

Compilatori

Parte Due

Iacopo Ruzzier

Ultimo aggiornamento: 11 marzo 2025

Indice

I	lab 1	2
1	la IR di LLVM	2
1.1	moduli llvm	2
1.2	iteratori	2
1.3	downcasting	2
1.4	interfacce dei passi llvm	2
1.5	new pass manager	2
2	esercizio 1 - IR e CFG	3
3	esercizio 2 - TestPass	3

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

motivazione: 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
- `callgraphscppass`: 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
```

3 esercizio 2 - TestPass

in questo corso scriveremo i passi di analisi e ottimizzazione come **plugin** per il pass manager di llvm - modo valido e conveniente per scrivere passi, evitando di dover ricompilare ogni volta llvm (andremo ad usare l'interfaccia plugin appunto, che ci consente uno sviluppo esterno al build tree di llvm - il compilato viene poi linkato come libreria dinamica)

istruzioni :

- crea un workspace con root dir es. `mkdir lab_compilatori && export ROOT_LABS=/path/to/lab_compilatori`
- scarica i file di lab 1 in `ROOT_LABS/Lab1`
- prova a settare l'env e compilare:

```
export LLVM_DIR=<installation/dir/of/llvm/19>
mkdir build
cd build
cmake -DLT_LLVM_INSTALL_DIR=dollarsignLLVM_DIR
source/dir/test/pass>/
make
```

tua posizione di llvm: `/opt/homebrew/opt/llvm`

- vedi script `setup.sh` (che prevede cartella build già fatta in precedenza, altrimenti aggiungi `mkdir` per come buildare il passo, o slide 41)
- inserisci cartella `test` con loop e fibonacci
- a questo punto invoca l'ottimizzatore `opt` con il flag di override del default pass manager:

```
opt -load-pass-plugin <path/to/build/dir>/LibTestPass.so -passes
    =test-pass test/Loop.bc -o test/LoopTestPass.bc
# .dylib e non .so se su MacOS
# .ll invece di .bc a seconda di quello su cui sto lavorando
```

- `load-pass-plugin` per caricare il plugin appena buildato
- `passes=test-pass` oppure `-p test-pass` per inserire il nuovo pass da noi creato
- al momento non stiamo ottimizzando ma solo analizzando, quindi posso anche sostituire il `-o` con `-disable-output`

l'esercizio prevede di estendere il passo TestPass, di modo che analizzi la IR e stampi alcune informazioni utili per ciascuna delle funzioni che compaiono nel programma di test:

1. nome
2. numero argomenti (N+* in caso di funzione variadica, vedi slide 46)
3. numero chiamate a funzione nello stesso modulo
4. numero BB
5. numero Istruzioni