Introduzione a LLVM per il corso di Compilatori (parte relativa al front-end)

Mauro Leoncini

December 10, 2024

1 Introduzione

`LLVM` è il nome generico di una serie di componenti software che possono essere utilizzati, o comunque essere di grande ausilio, nella realizzazione di compilatori[1]. Il componente principale è `clang`, un compilatore per la famiglia di linguaggi che, più o meno lontanamente, derivano dal `C` (`C`, `C++` e `Objective-C`). L'architettura di `clang` è quella tipica di un compilatore esemplare: è dotato di un *front-end* che, a partire dal codice sorgente, produce prima una rappresentazione ad albero (*Abstract Syntax Tree*, o semplicemente *AST*) e poi un codice "lineare" intermedio (*Intermediate Representation*, *IR*), da un *middle-end* che ottimizza il codice IR in maniera indipendente dall'architettura e da un *back-end* che converte l'IR ottimizzato in codice macchina per la piattaforma in uso.
LLVM tuttavia contiene molti altri strumenti di utilità. Le stesse componenti "inferiori", ovvero middle- e back-end, possono essere visti come strumenti a sè, incorporabili (e realmente incorporati) in compilatori per altri linguaggi. Un esempio notevole è `rustc`, un compilatore per il linguaggio `Rust`, il cui front-end produce appunto codice LLVM-IR.
Nella prima parte del corso di Compilatori svolgiamo un esercizio analogo a quello di `rustc`, sia pur in scala enormemente ridotta. Parallelamente all'approfondimento teorico, intendiamo cioè produrre il front-end per un semplice linguaggio funzionale, che abbiamo chiamato `lfm` (`linguaggio`

[1] Originariamente `LLVM` erano le iniziali di *Low Level Virtual Machine*. La versione ufficiale attuale è che esso non indichi nulla. È rimasto cioè solo un nome.

`funzionale minimo`), il cui output sia codice LLVM-IR. Con lo svolgimento di questo progetto ci proponiamo di raggiungere due importanti obiettivi.

1. Il primo e più ovvio è appunto di svolgere una sperimentazione semplice ma realistica, modellata cioè sulla falsariga della implementazione di linguaggi esistenti.

2. Un secondo obiettivo è di fornire allo studente una conoscenza di LLVM-IR già sufficiente ad affrontare con profitto lo studio di tecniche di ottimizzazione del codice (middle-end in particolare) che saranno oggetto della seconda parte del corso.

Tutti gli strumenti e le librerie LLVM sono scritti in C++ con ampio utilizzo di `STL` (`Standard Template Library`). Per la generazione della rappresentazione LLVM-IR di un programma `lfm`, partiremo da una rappresentazione del suo AST. La generazione sarà guidata da una visita in ordine posticipato dell'AST che, in corrispondenza di ogni nodo, produrrà il codice facendo uso dei metodi di opportune classi dell'infrastruttura LLVM. Alcune di tali classi corrispondono alle entità fondamentali che entrano in gioco nella descrizione strutturale della rappresentazione LLVM-IR (modulo, funzione, blocco, valore). Prima quindi di introdurre le classi e i metodi rilevanti per il nostro esercizio progettuale, dobbiamo descrivere almeno sommariamente il modello architetturale di LLVM-IR, che ovviamente include le istruzioni. Si tenga invece presente che, per la trasformazione del codice sorgente nella rappresentazione AST non abbiamo utilizzato l'infrastruttura LLVM.

Un'ultima precisazione. In questo documento presentiamo solo le caratteristiche essenziali di LLVM-IR che risultano utili per poter svolgere il progetto del front-end di `lfm`, un linguaggio in cui l'unico dato di base sono i numeri interi a 32 bit. Per la documentazione ufficiale ed esaustiva di LLVM, che include il manuale per il codice intermedio, si veda la pagina all'indirizzo `https://llvm.org/docs/`

2 La "forma" human-readable di LLVM-IR

Il risultato della compilazione di codice sorgente in LLVM-IR può assumere tre diverse forme, che chiaramente riflettono altrettanti utilizzi intesi per esse. Quando, nella precedente sezione introduttiva, abbiamo affermato di voler sviluppare un front-end che producesse codice intermedio LLVM-IR,

intendevamo una sola di queste rappresentazioni, ovvero la c.d. rappresentazione esterna (human-readable). Nel seguito, quando faremo riferimento al codice intermedio, intenderemo precisamente la sua forma human-readable. Per completezza, le altre due forme sono: una sorta di struttura dati in memoria, adatta ad essere oggetto di passaggi di ottimizzazione, e un formato binario, detto *bitcode*, particolarmente adatto per il caricamento rapido da parte di un compilatore *Just-in-Time*.

I file che memorizzano il codice intermedio (human-readable) hanno, di norma, l'estensione `ll` e possono essere compilati in codice macchina utilizzando uno degli strumenti di LLVM. Per i nostri scopi, avremo bisogno delle seguenti applicazioni.

1. Il compilatore `clang/clang++`. La caratteristica che qui ci interessa non è tanto la compilazione di codice `C++` in codice macchina, quanto il fatto che possiamo utilizzare lo strumento per generare codice intermedio da "studiare". In altri termini, faremo uso del solo front-end del compilatore, allo scopo di meglio comprendere la trasformazione da codice sorgente in codice intermedio LLVM-IR. Il comando per compilare un programma `c++` in LLVM-IR è

   ```
   clang++ -S -emit-llvm esempio.cpp # Produce esempio.ll
   ```

   dove `esempio.cpp` è un generico programma `c++` che contiene una funzione `main`.

2. Un interprete LLVM-IR che esegue direttamente programmi `.ll`.

   ```
   lli esempio.ll # Interpreta esempio.ll
   ```

   Si tratta in realtà di una scorciatoia per sperimentazioni rapide perché quel che succede davvero è che il codice viene compilato in codice macchina e poi eseguito.

3. Il compilatore LLVM-IR, che trasforma il codice intermedio in un codice assembly macchina effettivo.

   ```
   llc esempio.ll # Compila esempio.ll in assembly esempio.s
   ```

   Si noti tuttavia che il codice prodotto non è direttamente eseguibile perché va assemblato e "link-ato". Il modo più rapido per produrre eseguibile a partite da codice intermedio è quindi di usare direttamente `clang++` come driver.

   ```
   clang++ -o esempio esempio.ll # Produce l'eseguibile esempio
   ```

3 LLVM-IR: architettura dei programmi

Sono tre le nozioni che entrano in gioco a livello di architettura di un programma IR: *moduli*, *funzioni* e *blocchi di base* (*Basic Block* o, più sinteticamente, *BB*).

3.1 Moduli

Un file di programma LLVM-IR è detto *modulo*. Più moduli possono chiaramente essere collegati (linked) nel caso il programma sia composto da più file. La struttura di un modulo include innanzitutto la definizione delle variabili globali al modulo stesso, che sono identificatori preceduti dal simbolo `@`.

```
@greeting = constant [14 x i8] c"Hello, world!\00"
```

Con questa definizione, la variabile `@greeting` risulta visibile all'interno di tutte le funzioni del modulo.

NB. Nella descrizione dell'architettura faremo già uso di alcune istruzioni per esemplificare i concetti, come nel caso appena visto. Delle istruzioni ci occuperemo però più avanti con maggiori dettagli.

Se il programma utilizza tipi di dato *custom*, questi vengono definiti immediatamente dopo la definizione di variabili. Ad esempio, potremmo avere la definizione:

```
%complex = type {
 double,
 double
}
```

Dopo le definizioni di tipo abbiamo i prototipi delle (eventuali) funzioni esterne, introdotte usando la parola chiave `declare`. Ad esempio:

```
declare i32 @printf(i8*, ...)
```

Infine vengono le definizioni di funzione, una delle quali (se il programma è costituito da un solo modulo, ovvero se il modulo "attuale" è comunque quello principale), deve avere nome `@main`.

Non abbiamo detto che, in realtà, le prime informazioni presenti in un modulo sono in qualche modo la specifica dell'architettura target (hardware e sistema operativo) e dal cosiddetto *data layout*. Al momento (parlando cioè del front-end) questo non ci interessa molto.

3.2 Funzioni

Le funzioni in LLVM-IR mantengono la classica "signature" di un linguaggio ad alto livello, sono cioè caratterizzaate da un *nome*, dal *tipo* dei dati restituiti e dagli *argomenti* (numero e tipo). La seguente signature, data come esempio, è relativa ad una funzione che riceve in input una coppia di interi a 32 bit e restituisce un intero a 32 bit.

```
define i32 @euclid(i32 %n, i32 %m)
```

Anticipando quanto avremo da dire a riguardo delle singole istruzioni, è qui necessario sottolineare che una caratteristica notevole di LLVM-IR è proprio il meccanismo di chiamata delle funzioni, molto vicino a quello di un linguaggio ad alto livello. Esiste infatti una coppia di istruzioni `call`/`ret`, che astrae tutti i delicati passaggi di chiamata e ritorno (allocazione di spazio per gli argomenti, salvataggio del punto di ritorno, ripristino del contesto della chiamata, salto al punto di ritorno) lasciando ancora una volta il compito di gestire questi aspetti alle fasi successive di compilazione. Per chiamare una funzione, si usa l'istruzione `call`. Ad esempio, per la chiamata della funzione `euclid` potremmo scrivere:

```
%mcd = call i32 @euclidi(i32 %n, i32 %m)
```

Si noti che nelle chiamate di funzione dobbiamo specificare il tipo della funzione (nell'esempio precedente `i32(i32,i32)`, che è appunto il tipo di una funzione da una coppia di `i32` ad un `i32`), oltreché, naturalmente, gli argomenti veri e propri. Tutto ciò può risultare un po' pedante, ma è chiaro che quasi sempre il carico di lavoro è scaricato sul front-end.

3.3 Basic block

Il *body* di una funzione è a sua decomponibile in una serie di *Basic Block*, che sono le unità funzinali minime. Il concetto di *Basic Block* è strettamente collegato alla nozione di *grafo di controllo (di flusso)* (*Control Flow Graph* o ancora, più semplicemente, *CFG*). In effetti i nodi un un *CFG* sono esattamente *Basic Block*. Un tale blocco è una sequenza di istruzioni che vengono eseguite nell'ordine fisico in cui sono scritte perché non includono istruzioni di salto, condizionato o incondizionato. O meglio, esse includono sempre una

tale istruzione, o un'istruzione `ret`, ma questa è precisamente l'ultima della sequenza.

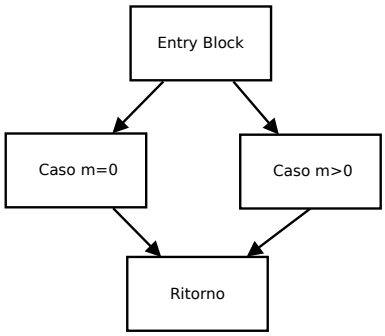
Le istruzioni di un *Basic Block* possono naturalmente includere chiamate di funzione, che non ne alterano l'ordine "sequenziale" di esecuzione proprio perché il "ritorno" della funzione restituisce il controllo all'istruzione immediatamente successiva a quella di chiamata.

Le istruzioni di salto definiscono quindi gli archi del *CFG* che collegano differenti *Basic Block* e ogni *Basic Block* è a sua volta caratterizzato da un certo numero di archi entranti e uscenti. Tutti i blocchi di una funzione, ad eccezione del primo, il cosiddetto *Entry Block*, sono etichettati con una *label*, che può essere il target di una o più istruzioni di salto. All'inizio del blocco, nel codice LLVM-IR sono indicati anche le etichette dei blocchi di base dai quali può provenire il controllo, ovvero le possibili origini degli archi entranti. quest'ultima informazione è però di fatto solo un commento, presente nella forma human-readable. Si noti che nel *CFG* sono ammessi i loop, il che significa che un'istruzione di salto che chiude un blocco può avere come target l'inizio dello stesso blocco.

Come esempio, si consideri la seguente funzione `C++` che calcola il *MCD* di due numeri mediante l'*algoritmo di Euclide*.

```
int euclid(int n, int m) {
   int retval;
   if (m==0) retval = n;
   else retval = euclid(m,n%m);
   return retval;
}
```

La funzione è stata scritta in modo che risulti chiaramente visibile la struttura del *CFG*, che sarà più evidente a livello di codice intermedio. Non è infatti difficile rendersi conto che in tale codice intermedio (generato dal compilatore `clang++`) sono presenti quattro *Basic Block* che formano il *CFG* illustrato in figura.



4 LLVM-IR: istruzioni

Al livello delle istruzioni, LLVM-IR può essere visto, almeno in prima battuta, come una forma più ricca di linguaggio assembly, in cui è possibile ignorare una serie di aspetti che invece sono cruciali in un linguaggio assembly vero e proprio. Fra queste caratteristiche di cui possiamo non preoccuparci rientrano il numero di registri, le dimensioni dei tipi di dato e i già citati dettagli legati alla chiamata delle funzioni.

Come nel caso di un linguaggio assembly, l'instruction set di IR include istruzioni logico/artimetiche, istruzioni di salto condizionato/incondizionato, e di lettura da/scrittura in memoria. Le istruzioni logico/aritmetiche, oltre che naturalmente quelle di caricamento dalla memoria, scrivono il risultato in un registro. Tuttavia, al posto dei registri fisici, che sono necessariamente in numero fisso in una macchina reale, in LLVM-IR esiste un insieme illimitato di registri virtuali di tipo *SSA* (*Static Single Assignment*), e viene lasciato al back-end il compito di mappare i registri virtuali in registri fisici.

A scanso di qualche equivoco che potrebbe sorgere, va chiarito che *single assignment* è una qualifica spaziale e non temporale. In altri termini, e come suggerito dalla presenza dell'ggettivo *static*, un registro SSA può essere il left value di una sola istruzione. Se l'istruzione viene eseguita più volte, anche il registro viene scritto più volte. Si noti che, per default, i registri vengono identificati mediante numerazione successiva, che coinvolge anche le etichette dei `BB`. Il programmatore può tuttavia decidere di attribuire nomi simbolici che siano più esplicativi del ruolo ricoperto nella logica del programma.

Per quanto riguarda i dati, la ragione per cui il programmatore LLVM-IR non si deve preoccupare delle dimensioni è semplicemente perché i dati sono tutti "tipizzati". Ed è ancora una volta il back-end che si occupa di

mappare le informazioni di tipo sulle corrette dimensioni in memoria. Le istruzioni LLVM-IR specificano quindi il tipo di ogni dato trattato. Si possono definire tipi semplici di virtualmente ogni dimensione (es. `int1`, `int8`, `int64`, `float16`) come pure tipi con struttura, quali `array` e `struct`.

Il meccanismo di chiamata delle funzioni è un'altra caratteristica notevole di LLVM-IR, che in questo aspetto si avvicina molto di più ad un linguaggio ad alto livello che non ad un linguaggio assembly. Ma di questo abbiamo già parlato.

4.1 Esempi di istruzioni

Di seguito vediamo alcuni esempi di istruzioni, raggruppate per "tipologia funzionale". Negli esempi, `T` indica un generico tipo e, naturalmente, `T*` il tipo "puntatore ad un dato di tipo `T`". `N` invece indica un numero intero.

4.1.1 Istruzioni load e store

load

```
%value = load T, T* %ptr, align N
```

L'istruzione specifica le seguenti informazioni:

- Il registro virtuale `%value` dove verrà caricato il dato
- Il tipo `T` del dato caricato
- Il registro virtuale `%ptr` il cui contenuto, un puntatore a dati di tipo `T`, viene precisamente utilizzato come indirizzo di memoria per prelevare il dato di tipo `T` ivi memorizzato
- L'allineamento del dato in memoria (tipicamente 1, 4, 8)

store

```
store T %value, T* %ptr, align N
```

i cui elementi componenti hanno significato analogo.

4.1.2 Istruzioni aritmetico/logiche

add e mul

```
%sum = add nsw i32 %op1, %op2
%prod = mul nsw i32 %op1, %op2
```

Calcolano la somma/il prodotto dei due *signed integer*, di lunghezza 32 bit, memorizzati nei registri `%op1` e `%op2`. La presenza del flag `nsw`, che "sta per" *no signed wrap* richiede una digressione di una certa importanza, che vedremo subito dopo.

srem

```
%remainder = srem i32 %dividend, %divisor
```

Calcola il resto della divisione dei due *signed integer*, di lunghezza 32 bit, memorizzati nei registri `%dividend` e `%divisor`. La "s" iniziale del codice mnemonico indica appunto che gli operandi sono numeri con segno.

and

```
%and = and i32 %op1, %op2
```

Calcola l'and logico bit a bit dei due operandi, che devono essere dello stesso tipo (per avere esattamente lo stesso numero di bit). Ovviamente non c'è alcuna interpretazione numerica; se gli operandi sono numeri, che abbiano o non abbiamo segno è del tutto ininfluente.

Digressione. Il flag `nsw` prescrive che le operazioni producano un risultato indefinito nel caso si verifichi *overflow*. Questo stato di cose da un lato sembra ovvio, dall'altro però non pare essere spiegato dalla scelta del termine *no signed wrap*. Cerchiamo di capire meglio e, al riguardo, supponiamo di lavorare in aritmetica con segno su 8 soli bit. In tal caso, nella rappresentazione in complemento a 2 il range va da $-2^7 = -128$ fino a $2^7 - 1 = 127$. Le corrispondenti rappresentazioni binarie sono, rispettivamente, 1000000 e 01111111. Concentriamoci sull'addizione (solo perché piú semplice) e supponiamo che `%op1` contenga il valore 127 e `%op2` contenga il valore 1, che

nella rappresentazione binaria è chiaramente scritto come 00000001. Se effettuiamo la somma in binario otteniamo la sequenza 1000000, che corrisponde al numero -128 quando interpretata in complemento a 2. Si verifica appunto un "wrap", una sorta di "ri-avvolgimento" della rappresentazione per cui incrementare di 1 l'intero positivo massimo produce l'intero negativo minimo. Piaccia o no, questo è un comportamento ben definito e il compilatore non può presumere che il programmatore non lo sappia o che, addirittura, non ne faccia un qualche uso consapevole. Se invece il comportamento è dichiaratamente indefinito, allora è il programmatore a non essere legittimato a sfruttare il "wrap"; in questo e in diversi altri casi, il compilatore può addirittura "sfruttare" la situazione per eseguire ottimizzazioni di codice. Non andiamo oltre su questo punto perché si tratta di argomenti (le ottimizzazioni) relativi alla seconda parte del corso.

4.1.3 Confronti e "salti"

salto incondizionato

```
br label %ciclo
```

Qui `%ciclo` è l'etichetta di un *Basic Block*

confronto e salto

```
%res = icmp eq i32 %val, 0
br i1 %res, label %one, label %zero
```

La prima istruzione effettua il confronto con 0 del numero intero a 32 bit presente nel registro `%val`. Il risultato è di tipo `i1` (intero di un solo bit) e viene memorizzato nel registro `%res`. La seconda istruzione "testa" l'intero di un bit presente nel registro `%res` e passa il controllo alla prima istruzione del *Basic Block* di etichetta `%one` se il bit ha valore 1 e alla prima istruzione del *Basic Block* di etichetta `%zero` altrimenti.

4.1.4 Nodi phi

Un'istruzione fondamentale è la cosiddetta *istruzione phi*. Essa definisce, nel senso che sta all'inizio di, un vertice/nodo del *Control Flow Graph*, detto per questo nodo `phi`. In un tale nodo si riuniscono più possibili flussi di esecuzione. Consideriamo, il seguente frammento parziale di codice:

```
inizio:
 ...
ciclo:
 %n = phi i32, [0 %inizio], [%acc %ciclo]
 ...
 br label %ciclo
```

In questo caso l'istruzione `phi` prescrive che il valore da scrivere nel registro `%n` (un intero a 32 bit) sia 0, se il controllo proviene dal blocco etichettato `inizio`, e che sia invece il valore nel registro `%acc`, se proviene dal blocco etichettato `ciclo`, ovvero lo stesso blocco che include l'istruzione `phi`.

4.1.5 Allocazione sullo stack locale

L'istruzione `alloca` riserva spazio di memoria all'interno del frame sullo stack relativo alla funzione che la esegue. Ne consegue che lo spazio allocato viene automaticamente rilasciato nel momento in cui la funzione termina l'esecuzione. L'istruzione restituisce un puntatore del tipo di dato allocato. Vediamo un paio di esempi.

```
 %ptr = alloca double, align 8
 %ptr = alloca i32, i32 8, align 32
```

Nel primo caso viene allocato spazio per un numero floating point a 64 bit e restituito il corrispondente puntatore. Viene specificato che l'indirizzo di memoria deve essere multiplo di 8. Nel secondo caso viene invece allocato un "vettore" di 8 interi a 32 bit ad un indirizzo di memoria che deve essere multiplo di 32. Viene comunque restituito un puntatore ad intero a 32 bit.

4.1.6 Istruzione getelementptr

L'istruzione `getelementptr` serve per calcolare l'offset di un elemento in una struttura rispetto al puntatore alla struttura. L'istruzione, indicata spesso con `GEP`, esegue solo dei calcoli, a partire da dati in registri locali, senza alcun accesso alla memoria. Il caso più semplice riguarda ovviamente l'accesso ad una struttura lineare di cui, tipicamente, possediamo il puntatore al primo elemento. Consideriamo, ad esempio, il seguente codice `C`.

```
 int V = arr[0] + arr[2];
```

dove supponiamo che `arr` sia definito come struttura globale, ad esempio, mediante l'istruzione

```
int *arr = (int[]){10,20,30};
```

Il compilatore `clang` "traduce" gli accessi agli elementi `arr[0]` 3 `arr[2]` con il seguente codice LLVM IR.

```
%3 = load ptr, ptr @arr, align 8
%4 = getelementptr inbounds i32, ptr %3, i64 0
%5 = load i32, ptr %4, align 4
%6 = load ptr, ptr @arr, align 8
%7 = getelementptr inbounds i32, ptr %6, i64 2
%8 = load i32, ptr %7, align 4
```

La `GEP` specifica il tipo di dato all'indirizzo calcolato (`i32` in questo caso); gli altri due operandi sono l'indirizzo base (un puntatore) e l'offset (qui un intero `i64`, 0 nel primo caso e 2 nel secondo). Il resto del codice sopra riportato evidenzia il contesto tipico di utilizzo dell'istruzione. Nelle istruzioni `%2` e `%5` l'indirizzo base della struttura viene caricato nel registro omonimo[2] per essere poi utilizzato nella `GEP`. Le istruzioni `%4` e `%7` utilizzano l'indirizzo calcolato dalla `GEP` per prelevare il dato dalla memoria.

Se l'indirizzamento del dato richiede più offset (ad esempio, se la struttura è un array a più dimensioni), allora essi devono essere specificati tutti. Diversamente da quanto accade in `C`, LLVM richiede l'offset anche per il puntatore. Si consideri il seguente frammento di programma `C`.

```
int arr[] = {7,11,13};
int *ptr = arr;
int index_first(void) {
   int A = arr[2];
   int B = ptr[2];
   return A+B;
}
```

La funzione `index_first` accede allo stesso intero usando nomi diversi, `arr` e `ptr`, che però denotano lo stesso indirizzo di memoria. In `C` tutto ciò

[2]Si ricordi che il modello SSA consente di identificare ogni registro usato con l'istruzione che in quel registro scrive.

viene tradotto allo stesso modo, ovvero calcolando l'indirizzo dell'intero aggiungendo un offset all'indirizzo del primo elemento della struttura. In LLVM IR le due cose sono invece distinte. Vediamo il risultato della traduzione

```
@arr = dso_local global [3 x i32] [i32 7, i32 11, i32 13], align 4
@ptr = dso_local global ptr @arr, align 8

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @index_first() #0 {
   %1 = alloca i32, align 4
   %2 = alloca i32, align 4
   %3 = load i32, ptr getelementptr inbounds
                  ([3 x i32], ptr @arr, i64 0, i64 2), align 4
   store i32 %3, ptr %1, align 4
   %4 = load ptr, ptr @ptr, align 8
   %5 = getelementptr inbounds i32, ptr %4, i64 2
   %6 = load i32, ptr %5, align 4
   store i32 %6, ptr %2, align 4
   %7 = load i32, ptr %1, align 4
   %8 = load i32, ptr %2, align 4
   %9 = add nsw i32 %7, %8
   ret i32 %9
}
```

Si tenga presente, nel ragionamento che andremo ora a fare, che l'istruzione `%3` (una `load` che include in qualche modo la stessa GEP per il calcolo dell'indirizzo dell'operando) è del tutto equivalente alla seguente coppia di istruzioni:

```
%tmp = getelementptr inbounds[3 x i32], ptr @arr, i64 0, i64 2
%3 = load i32, ptr %tmp, align 4
```

Come si può vedere, il codice che implementa il calcolo dell'indirizzo dell'intero `arr[2]`, utilizzato nella successiva istruzione `load`, è diverso. Quando per l'accesso viene usata la variabile globale `arr` sono presenti due valori di offset (`i64 0` e i64 2). Invece, quando viene usata la variabile globale `ptr`, la `GEP` specifica un solo offset (i64 2). La ragione si evince dalle due definizioni globali: `@arr` è il puntatore ad un array di interi mentre `@ptr` è

un "semplice" puntatore ad intero. Quando si tratta di usare quest'ultimo, basta un offset. È come se l'istruzione `GEP` prescrivesse semplicemente di incrementare il puntatore di una quantià pari a 2 (2 interi). 2 non va cioè inteso come l'offset dell'elemento rispetto al puntatore bensì proprio come uno "spostamento" (implicito) del puntatore stesso. Se si comprende questo, si capisce perché nel primo caso ci sono due valori di offset e non uno solo. Il primo va infatti inteso come lo spostamento del puntatore e il secondo è l'offset. Riformulato in `C` è come se l'istruzione di accesso dovesse essere scritta come `arr[0][2]`.

In generale, l'accesso ad un elemento che si trova annidato a livello n in una struttura composta richiede la specifica di $n + 1$ offset, di cui il primo sempre riferito al puntatore.

4.1.7 Un esempio completo

Consideriamo il seguente esempio di programma `C++` che implementa l'algoritmo di Euclide per il calcolo del MCD di due numeri interi positivi.

```
int n = 15;
int m = 42;

int euclid(int n, int m) {
  int retval;
  if (m==0) retval = n;
  else retval = euclid(m,n%m);
  return retval;
}

int main() {
   return euclid(n,m);
}
```

Se trascuriamo un pò di metadati, il corrispondente codice IR prodotto dal compilatore `Clang ++` è illustrato in figura 1. Il comando utilizzato è il "solito":

```
clang++ -S emit-llvm euclid.cpp
```

Una rapida scorsa del codice permette di individuare alcune delle caratteristiche notevoli menzionate, come la tipizzazione dei dati e la semplicità del meccanismo di chiamata/ritorno.

```
@n = dso_local global i32 15, align 4
@m = dso_local global i32 42, align 4
define dso_local noundef i32 @_Z6euclidii(
    i32 noundef %0, i32 noundef %1) #0 {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    %5 = alloca i32, align 4
    store i32 %0, ptr %3, align 4
    store i32 %1, ptr %4, align 4
    %6 = load i32, ptr %4, align 4
    %7 = icmp eq i32 %6, 0
    br i1 %7, label %8, label %10

    8:                          ; preds = %2
    %9 = load i32, ptr %3, align 4
    store i32 %9, ptr %5, align 4
    br label %16

    10:                         ; preds = %2
    %11 = load i32, ptr %4, align 4
    %12 = load i32, ptr %3, align 4
    %13 = load i32, ptr %4, align 4
    %14 = srem i32 %12, %13
    %15 = call noundef i32 @_Z6euclidii(
         i32 noundef %11,  i32 noundef %14)
    store i32 %15, ptr %5, align 4
    br label %16

    16:                         ; preds=%10,%8
    %17 = load i32, ptr %5, align 4
    ret i32 %17
}
define dso_local noundef i32 @main() #1 {
    %1 = alloca i32, align 4
    store i32 0, ptr %1, align 4
    %2 = load i32, ptr @n, align 4
    %3 = load i32, ptr @m, align 4
    %4 = call noundef i32 @_Z6euclidii(
    i32 noundef %2, i32 noundef %3)
    ret i32 %4
}
```

Figure 1: Codice LLVM-IR per la funzione `C++ euclid.cpp`

5 Le API C++ di supporto alla generazione del codice

Per gli scopi del corso, la componente dell'intero sistema che più interessa è costituita da un insieme di classi C++ che innalzano notevolmente il livello di astrazione del processo di generazione di codice LLVM-IR. Nel seguito di questa sezione descriviamo le classi dell'infrastruttura LLVM che rivestono maggiore rilevanza per i nostri scopi.

LLVMContext. Un oggetto della classe `context` implementa una sorta di "contenitore" per le entità che definiscono lo `stato di esecuzione` del processo di compilazione. La classe include, fra gli altri, metodi per la definizione di tipi di dato e valori costanti. Si tenga infatti presente che LLVM IR è un linguaggio del tutto indipendente dal linguaggio del programma sorgente; in particolare, questo vuol dire che non possiede un insieme di tipi predefiniti. Se il programmatore (eventualmente tramite il front-end) richiede ad esempio la creazione di variabili di tipo "intero a 48 bit", il contesto ne tiene traccia per evitare duplicazioni nel caso di altre richieste di allocazioni simili. Per questa ragione l'allocazione di questi oggetti viene effettuata tramite una chiamata ad un metodo statico anziché tramite l'uso diretto di un costruttore.

Nella pratica, per la generazione di codice ci basterà definire una singola istanza della classe `LLVMContext` e utilizzare il puntatore a tale oggetto come argomento in tutti i metodi che lo richiedono. Nella stessa documentazione di LLVM tale classe è infatti spesso definita *opaca*, proprio nel senso che non è necessario conoscerne i dettagli.

```
LLVMContext *context = new LLVMContext
```

Metodi in cui è coinvolto il contesto

Tutti i metodi qui elencata sono metodi *statici*.

```
IntegerType::get(*context,32))
Type::getInt32Ty(*context)
Type::getDoubleTy(*context)
```

I primi due metodi restituiscono (una rappresentazione de) il tipo intero a 32 bit. Il terzo metodo restituisce il tipo `float` a precisione doppia.

```
ConstantFP::get(*context, APFloat(Val))
ConstantInt::get(*context, APInt(1,boolVal))
ConstantInt::get(*context, APInt(32,Val))
```

Metodi che restituiscono, rispettivamente, un `float` di valore `val`, un intero di un bit (interpretabile come valore logico) di valore `boolval` e in intero di 32 bit di valore `val`.

```
BasicBlock::Create(*context,"label")
BasicBlock::Create(*context, "entry", function);
```

Il primo metodo crea un `BB` non ancora "ancorato" in alcuna funzione. Il secondo metodo crea invece un `BB` e lo inserisce subito nella funzione `function`, come `entry block`.

```
FunctionType::get(IntegerType::get(*context,32),vec,false);
```

Questo metodo statico restituisce un tipo che può essere applicato ad una funzione. Il primo argomento, un intero su 32 bit, è il tipo di ritorno della funzione, mentre il secondo è un vettore di tipi, corrispondenti ai tipi dei parametri. L'ultimo argomento indica se la funzione ha un numero variabile di parametri (in questo caso no).

Module. I moduli sono le unità di compilazione. Essi contengono le definizioni di funzioni e la dichiarazione delle funzioni esterne. Possono anche contenere la definizione di altri "oggetti" globali, come variabili e tipi di dato. Come per il caso della classe `LLVMContext`, nel nostro codice ci limiteremo a creare separatamente ogni singola unità di compilazione e dunque utilizzeremo un solo oggetto della classe `Module`.

```
Module *module = new Module("Nome modulo", *context);
```

Metodi in cui è coinvolto il modulo

```
module->getFunction(fn)
```

Qui `module` si suppone essere un oggetto della classe `LLVMContext` creato dinamicamente come indicato sopra (altrimenti avremmo dovuto usare la notazione `module.getFunction(fn)`) Il ogni caso, il metodo `getFunction(fn)` cerca nel modulo la funzione `fn` e ne restituisce un puntatore al chiamante.

```
module->getNamedGlobal(Name)
```

Metodo di `Module` dal significato evidente. Ricerca il nome `name` in `module` e restituisce il puntatore al valore trovato (o il puntatore nulle se il nome non è presente)

```
Function *F = Function::Create(FT,
                 Function::ExternalLinkage,Name,*module)
```

Metodo statico che crea una funzione (nel modulo `*module`) con nome `Name`. La funzione ha *linkage* esterno, che significa semplicemente visibilità al di fuori del modulo (è cioè un nome che sarà noto al linker per la risoluzione di eventuali riferimenti esterni) e tipo specificato da `FT`. Il tipo di una funzione è essenzialmente una coppia costituita dal tipo di ritorno e da un vettore con il tipo dei parametri (si veda poco sopra).

```
G = new GlobalVariable(*module,
                       IntegerType::get(*context,32),
                       true,
                       GlobalValue::WeakAnyLinkage,
                       V,
                       Name)
```

Metodo che crea una variabile globale di tipo `i32`, nome `name`, e valore iniziale `V`. Quest'ultimo deve essere una costante LLVM, e dunque valutabile a tempo di compilazione. Il terzo argomento (qui `true`) indica che il valore è immutabile, e dunque che la variabile definita è in realtà una costante. Infine `WeakAnyLinkage` fornisce informazione al linker e indica che nella *linking unit* il nome può essere "sovrascritto" (overriden) da uno stesso nome esportato da una diversa unità di compilazione.

IRBuilder. Gli oggetti di questa classe vengono utilizzati per generare fisicamente le istruzioni LLVM IR. La più importante funzione svolta da un (oggetto) IRBuilder è di tenere traccia della posizione di inserimento del codice all'interno di un modulo, di una funzione o di un `Basic Block`. La classe dispone poi di un gran numero di metodi per l'inserimento fisico delle istruzioni nel codice.

Anche di questa classe utilizzeremo un solo oggetto perché la compilazione sarà strutturata come singolo processo (sequenziale). Tuttavia, per la generazione delle istruzioni di allocazione di memoria utilizzeremo builder temporanei.

```
IRBuilder<> *builder = new IRBuilder(*context);
```

Un oggetto della classe `IRBuilder` può essere pensato come un *file pointer* il cui stato include il punto attuale di scrittura nel file. La classe contiene quindi, oltre ai metodi per generare specifiche istruzioni, anche l'equivalente di metodi per determinare il punto di scrittura

Metodi in cui è coinvolto il builder

```
builder->SetInsertPoint(ExprBB)
```

Questo metodo posiziona il builder all'inizio del blocco `ExprBB`.

```
builder->GetInsertBlock()->getParent();
```

Questo codice consente di "risalire" dal builder alla funzione in cui è posizionato il attualmente il builder. Infatti, dapprima il metodo `GetInsertBlock()` restituisce il `BB` in cui effettivamente il builder sta scrivendo. Dopodichè, il metodo `getParent()` dei `BasicBlock` restituisce un puntatore alla funzione in cui il blocco è inserito. Questo consente alle differenti funzioni coinvolte nella sintesi del codice (come è il caso dei vari metodi `codegen` che compongono il nostro compilatore) di condividere il solo puntatore al builder, senza passare riferimenti a `BB` o a funzioni.

Il builder poi contiene tutti i metodi per generare le singole istruzioni. È un lungo elenco, anche limitandoci alle sole istruzioni che sono servite per costruire il compilatore del linguaggio `LFM`.

- `builder->CreateAlloca(T, size, ide)`
  alloca un'area di memoria sufficiente per memorizzare un singolo oggetto di tipo `T`, nel caso `size==nullptr`, o altrimenti un array di oggetti di tipo `T` il cui size è `size`. Si noti che il tipo di quest'ultimo

parametro non è semplicemente un intero (circostanza che sarebbe in contrasto con la possibilità di specificare `nullptr` come argomento), bensì il puntatore ad un oggetto della classe `ArraySize` che descrive precisamente la struttura. Restituisce il puntatore all'area allocata.

- `builder->CreateLoad(type,ptr,name)`,

  crea un'istruzione `load` che "preleva" il dato di tipo `type` dall'indirizzo di memoria puntato da `ptr` (a sua volta presumibilmente il risultato di un'istruzione `alloca`) e lo memorizza in un registro di nome `name`, eventualmente seguito da un suffisso numerico.

- `builder->CreateStore(reg, ptr);`,

  crea un'istruzione `store` che memorizza il valore contenuto nel registro `reg` nell'area di memoria puntata da `ptr`. Si noti che, a differenza di quanto accade per la `load`, qui non si deve specificare il tipo

- `builder->CreateNSWAdd(L,R,"sum")`

  `builder->CreateNSWSub(L,R,"diff")`

  `builder->CreateNSWMul(L,R,"prod")`

  `builder->CreateSDiv(L,R,"quot")`

  `builder->CreateSRem(L,R,"rem")`

  creano istruzioni per le cinque operazioni aritmetiche sugli interi. `L` ed `R` sono i registri dove sono memorizzati gli operandi, mentre in tutti i casi il terzo operando (opzionale) è il nome del registro in cui viene memorizzato il risultato.

- `builder->CreateICmpSLT(L,R,"cmplt")`

  `builder->CreateICmpSLE(L,R,"cmple")`

  `builder->CreateICmpSGT(L,R,"cmpgt")`

  `builder->CreateICmpSGE(L,R,"cmpge")`

  `builder->CreateICmpEQ(L,R,"cmpeq")`

  `builder->CreateICmpNE(L,R,"cmpne")`

  creano istruzioni per il confronto fra numeri interi con segno (nei primi quattro casi, ovvero quando questa caratteristica è rilevante). Gli operandi hanno lo stesso significato delle operazioni aritmetiche. Qui però il tipo del risultato è un $i1$ (un intero di 1 bit, interpretabile dunque come valore logico).

- `builder->CreateBr(label)`

  crea un'istruzione di salto incondizionato al `BasicBlock` di etichetta `label`

- `builder->CreateCondBr(V,trueBB,falseBB)`

  crea un'istruzione di salto al BasicBlock di etichetta `trueBB` o al BasicBlock di etichetta `falseBB` in base al valore contenuto nel registro `V`.

- `PHINode *PN = builder->CreatePHI(type,N,"val")`

  crea un cosiddetto nodo `phi`, ovvero un nodo nel *CFG* che sarà il target di N istruzioni di salto. Successivamente all'esecuzione di questa istruzione, devono infatti essere eseguite N istruzioni mediante le quali si aggiungono al nodo `phi` altrettante coppie $\langle$`valore`, `blocco`$\rangle$, in cui blocco è il `BasicBlock` di provenienza (il nodo nel *CFG* in cui è inserita la corrispondente istruzione di salto) e il valore è un registro, sempre dello stesso tipo indicato nella funzione di creazione del nodo. L'struzione per aggiungere le coppie al nodo è un metodo della classe `PHINode`: `PN->addIncoming(val,block)`

- `builder->CreateCall(FN, ArgsV, "call")`

  `builder->CreateRet(val)`,

  creano rispettivamente istruzioni di chiamata di una funzione e di ritorno dalla funzione. `FN` è il nome della funzione mentre `ArgV` è un vettore che specifica gli argomenti (più precisamente i registri SSA dove sono memorizzati gli argomenti). Anche `val`, nell'istruzione di ritorno, specifica un registro SSA.

Function. La classe `Function` ha lo scopo di modellare in LLVM IR le funzioni definite nel programma sorgente. Sono pochi i metodi della classe che utilizziamo nel front-end del linguaggio `lfm`.

- `function->end()`

  indica la fine della funzione, ovvero la posizione che segue l'ultima istruzione della funzione. Qui e nei metodi successivi `function` è il puntatore ad un oggetto della classe. Viene utilizzata per posizionare un `BasicBlock` come ultimo blocco della funzione (si veda il metodo successivo).

- `function->insert(function->end(), blocco)`
  posiziona il blocco `blocco` in fondo alla funzione.
- `function->getEntryBlock()`
  restituisce l'`entry block` della funzione, ovvero il blocco che segue immediatamente il prototipo della funzione.

Value. È la classe base per qualsiasi valore calcolato dal programma. In particolare, le classi che rappresentano le costanti (ad esempio `ConstantInt`), le funzioni, gli oggetti globali, le stesse istruzioni e i `BasicBlock` sono sottoclassi di `Value`. Per noi è soprattutto importante tenere presente che i metodi che generano codice restituiscono un puntatore ad un oggetto della classe `Value` o di una delle sue sottoclassi. Se l'istruzione scrive in un registro, per il fatto che istruzioni e registri SSA sono in corrispondenza biunivoca, può essere utile pensare che l'oggetto restituito denoti proprio il registro virtuale dove verrà memorizzato il risultato dell'istruzione.

Nel codice non utilizziamo metodi della classe. È però utile menzionare il fatto che, in certi casi, può essere necessario effettuare il cast di un oggetto della classe `Value`, che in realtà sappiamo appartenere ad una sottoclasse di `Value`, assegnandolo ad una variabile/puntatore di una classe più specifica.

Ci sono due tipi di cast, statico e dinamico, che possono essere effettuati su oggetti (o puntatori ad oggetti) di una gerarchia di classi. L'operatore `static_cast<>` fa sì che il compilatore esegua un controllo sulla legittimità della conversione in fase di compilazione. Questo avviene nei casi in cui si esegue il cast verso l'alto (da classe derivata a classe base) ovvero, ed è ciò che più ci interessa, verso il basso. Come detto, il controllo avviene solo in fase di compilazione e il compilatore presuppone che il programmatore sappia cosa sta facendo.

L'operatore `dynamic_cast<>` può invece essere utilizzato solo nel caso di un cast di puntatori o riferimenti e, oltre al controllo in fase di compilazione, esegue un controllo aggiuntivo in fase di esecuzione per verificare che il cast sia legale. Richiede però che la classe in questione abbia almeno un metodo virtuale, il che consente al compilatore (se supporta *Run-Time Type Information*, *RTTI*) di eseguire questo controllo aggiuntivo. Se la classe in questione non ha alcun metodo

virtuale, allora questo tipo di cast non può essere utilizzato. Nel caso di *downcasting*, che è quello che maggiormente interessa, il compilatore produrrà un messaggio di errore relativo al fatto che la classe base non è *polimorfica*.

Quest'ultimo è proprio il caso della classe `Value`, che "nonè polimorfica". Tuttavia, se sappiamo con certezza che un puntatore "punta" ad un oggetto di una sottoclasse di `Value`, possiamo usare `static_cast<>`.

C'è da aggiungere però che `LLVM` include un altro operatore di casting dinamico, che implementa una propria forma di *RTTI* e non richiede che la classe base includa metodi virtuali. Si tratta dell'operatore `dyn_cast<>`, il quale controlla che l'operando sia del tipo specificato e, in tal caso, restituisce un puntatore ad esso. Se l'operando non è del tipo corretto, viene restituito un puntatore `null`.

Per fare due esempi conclusivi, se sappiamo che (il puntatore `Val` ad) un oggetto della classe `Value` è in realtà un `ConstantInt`, possiamo utilizzare una delle seguenti istruzioni di conversione:

```
ConstantInt* V = dyn_cast<ConstantInt>(Val);
```

ovvero

```
ConstantInt* V = static_cast<ConstantInt*>(Val);
```