Compilatori

Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2024/2025

1. Generazione di codice intermedio LLVM
   - Cenni su LLVM-IR
   - API LLVM per la generazione della IR

Compilatori

Rappresentazione intermedia LLVM

- Per completare il progetto del front-end, dobbiamo ora concentrarci sulla produzione di codice secondo un modello di *rappresentazione intermedia* (*IR*) esistente
- La nostra scelta è ricaduta su IR di *LLVM* (originariamente acronimo di *Low Level Virtual Machine*)
- In realtà LLVM è oggi un insieme di strumenti per la costruzione di compilatori
- Nella seconda parte del corso studierete, in particolare, le ottimizzazioni effettuate dal cosiddetto *middle-end* su IR LLVM
- IR LLVM (da ora in avanti semplicemente IR) è dunque non solo l'output del front-end, ma anche il "testimone" passaggio fra le due parti del corso
- Inizieremo noi a fornirne una descrizione sommaria, che verrà affinata nella seconda parte

Tre differenti forme per la IR di LLVM

- Il codice IR può servire per tre scopi differenti e, di conseguenza, essere presentato in tre diverse forme, tutte equivalenti
  - come rappresentazione intermedia in memoria, manipolata dal compilatore per l'analisi e l'ottimizzazione
  - come rappresentazione bitcode su disco, caricabile per una compilazione Just-In-Time
  - come linguaggio assembly human-readable.
- Nei file su disco, per la versione human-readable si usa l'estensione `.ll` mentre per quella in bit-code si usa `.bc`
- Come brevemente vedremo, il codice IR ha istruzioni assembly ma nel complesso ha una struttura generale relativamente di alto livello

Il modello di calcolo

- Il modello di calcolo di riferimento per IR è una macchina con un numero illimitato di registri (*URM*, *Unlimited Register Machine*).
- A questo sono poi associate due specifiche caratteristiche.
  - I registri sono del tipo *SSA* (*Static Single Assignment*)
  - I valori assegnati ad un registro sono tipizzati.
- SSA significa che, "staticamente" (ovvero nel codice scritto), l'assegnamento al registro viene effettuato da una sola istruzione
- Si dice anche che l'istruzione *definisce* il registro
- A tempo di esecuzione (cioè *dinamicamente*), un registro può invece essere riscritto più volte, se l'istruzione che lo definisce viene ri-eseguita

Istruzione di IR LLVM

- IR può essere visto come il set di istruzioni di un'architettura *RISC*
- Ad eccezione delle istruzioni `LOAD` e `STORE`, che operano trasferimenti da e verso la memoria, tutte le altre istruzioni coinvolgono solo registri virtuali
- IR include istruzioni aritmetico/logiche, di salto (condizionato o incondizionato), di aritmetica dei puntatori, di accesso alla memoria e di controllo del flusso

Il grafo di controllo del flusso

- Un programma in IR è composto da uno o più *moduli*, ognuno dei quali è un file (in formato assembly o bitcode, come visto)
- Un modulo è composto dalla definizione di variabili globali e funzioni, a loro volta formate da una serie di "porzioni" di codice dette *blocchi di base* (*Basic Block, BB*)
- Ogni BB inizia con un'etichetta e termina con un'istruzione di salto o una return e non include altre istruzioni che trasferiscono il controllo (con la sola possibilità di chiamate di funzione)
- Ogni blocco può essere visto come nodo di un grafo, detto *grafo di controllo* del flusso di esecuzione (*Control Flow Graph, CFG*), di cui istruzioni di salto ed etichette individuano gli *archi*
- Una specifica istruzione LLVM, detta *istruzione phi*

```
phi tipo [val1, pred1], [val2, pred2], ...
```

consente di "sfruttare" la conoscenza del blocco di provenienza in quanto il suo valore sarà `val`i sse il flusso di esecuzione procede dal

Uno sguardo diretto a IR compilando un semplice esempio

- Consideriamo la seguente semplice funzione `fact`, che supponiamo memorizzata nel file `fact.cpp`

```
#include "llvm/IR/LLVMContext.h"

int fact(int n) {
        if (n==0) return 1;
        else return n*fact(n-1);
}
```

- Per compilare il file in IR human-readable format possiamo dare il seguente comando

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

Il file fact.ll

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 %0, i32* %3, align 4
   %4 = load i32, i32* %3, align 4
   %5 = icmp eq i32 %4, 0
   br i1 %5, label %6, label %7

6:                                                ; preds = %1
   store i32 1, i32* %2, align 4
   br label %13
```

Il file fact.ll

```
  7:                                                ; preds = %1
    %8 = load i32, i32* %3, align 4
    %9 = load i32, i32* %3, align 4
    %10 = sub nsw i32 %9, 1
    %11 = call noundef i32 @_Z4facti(i32 noundef %10)
    %12 = mul nsw i32 %8, %11
    store i32 %12, i32* %2, align 4
    br label %13
  13:                                               ; preds = %7, %
    %14 = load i32, i32* %2, align 4
    ret i32 %14
}
```

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 %0, i32* %3, align 4
   %4 = load i32, i32* %3, align 4
   %5 = icmp eq i32 %4, 0
   br i1 %5, label %6, label %7

6:                                                ; preds = %1
   store i32 1, i32* %2, align 4
   br label %13
```

Il simbolo @ denota un nome globale (di variabile o funzione)

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  %4 = load i32, i32* %3, align 4
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %7

6:                                                ; preds = %1
  store i32 1, i32* %2, align 4
  br label %13
```

Blocchi di base (BB) e nodi del CFG

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 %0, i32* %3, align 4
   %4 = load i32, i32* %3, align 4
   %5 = icmp eq i32 %4, 0
   br i1 %5, label %6, label %7

6:
   store i32 1, i32* %2, align 4
   br label %13
```

Individua l'arco entrante del CFG

```
; preds = %1
```

Osservazioni

```
7:                                                ; preds = %1
  %8 = load i32, i32* %3, align 4
  %9 = load i32, i32* %3, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call noundef i32 @_Z4facti(i32 noundef %10)
  %12 = mul nsw i32 %8, %11
  store i32 %12, i32* %2, align 4
  br label %13
13:                                               ; preds = %7, %6
  %14 = load i32, i32* %2, align 4
  ret i32 %14
}
```

Due altri BB del programma per il fattoriale

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  %4 = load i32, i32* %3, align 4
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %7

6:                                                ; preds = %1
  store i32 1, i32* %2, align 4
  br label %13
```

Registri virtuali “anonimi”
La numerazione è automatica da parte di CLANG
Un programmatore può definirli assegnando un nome simbolico
Il simbolo % denota nome locale

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  %4 = load i32, i32* %3, align 4
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %7

6:                                                ; preds = %1
  store i32 1, i32* %2, align 4
  br label %13
```

Alcune istruzioni assegnano un valore ad un registro virtuale
Si dice che l'istruzione "definisce" il registro
I registri non possono essere definiti più volte
=> un registro non può essere riscritto

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 %0, i32* %3, align 4
   %4 = load i32, i32* %3, align 4
   %5 = icmp eq i32 %4, 0
   br i1 %5, label %6, label %7

6:                                                ; preds = %1
   store i32 1, i32* %2, align 4
   br label %13
```

Registri e operazioni sono tipizzati

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 %0, i32* %3, align 4
   %4 = load i32, i32* %3, align 4
   %5 = icmp eq i32 %4, 0
   br i1 %5, label %6, label %7

6:                                                ; preds = %1
   store i32 1, i32* %2, align 4
   br label %13
```

L'istruzione alloca riserva spazio sullo stack e restituisce un puntatore
A differenza dei registri, la memoria è ovviamente riscrivibile

Osservazioni

```
7:                                                ; preds = %1
  %8 = load i32, i32* %3, align 4
  %9 = load i32, i32* %3, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call noundef i32 @_Z4facti(i32 noundef %10)
  %12 = mul nsw i32 %8, %11
  store i32 %12, i32* %2, align 4
  br label %13
13:                                               ; preds = %7, %6
  %14 = load i32, i32* %2, align 4
  ret i32 %14
}
```

Attributi/flag di dati e operazioni
(NSW = No Signed Wrap)

Osservazioni

```
7:                                                ; preds = %1
  %8 = load i32, i32* %3, align 4
  %9 = load i32, i32* %3, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call noundef i32 @_Z4facti(i32 noundef %10)
  %12 = mul nsw i32 %8, %11
  store i32 %12, i32* %2, align 4
  br label %13
13:                                               ; preds = %7, %6
  %14 = load i32, i32* %2, align 4
  ret i32 %14
}
```

Si può notare come la chiamata di funzione sia un costrutto di alto livello rispetto ad un linguaggio assembly

Osservazioni sull'esempio: i moduli LLVM

- Il "programma" che stiamo analizzando è composto da più blocchi di base ma da un solo *modulo*
- Un generico programma può però essere composto da più moduli che, compilati separatamente, vengono poi uniti nel processo di *linking*
- Il nostro modulo non è ovviamente eseguibile e deve essere "linkato" ad un modulo dotato di `main` program
- Un modulo contiene tutti i dati che compongono un singolo file di programma
- In generale un modulo contiene dichiariazioni di *variabili globali*, *funzioni* (dichiarazioni per esterne, dichiarazioni e definizioni per interne), *dichiarazioni di tipo*, ...
- Il programma della seguente slide può risultare utile a chi sia interessato a fare qualche semplice prova

Programma che stampa i numeri da 1 a 10

```
@fmt = constant [4 x i8] c"%d\0A\00"
declare i32 @printf(i8*, i32)

define i32 @main() {
init:
   %counter = alloca i32
   store i32 0, i32* %counter
   br label %loop
loop:
   %currval = load i32, i32* %counter
   %nextval = add i32 %currval, 1
   %end = icmp eq i32 %nextval, 11
   br i1 %end, label %exit, label %cont
```

Programma che stampa i numeri da 1 a 10

```
cont:
   store i32 %nextval, i32* %counter
   %ptr = getelementptr [4 x i8], [4 x i8]* @fmt, i32 0, i32 0
   call i32(i8*,i32) @printf (i8* %ptr, i32 %nextval)
   br label %loop
exit:
   ret i32 0
}
```

NB La costante globale `@fmt` è una stringa di formato, usata come primo argomento per la funzione esterna `@printf`. Essa specifica la stampa di un numero intero (carattere d) seguito da un newline (codice ascii 10).

Compilazione ed esecuzione

- Il precedente programma (che chiameremo `easyprint.ll`) può essere eseguito in (almeno) due modi differenti
- Il modo più veloce usa l'interprete e compilatore JIT `lli`

```
> lli easyprint.ll
```

- Il secondo modo utilizza lo stesso compilatore clang che completa la fase di compilazione (middle-end, back-end, linkink)

```
> clang -o easyprint easyprint.ll
```

Compilatori

L'infrastruttura LLVM

- L'infrastruttura LLVM è scritta in linguaggio C++
- Essa include diverse classi che rendono la generazione della IR un procedimento "sufficientemente" agevole
- La buona notizia è che non è necessario conoscere i dettagli delle istruzioni né avere una chiara "mappa" della loro collocazione fisica nel programma (nel suo insieme)
- In questa seconda parte discuteremo le classi che utilizzeremo nel nostro front-end:
  - `LLVMContext`
  - `Module`
  - `Function`
  - `BasicBlock`
  - `IRBuilder`
  - `Value`

La classe `LLVMContext`

- Viene definita *opaca* perché, nonostante la classe includa strutture dati cruciali per il funzionamento di LLVM, non è in generale necessario avere di essa alcuna conoscenza in ordine alla generazione di IR
- Per i nostri scopi sarà sufficiente creare una singola istanza della classe e "passare" il relativo puntatore alle funzioni che ne fanno uso
- Il contesto è specialmente importante quando si tratta di definire il *tipo* di una variabile
- Si tenga presente che IR è indipendente dal linguaggio sorgente e dunque non possiede un insieme di tipi predefiniti, ad esempio non possiede interi di lunghezze prefissate
- Se viene richiesta l'allocazione di, poniamo, variabili intere di 8, 16 e 96 bit, il contesto ne tiene traccia per evitare duplicazioni nel caso di altre richieste di allocazioni simili

La classe `Module`

- La classe `Module` rappresenta il modulo, come lo abbiamo definito in precedenza, ovvero un insieme di definizioni di variabili globali e di funzioni
- Anche della classe `Module` sarà sufficiente disporre di una sola istanza per ogni file sorgente
- Chiaramente il modulo espone metodi per l'accesso ai componenti
- Un metodo della classe che useremo è `getFunction` che, dato il nome di una funzione, restituisce un puntatore all'oggetto che la rappresenta
- Nell'istanza di `Module` si potrebbero poi inserire le informazioni relative all'architettura, che però noi ignoriamo perché ci fermiamo alla sola produzione dell'IR in formato human-readable

La classe `Function`

- Gli oggetti della classe `Function` sono chiaramente la controparte in IR delle funzioni definite nel programma sorgente
- I metodi della classe che utilizzeremo sono:
  - `getEntryBlock()`, che restituisce un puntatore al BB iniziale della funzione
  - `insert()`, che inserisce un BB in un determinato punto della funzione
  - `end()`, da utilizzare tipicamente con `insert()`, che indica la (attuale) fine del body della funzione
  - `eraseFromParent()`, che cancella una definizione (parziale) non completata con successo

La classe `BasicBlock`

- Gli oggetti della classe `BasicBlock` (BB) sono gli effettivi "contenitori" del codice
- Un BB è dunque sempre un argomento nei metodi che generano IR
- Ad ogni oggetto BB può essere (ed in genere è) attribuita un'etichetta, da utilizzare come riferimento per istruzioni di salto
- Come abbiamo già sottolineato, i BB sono i nodi del CFG e giocano un ruolo fondamentale nella pianificazione della sintesi di codice
- Si pensi al semplice caso di un'istruzione condizionale a due vie
- In tal caso, il generatore del codice dovrà creare (nel BB corrente) le istruzioni per eseguire il test e dovrà poi generare tre BB per le due possibili vie del condizionale e per l'inserimento delle istruzioni successive (dove il flusso si "riunisce")

La classe `IRBuilder`

- Un unico oggetto della classe è sufficiente, il cui scopo è tenere traccia del punto di inserzione del codice e di generare, attraverso i suoi numerosi metodi, tutte le istruzione del set
- Esempi di metodi che generano singole istruzioni sono
  - `CreateLoad()`, `CreateStore`, istruzioni load e store
  - `CreateNSWNeg()`, meno unario
  - `CreateNSWAdd()`, `CreateNSWSub()`, `...`, operazioni aritmetiche
  - `CreateICmpULE()`, `CreateICmpULT()`, , confronti fra numeri interi
  - `CreateCall()`, chiamata di funzione
  - `CreateBr()`, `CreateCondBr()`, salto incondizionato e condizionato
  - `CreatePHI()`, istruzione `phi`
  - `CreateRet()`, istruzione `return`
  - `SetInsertPoint()`, stabilisce il punto di inserimento delle istruzioni
  - `GetInsertBlock()`, restituisce il blocco dove vengono correntemente inserite le istruzioni

La classe `Value`

- `Value` è la classe base per qualsiasi valore calcolato dal programma
- `Function` e `BasicBlock`, ad esempio, sono sottoclassi di `Value`
- I metodi che generano codice per ogni costrutto del linguaggio restituiscono un puntatore ad un oggetto della classe `Value`
- Si può pensare che tale oggetto denoti il *registro virtuale* dove verrà memorizzato il risultato della computazione rappresentata da "quel" codice