Compilatori
Parte Due

Iacopo Ruzzier

Ultimo aggiornamento: 5 marzo 2025

Indice

Parte I
lab 1

1 la IR di LLVM

ricordiamo: IR di llvm ha sintassi e semantica simili all'assembly a cui siamo abituati
domanda: come scrivere un passo llvm?
prima chiariamo alcuni punti:

- moduli llvm
- iteratori
- downcasting
- interfacce dei passi llvm

1.1 moduli llvm

un modulo rappresenta un singolo file sorgente (corrisponde) - vedremo che gli iteratori permettono di "scorrere" attraverso tutte le funzioni di un modulo
recupera slide 23 bene

1.2 iteratori

nota su cambiamento a nuovo llvm pass manager (vedi link a slide 22)
vediamo che in generale un iteratore permette di puntare al "livello sottostante" della gerarchia appena vista
nota: sintassi simile a quella del container STL `vector`
IMG slide 25
caption: l'attraversamento delle strutture dati della IR llvm normalmente avviene tramite doubly-linked lists

1.3 downcasting

tecnica che permette di istanziare ... recupera
omtivazione: es. capire che tipo di istruzione abbiamo davanti → il downcasting aiuta a recuperare maggiore informazione dagli iteratori
uso esempio del downcasting dunque: specializzare l'estrazione di informazione durante il pass (?)

1.4 interfacce dei passi llvm

llvm fornisce già interfacce diverse:

- basicblockpass: itera su bb
- callgraphsccpass: itera sui nodi del cg
- functionpass: itera sulla lista di funzioni del modulo
- eccetera

diversificate appositamente per passi con intenzione diversa: permette di scegliere a che "grana" opera il pass di ottimizzazione (di che livello di informazione ho bisogno? magari non mi serve essere a livello di modulo ma direttamente a livello di ad es. loop)

1.5 new pass manager

solitamente ha una pipeline "statica" (predefinita) di passi → alterabile invocando una sequenza arbitraria tramite cmd line: `opt -passes='pass1,pass2' /tmp/a.ll -S` o `opt -p pass1, ...`

2 esercizio 1 - IR e CFG

- per ognuno dei test benchmarks produrre la IR con clang e analizzarla, cercando di capire cosa significa ogni parte
- disegnare il CFG per ogni funzione

usiamo clang per produrre la IR da dare in pasto al middle-end:

```
clang -O2 -emit-llvm -S -c test/Loop.c -o test/Loop.ll
```

oppure prima produco bytecode e poi disassemblo per produrre la forma assembly

```
clang -O2 -emit-llvm -c test/Loop.c -o test/Loop.bc
llvm-dis test/Loop.bc -o=./test/Loop.ll
```