.maxstack  1
  .locals init ([0] bool eins,
           [1] bool zwei)
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.0
  IL_0002:  ldc.i4.1
  IL_0003:  stloc.1

IL_0004:  ldloc.0
  IL_0005:  brfalse.s  IL_0014
  IL_0007:  ldloc.1
  IL_0008:  brfalse.s  IL_0014
  IL_000a:  ldstr      "if-Schleife betreten"
  IL_000f:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0014:  ret
```

In den Beispielen 1 und 2 soll anhand auf einfache Weise veranschaulicht werden, wie der Compiler die Kurzschlussauswertung von boolschen Ausdrücken im Kopf einer if-Schleife realisiert.

In Beispiel 1 sind die beiden boolschen Variablen durch ODER verknüpft. Das bedeutet, falls die erste boolsche Variable eins true ist, dass sofort der Schleifenrumpf ausgeführt werden kann. Falls eins wirklich true ist, passiert dies auch: Der Kontrollfluss springt zur Anweisung in IL_000a, wo der Rumpf der Schleife beginnt.

Geht der erste Test negativ aus (d. h. eins war false), besteht immer noch die Chance, dass zwei true ist. zwei wird daher auf false gestestet: Falls der Test negativ ausfällt (zwei war true), fällt die Flusskontrolle durch und gelangt zum Schleifenrumpf. Im anderen Fall (Test positiv – zwei ist ebenfalls false) braucht die Schleife nicht ausgeführt zu werden – Sprung zu IL_0014 (brfalse.s IL_0014).

In Beispiel 2 hingegen sind eins und zwei mit UND verknüpft. Alle Variable (eins und zwei) müssen den Wahrheitswert von true aufweisen, damit der Rumpf der Schleife zur Ausführung gelangt. Sobald eine Variable mit false belegt ist, kann die Schleife übersprungen werden. Im CIL-Code wird eine Variable des boolschen Ausdrucks nach der anderen auf den Stack geladen und auf false getestet. Bei false erfolgt ein Sprung hinter den Schleifenkörper zu IL_0014 (brfalse.s IL_0014). Fällt die Flusskontrolle durch alle Tests durch, so gelangt sie in den Schleifenrumpf und kann die Console.WriteLine(...) Methode aufrufen.

Beispiele:

while-Schleifen:

konventionelle while-Schleife.

```
int a = 10;
int b = 20;

while (b > a)
{
  b--;
}
```

CIL:

```
.maxstack  2
  .locals init ([0] int32 a,
           [1] int32 b)
  IL_0000:  ldc.i4.s   10
  IL_0002:  stloc.0
  IL_0003:  ldc.i4.s   20
  IL_0005:  stloc.1
  IL_0006:  br.s       IL_000c
  IL_0008:  ldloc.1
  IL_0009:  ldc.i4.1
  IL_000a:  sub
  IL_000b:  stloc.1
  IL_000c:  ldloc.1
  IL_000d:  ldloc.0
  IL_000e:  bgt.s      IL_0008
  IL_0010:  ret
```

Der Compiler löst die while-Schleife so auf, dass der CIL-Code zur Überprüfung der Ausführungsbedingung nach dem Code für den Schleifenrumpf steht.
Wie sieht das konkret aus?
Im Beispiel der oben angeführten einfachen while-Schleife, erfolgt die Überprüfung der Bedingung (b > a) ab Zeile IL_000c: Die beiden Werte (gespeichert in den lokalen Integervariablen a und b) werden auf den Stack geladen, anschließend wird die Bedingung geprüft (größer, bgt.s). Fällt die Überprüfung positiv aus, verzweigt das Programm zum Schleifenrumpf in Zeile IL_0008.
Beim erstmaligen (versuchten) Einsprung in die Schleife (Zeile IL_0006: br.s IL_000c) wird unbedingt zur Überprüfung der Bedingung in Zeile IL_000c verzweigt. Auf diese Weise braucht der Code für die Schleifenbedingung nur einmal ausgegeben zu werden.
Eine *while*-Schleife wird vom Compiler eigentlich in eine *do-while*-Schleife übersetzt.

do-while-Schleife:

```
int a = 10;
int b = 20;

do
{
   b--;
} while (b > a);
```

CIL:

```
.maxstack  2
  .locals init ([0] int32 a,
           [1] int32 b)
  IL_0000:  ldc.i4.s   10
  IL_0002:  stloc.0
  IL_0003:  ldc.i4.s   20
  IL_0005:  stloc.1
  IL_0006:  ldloc.1
  IL_0007:  ldc.i4.1
  IL_0008:  sub
  IL_0009:  stloc.1
  IL_000a:  ldloc.1
  IL_000b:  ldloc.0
  IL_000c:  bgt.s      IL_0006
  IL_000e:  ret
```

Die do-while-Schleife wird vom Compiler straight-forward in CIL übersetzt: Zuerst der Schleifenrumpf, der zumindest einmal ausgeführt wird, dann die Schleifenbedingung.

Eine ausführliche Diskussion von Übersetzungstechniken für Programmkonstrukte, die den Control Flow beeinflussen, findet in Kapitel 9 von [Gou02] statt.

Literatur

[BBMW02]: Beer W., Birngruber D., Mössenböck H., Wöß A.: Die .NET-Technologie, dpunkt.verlag, 2002

[Bur02]: Burton K.: .NET Common Language Runtime Unleashed, Sams Publishing 2002

[ECMA02]: ECMA-International: Standard ECMA-335: Common Language Infrastructure (CLI) Partitions I to V, http://www.ecma-international.org, December 2002

[Gou02]: Gough J.: Compiling for the .NET Common Language Runtime, Prentice Hall, 2002

[Ven99]: Venners B.: Inside the Java 2 Virtual Machine, McGraw Hill, 1999

[YeL97]: Yellin F., Lindholm T.: The Java Virtual Machine Specification, Addison-Wesley, 1997

Anmerkungen zur Literatur:

Vollständige Spezifikation der CIL in:
ECMA-335, CLI Partition III: CIL Instruction Set

John Gough:
Compiling for the .NET Common Language Runtime
Prentice Hall, 2002
Kapitel 2 (bzw. 3 & 4), sowie Kapitel 8 & 9

Kevin Burton:
.NET Common Language Runtime Unleashed
Sams Publishing, 2002
Kapitel 5

Ergänzender Überblick und Einführung:
W. Beer, D. Birngruber, H. Mössenböck, A. Wöß:
Die .NET-Technologie, dpunkt.verlag 2002
Kapitel 3

Java-Ressource (zum Vergleich CLR – Java VM, CIL – Java-Bytecode):
Bill Venners:
Inside the Java 2 Virtual Machine
McGraw Hill, 1999

Appendix: Exception Handling Instruktionen

In C# kann der Programmierer eine selbstdefinierte Ausnahme-Klasse von der Klasse Exception des System-Namespaces (*System.Exception*) ableiten:

```
using System;

…
…

public class MyException : System.Exception
{

	public override string Message
	{
		get
		{
			string msg = "Value in Array bigger than 10";
			return msg;
		}
	}

}
```

Die Ausnahme MyException kann jetzt – zum Beispiel, wenn die in der Exception definierte Fehlerbedingung eintritt – explizit geworfen werden. Soll die Ausnahme auch behandelt werden, dann muss das Codestück, in dem die Ausnahme auftreten kann, in einen try-Block gestellt werden, daran schließen catch-Blöcke an, in denen speziell definierte Behandlungen einer Ausnahme möglich sind.

```
try {
	int [] a = new int[3];
	a[1] = 6 + 7;
	if (a[1] > 10) {
		throw new MyException();
	}
}

catch (MyException e) {
	Console.WriteLine("MyException occurred!");
}
catch (Exception e) {
	Console.WriteLine(e.ToString());
}
finally {
	Console.WriteLine("Finally-Block reached!");
}
```

Optional kann hinter die try-/catch-Blöcke noch eine finally-Klausel gestellt werden, die auf jeden Fall ausgeführt wird.

```
.maxstack  3
  .locals init ([0] int32[] a,
           [1] class DemoApplication3.Class1/MyException e,
           [2] class [mscorlib]System.Exception V_2).
```

TRY-BLOCK

```
.try  {
    .try {
      IL_0000:  ldc.i4.3
      IL_0001:  newarr     [mscorlib]System.Int32
      IL_0006:  stloc.0
      IL_0007:  ldloc.0
      IL_0008:  ldc.i4.1
      IL_0009:  ldc.i4.s   13
      IL_000b:  stelem.i4
      IL_000c:  ldloc.0
      IL_000d:  ldc.i4.1
      IL_000e:  ldelem.i4
      IL_000f:  ldc.i4.s   10
      IL_0011:  ble.s      IL_0019
      IL_0013:  newobj     instance void DemoApplication3.Class1/MyException::.ctor()
      IL_0018:  throw
      IL_0019:  leave.s    IL_0036
    }  // end .try
```

Die Ausnahme MyException wird nur dann geworfen, wenn die Bedingung a[1] > 10 erfüllt ist. Die Überprüfung der Bedingung ist rot gekennzeichnet. Falls sie nicht erfüllt ist, geschieht gar nichts, der Kontrollfluss springt zu leave in Zeile IL_0019.
Trifft die Bedingung zu, so wird ein Objekt der Klasse MyException am Heap erstellt. Zurück am Stack hinterlässt newobj die Objektreferenz auf das Exceptionobjekt. throw wirft nun die Ausnahme und löst ihre Behandlung aus, d. h. der Kontrollfluss wird zu einem Catch-Block transferiert, der den Typ (zuerst spezielle Typ(en), dann der allgemeine: System.Exception) der geworfenen Ausnahme behandelt: In diesem Catch-Block findet nun die speziell definierte Behandlung der Ausnahme statt.

throw throw an exception
..., object → ..., object (?)
object ist eine Objektreferenz (Typ o) auf ein Exceptionobjekt am Heap

CATCH-BLÖCKE

```
  catch DemoApplication3.Class1/MyException
    {
      IL_001b:  stloc.1
      IL_001c:  ldstr      "MyException occurred!"
      IL_0021:  call       void [mscorlib]System.Console::WriteLine(string)
      IL_0026:  leave.s    IL_0036
```

```
  } // end handler
  catch [mscorlib]System.Exception
  {
    IL_0028:  stloc.2
    IL_0029:  ldloc.2
    IL_002a:  callvirt   instance string [mscorlib]System.Exception::ToString()
    IL_002f:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_0034:  leave.s    IL_0036
  } // end handler
  IL_0036:  leave.s    IL_0043
} // end .try
```

FINALLY-KLAUSEL:

```
finally
 {
  IL_0038:  ldstr      "Finally-Block reached!"
  IL_003d:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0042:  endfinally
 } // end handler
 IL_0043:  ret
```

try / catch – Blöcke können nicht mit den konventionellen Branch- bzw. Jump-Instruktionen der CIL verlassen werden; Ein Catch- bzw. try-Block wird mittels leave *target* verlassen, die Ablaufkontrolle wird zur Sprungmarke *target*, dem Ziel des Sprunges übertragen.
Die optionale finally-Klausel anschließend an die Catch-Blöcke einer Ausnahmebehandlung wird auf jeden Fall ausgeführt. Sie endet mit der endfinally Anweisung.