Compilatori
Parte Due

Iacopo Ruzzier

Ultimo aggiornamento: 4 aprile 2025

Indice

1 Introduzione (25 feb)

1.1 Motivazione

Ricordiamo il ruolo del compilatore tra le tecnologie informatiche, quello dell'ISA e del linguaggio assembly, i passaggi gestiti dal compilatore, dall'assembler, eccetera

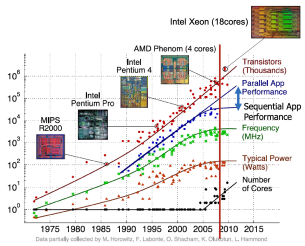
- Il compilatore **traduce un programma sorgente in linguaggio macchina**
- L'ISA agisce da "interfaccia" tra HW e SW (fornisce a SW il set di istruzioni, e specifica a HW che cosa fanno)

1.1.1 La funzione dei compilatori

- Funzione principale e più nota: trasformare il codice **da un linguaggio ad un altro** (es. C → Assembly RISC-V) (ricordiamo che è solo il primo passo di un'intera toolchain di programmi per creare eseguibili)
- Gestendo la traduzione a linguaggio macchina al posto dei programmatori, l'altra funzione importante è l'**ottimizzazione** del codice, che permette la **produzione di eseguibili di stesse funzionalità**, ma diversi a livello di **dimensioni** (es. per sistemi embedded e high-performance), **consumo energetico**, **velocità di esecuzione**, ma anche in termini di determinate **caratteristiche architetturali** utilizzate (es. proc. multicore)

1.1.2 L'evoluzione dei compilatori

Le rivoluzioni in termini di "classe" di dispositivi e di dimensioni dei transistor sono molto frequenti (Bell, Moore), e nei primi 2000 si arriva ai **limiti fisici della miniaturizzazione e della frequenza** operativa dei processori (problemi di dissipazione del calore) → idea di cambiare il paradigma di sviluppo di un processore: dal singolo core sempre più potente passo a **più core "isopotenti"** sullo stesso chip



∼ 2005: plateau di consumo, frequenza e performance di programmi *sequenziali*, aumento di performance di p. che **sfruttano la parallelizzazione** → i programmi devono essere "consapevoli" che il processore è multicore!
Il compilatore mantiene un ruolo fondamentale: oltre a rendere meno "traumatico" il passaggio alla programmazione parallela, (non sono ancora auto-parallelizzanti) si interfaccia con i nuovi paradigmi di programmazione parallela offerti ai programmatori: il programmatore sfrutta interfacce semplici e astratte, mentre il compilatore traduce i costrutti in codice parallelo eseguibile (es. OpenMP)

1.1.3 Eterogeneità architetturale

La programmazione parallela e il parallelismo architetturale sono oggi paradigmi consolidati, e i processori general purpose (seppur multicore e ottimizzati) non sono sufficienti per attività specializzate come la grafica → nascono componenti **acceleratori** di vario tipo: GPU, GPGPU, FPGA, TPU, NPU... Questo complica ulteriormente la scrittura del software, e dunque impone altre evoluzioni nei compilatori e nelle ottimizzazioni.

1.2 Ottimizzazione

Ricordiamo le metriche usate:

$$\text{Performance} = \frac{1}{\text{Execution Time}} \qquad \text{Execution Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Frequency}}$$

Le ottimizzazioni possono avvenire dal punto di vista HW (parametri architetturali) e da quello SW (p. di programma). Il compilatore può agire anche ad es. a livello di cache, aiutando a ridurre i miss e dunque i CPI delle istruzioni `load` e `store` (sappiamo che il costo di accesso aumenta di ordini di grandezza)

1.2.1 Esempi di ottimizzazione

Distinguiamo le ottimizzazioni che avvengono a compile time o a runtime (statiche o dinamiche)

- **AS (Algebraic Semplification)**Semplification: ottimizzazione a runtime

```
-(-i); → i;
b or true; → true; //cortocircuito logico
```

- **CF (Constant Folding)**: valutare ed espandere espressioni costanti a compile time

```
c = 1+3; → c = 4;
(100<0) → false
```

- **SR (Strength Reduction)**: sostituisco op. costose con altre più semplici: classico es. `MUL` rimpiazzate da `ADD/SHIFT` (esecuzione in 1 ciclo invece di multic.):

```
y = x*2;
y = x * 17;
```
→
```
y = x+x;
y = (x<<4) + x;
```

es. sofisticato: `for` con operazioni su array, sostituito da operazioni su puntatori (aritmetica dei pt.) → il risultato si vede nel codice assembly

```
for (i=0; i<100; i++)
  a[i] = i*100;
```
→
```
t = 0;
for (; t<10000; t += 100) {
  *a = t;
  a = a + 4;
}
```

```
li s0, 0 // i = 0
li s1, 100
LOOP:
bge s0, s1, EXIT
slli s2, s0, 2
add s2, s2, a0
mul s3, s0, 100
sw s3, 0(s2)
addi s0, s0, 1
jal zero, LOOP
EXIT:
```
→
```
li s0, 0 // t = 0
li s1, 10000
LOOP:
bge s0, s1, EXIT
sw s0, 0(a0)
addi a0, a0, 4
jal zero, LOOP
EXIT:
```

- **CSE (Common Subexpression Elimination)**: elimino i calcoli ridondanti di una stessa espressione riutilizzata in più istruzioni (statement)

```
y = b * c + 4
z = b * c - 1
```
→
```
x = b * c
y = x + 4
z = x - 1
```

- **DCE (Dead Code Elimination)**: elimino tutte le istruzioni che producono codice mai letto (e dunque utilizzato), es. variabili assegnate e mai lette, codice irraggiungibile → uno dei passi eseguiti più di frequente durante l'ottimizzazione del codice da parte del compilatore, per rimuovere anche tutto il dead code generato dagli altri passi di ottimizzazione

```
b = 3
c = 1 + 3
d = 3 + c
```
→
```
c = 1 + 3
d = 3 + c
```

```
if (100<0)
{a = 5}
```
→
```
if (false)
{}
```
→

- **Copy Propagation**: per uno statement `x = y`, sostituisco gli usi futuri di `x` con `y` se non sono cambiati nel frattempo (propedeutico alla DCE)

```
x = y;
c = 1 + x;
d = y + c;
```
→
```
x = y;
c = 1 + y;
d = y + c;
```
$\xrightarrow{\text{DCE}}$
```
c = 1 + y;
d = y + c;
```

- **CP (Constant Propagation)**: sostituisco usi futuri di una variabile con assegnato valore costante con la costante stessa (se la variabile non cambia) (sempre ipotesi che i valori a fine es. siano poi **usati**, e non dead code)

```
b = 3;
c = 1 + b;
d = b + c;
```
$\xrightarrow{\text{CP}}$
```
b = 3;
c = 1 + 3;
d = 3 + c;
```
$\xrightarrow{\text{CF}}$
```
b = 3;
c = 4;
d = 3 + c;
```
$\xrightarrow{\text{CP}}$
```
b = 3;
c = 4;
d = 3 + 4;
```
$\xrightarrow{\text{CF}}$
```
b = 3;
c = 4;
d = 7;
```
$\xrightarrow{\text{DCE}}$
```
d = 7;
```

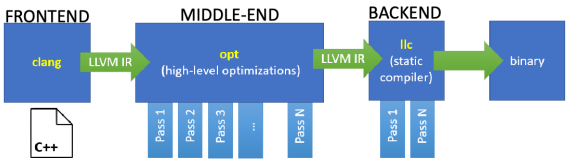
- LICM (Loop Invariant Code Motion): si occupa di muovere fuori dai loop tutto il codice **loop invariant**; evita i calcoli ridondanti

```
while (i<100) {
  *p = x/y + i;
  i = i + 1;
}
```
→
```
t = x + y;
while (i < 100) {
  *p = t + i;
  i = i + 1;
}
```

1.2.2 Ottimizzazioni sui loop

- grande impatto sulla performance dell'intero programma (per ovvie ragioni)
- spesso sono ottimizzazioni propedeutiche a quelle machine-specific (effettuate nel backend): register allocation, instruction level parallelism, data parallelism, data-cache locality

1.3 Anatomia di un compilatore



- almeno due compiti: **analisi del sorgente** e **sintesi di un programma in linguaggio macchina**, operando su una IR che si interpone tra frontend e backend, e tra source code e target code
- Il blocco di middle-end agisce su IR, e in vari passaggi lo trasforma e ottimizza ($\neq$ a seconda del compilatore)
- caso llvm: `clang` (frontend) → `opt` (middleend) → `llc` (backend)
- `opt` si basa su una serie di **passi di ottimizzazione (o di analisi)**: un passo di analisi scorre l'IR e lo analizza (non lo trasforma, ma produce informazioni utili); un passo di ottimizzazione sfrutta informazioni conosciute per trasformare l'IR (applica le ottimizzazione)
- alcune ottimizzazioni non possono essere effettuate o finalizzate senza conoscere l'architettura target (es. sulle cache), e dunque vengono eseguite dal backend

1.3.1 Flag di ottimizzazione

sono flag che passo al compilatore (al pass manager) per influenzare **ordine e numero dei passi di ottimizzazione**

- `-g`: solo debugging, nessuna ottimizzazione
- `-O0`: nessuna ottimizzazione
- `-O1`: solo ott. semplici
- `-O2`: ott. più aggressive
- `-O3`: ordine dei passi che sfrutta compromessi tra velocità e spazio occupato
- `-Os`: ottimizza per dimensione del compilato

1.3.2 Uso di IR

un backend che fa uso di IR permette di disaccoppiare con facilità frontend e backend, lavorare su ottimizzazioni machine-independent, semplificare il supporto per molti linguaggi, eccetera

Per supportare un nuovo linguaggio o una nuova architettura, basta scrivere un nuovo front/backend - il middle-end può rimanere lo stesso!

1.3.3 Ingredienti dell'ottimizzazione

- **formulare un problema di ottimizzazione** con molti casi di applicazione, sufficientemente efficiente e impattante su parti significative

→ **rappresentazione** che astrae dettagli rilevanti → **analisi** di applicabilità → **trasformazione del codice** → **testing** → ↻

2 Rappresentazione intermedia (4 mar)

Ricordiamo: middle end come sequenza di passi, di analisi o di trasformazione → per analizzare e trasformare il codice occorre una rappr. intermedia (IR) **espressiva** che **mantenga le informazioni importanti da un passo all'altro**

2.1 Proprietà di una IR

scegliamo IR diverse a seconda del loro uso, in generale alcune caratteristiche sono sempre richieste:

- facilità di **generazione** (effetti sul frontend)
- facilità e costo di **manipolazione**
- livello di astrazione e di **dettaglio esposto**: effetti su frontend e backend ($\neq$ IR da un lato e dall'altro, a seconda di astraz. e dettaglio necessari)

2.2 Tipi di IR

- AST (abstract syntax tree)
- DAG (grafi diretti aciclici)
- 3AC (3-address code): simile all'assembly (3 indirizzi: registro destinazione e max 2 operandi)
- SSA (Static Single Assignment): evoluzione di 3ac con ulteriori proprietà di control flow
- CFG (control flow graphs): rappresenta "come" vengono chiamate le funzioni (a partire dal main)
- CG (call graph)
- PDG (program dependence graphs): fondamentale per lavorare sul parallelismo, multithreading...

Le ott. inter-procedurali devono per forza basarsi su IR di tipo CG (es. per decidere quando fare **inlining** - espandere il codice della funzione invece di chiamarla - evidente tradeoff tra dimensione del codice e overhead dovuto alla chiamata di funzione)

2.3 Categorie di IR

- grafiche (o strutturali)
  - orientate ai grafi
  - molto usate nella source-to-source translation, tipicam. per ott. che non hanno bisogno della struttura sofisticata di un middle-end
    es. openMP: di fatto annotazioni sul codice, come strumento semplice per la parall. (es. **outlining**: prendo es un loop e lo impacchetto in una funzione che poi dovra essere eseguita dai thread per la parallelizzazione) - non sto ottimizzando nel senso proprio del termine, ma sto trasformando il codice e lo sto rendendo eseguibile in maniera parallela
  - solitamente voluminose (basate su grafi) - tradeoff con il fatto che non coinvolgono il middle-end
  - es. ast, dag
- lineari
  - pseudocodice per macchine astratte
  - livello di astrazione vario
  - strutture dati semplici e compatte
  - facile da riarrangiare (evidentemente il più comodo per eseguire le ottimizzazioni)
  - es. 3ac
- ibride (sfruttano combinazioni delle prime due) (es cfg)

2.4 Esempi di rappresentazione

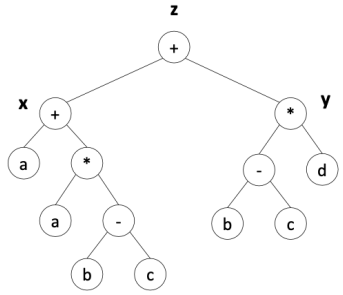
2.4.1 Sintassi concreta (testo)

Più semplice in quanto più vicina al livello di astrazione "umano" di ragionamento sul programma, ma non il livello corretto per ottimizzare ne comprendere correttamente la semantica del programma

```
let value = 8;
let result = 1;
for (let i = value; i>0; i = i - 1) {
  result = result * i;
}
console.log(result);
```

2.4.2 AST (Abstract Syntax Tree)

Albero i cui nodi rappresentano diverse parti del programma: il nodo radice rappresenta il **programma**, il quale a sua volta contiene un blocco di istruzioni dal quale discendono tanti figli quante le sue istruzioni




```
x = a + a * (b - c)
y = (b - c ) * d
z = x + y
```

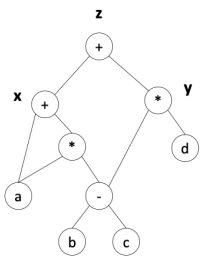
PRO: molto comodo per interpreti (basta usare una fz. ricorsiva per processare l'albero)
CONTRO: un nodo è un oggetto troppo generico → analizzare un ast per l'ottimizzazione impone ogni volta di ragionare sulla differenza semantica tra i nodi (complica molto)

2.4.3 DAG (Directed Acyclic Graph)

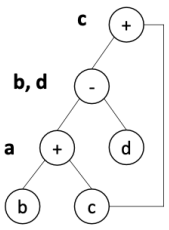
Contrazione di ast che evita la duplicazione di espressioni → **rappresentazione più compatta**
Limite: il riuso e possibile solamente dimostrando che il suo **valore non cambia** nel programma

essendo assegnamenti e chiamate frequentissimi, il fatto che il dag non abbia nozione di come le espr. cambino valore nel tempo non lo rende un buon candidato per le ottimizzazioni

Esempi




```
x = a+a*(b-c);
y = (b-c)*d;
z = x+y;
# espr. trovate
t1 = b-c;
t2 = a*t1;
x = a+t2;
y = t1*d;
z = x+y;
```




```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
# espr. trovate (ERRORE)
a = b + c; # cambia val.
d = a - d;
c = d + c;
```

2.4.4 3AC (3-Address Code)

Evidentemente adatto: tutte le istr. del programma vengono spezzettate in istr. di forma semplice simile all'assembly, di tipo `x = y op z` (1 operatore, massimo 3 operandi)

assignments	`x = y op z`
	`x = op y`
	`x = y[i]`
	`x = y`
branches	`goto L`
conditional branches	`if x relop y goto L`

procedure calls	`param x`
	`param y`
	`call p`

address and pointer assignments	`x = &y` `*y = z`

```
x - 2 * y   →   t1 = 2 * y
                t2 = x - t1
```

PRO:

- espressioni complesse spezzettate
- forma compatta e simil-assembly
- registri temporanei **intermedi, virtuali e illimitati** (tralascio problemi architetturali - n. di r. fisici a disposizione e eventuali op. di spill, cioè aggiungere load o store in mancanza di r. fisici)

Varianti di 3AC

A seconda dei vincoli che ho per l'implementazione pratica:

- **quadruple**: id istruzione, opcode, i 3 registri → semplice struttura record, facile da analizzare e riordinare ma i nomi espliciti prendono più spazio
- **triple**: id istruzione, opcode, 2 operandi → uso l'indice dell'espressione nell'array come "nome" del registro destinazione → risparmio spazio, ma diventa piu complesso da analizzare (nomi impliciti) e riordinare

Quadruple

`x - 2 * y`

(1)	load	t1	y	
(2)	loadi	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

Triple

`x - 2 * y`

(1)	load	y	
(2)	loadi	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

Inapplicabilità diretta della Constant Propagation con forma 3AC

La CP è applicabile solo se la variabile **se non cambia nel frattempo** → una IR di tipo 3AC non può applicarla immediatamente (devo prima analizzare il resto del codice)

2.4.5 SSA (Static Single Assignment)

- Evoluzione di 3AC che impone che la **definizione (assegnamento)** delle variabili avvenga **solo una volta** (def. multiple sono tradotte in multiple versioni della var)
- **PRO**: ogni definizione ha associata direttamente una **lista di tutti i suoi usi** - semplifica enormemente le ottimizzazioni di tipo CP e non solo

Quasi sempre uno dei passi di ottimizzazione prevede il passaggio a forma SSA

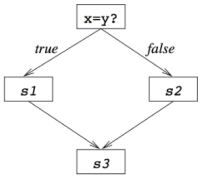
La scelta della IR dipende ovviamente dal livello di dettaglio necessario per ogni specifico compito → **in un compilatore coesistono più IR** (anche per questo esistono forme ibride)

2.4.6 CFG (Control Flow Graph)

- modella il trasferimento (flusso) del controllo in un programma tra **blocchi** di istruzioni
- permette di aggiungere informazioni sui **salti** al di sopra di una IR lineare
- i suoi nodi sono Basic Block
- gli archi rappresentano il flusso di controllo del programma (loop, condizioni, ecc.)

```
if (x == y)
   S1;
else
   S2;

S3
```



- un BB è una seq. di istruzioni in forma 3AC
  - singolo *entry point*: solo la prima istruzione puo essere raggiunta dall'esterno
  - singolo *exit point*: se eseguo la prima istr. **devo eseguire tutte le altre** - garantisco che venga **eseguito interamente**
  - Le chiamiamo sezioni single-entry, single-exit (possono essere sezioni anche piu grandi, ma le piu piccole di questo tipo sono i BB)
- un arco connette due nodi $B_i \to B_j \iff b_j$ può eseguire dopo B_i in qualche percorso del ctrl flow del programma
  - prima istr. di B_j è target dell'istr. di salto al termine di B_i
  - $\vee$ B_i non ha un istr. di salto come ultima istr. (nodo *falltrough*) e B_j è suo unico successore
- un CFG **normalizzato** ha i BB **massimali**
  - non possono essere resi più grandi senza violare condizioni
  - unisco i BB fallthrough che non hanno label all'inizio
  - posso avere CFG non norm. dopo qualche generico passo di ottimizzazione (non le facciamo accadere "spontaneamente")

Algoritmo per la costruzione del CFG

1. identificare il **leader** di ogni BB:
   - la prima istruzione
   - il target di un salto
   - ogni istruzione dopo un salto
2. il BB **comincia** con il leader e **termina** con l'istruzione immediatamente precedente un nuovo leader (o l'ultima istruzione)
3. **connettere** i BB tramite archi di 3 tipi:
   - **fallthrough** (o fallthru): esiste solo un percorso che collega i due blocchi
   - **true**: il secondo blocco è raggiungibile dal primo se un condizionale è `true`
   - **false**: il secondo blocco è raggiungibile dal primo se un condizionale è `false`

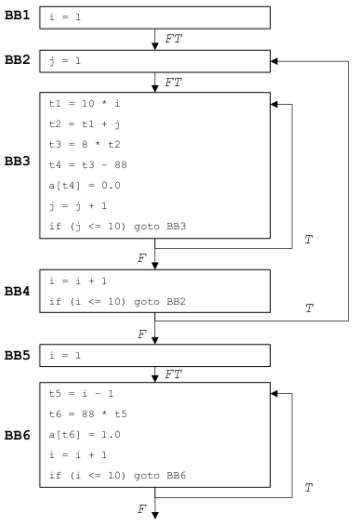


Fig. 2.1: Esempio di CFG

2.4.7 Dependency Graph

I nodi di un DG sono istruzioni; un arco connette due nodi di cui **uno usa il valore definito dall'altro**. Sono indispensabili per l'*instruction scheduling* e per mantenere il CPI della pipeline (ricordiamo quella RISC-V):

↷ Instruction Fetch

↓ Instruction Decode

↓ Execute

↓ Memory Access

↩ Write Back

ricordiamo pipeline riscv e data hazard: il fatto che un registro voglia leggere da un registro usato in un'op precedente (e non ancora salvato? recupera)

if id exe mem wb le fasi di pipelining

esempio con una mul: se provo a usare il registro usato nell'istr. prima, il risultato ancora non è stato scritto - nella fase di decode ancora non contiene il risultato aggiornato, pronto appena tra 2 cicli

Data hazard, generalmente gestito dalla **forwarding unit** che bypassa le fasi successive e inoltra direttamente il risultato appena ottenuto

Per quanto i controlli vengano svolti dalla fw. unit, in generale l'unico modo per evitare questo tipo di hazard è distanziare le istruzioni tra loro affinché il dato sia disponibile → inserisco nop (cicli di stallo), ma vado a "rompere" l'IPC pari a a 1 della pipeline sempre piena (< performance)

Soluzione migliore: **scheduling** del programma, spostando istruzioni che non dipendono da quei registri al posto di aggiungere `nop` questo è uno dei compiti principali di un backend - **in che ordine genero le istruzioni per massimizzare l'efficienza del programma**

come faccio a fare questa cosa? manualmente: vado a guardare le istruzioni e cerco a mano le dipendenze; il backend sfrutta la IR di tipo dg che fornisce esattamente le informazioni sulle dipendenze tra istruzioni

qui si capisce cosa si intende per instruction level parallelism: dunque, quando è possibile sfruttare tutte le parti di architettura per "parallelizzare il codice" anche in caso di single thread (ottimizzazione senza reale parallelismo)

2.4.8 data dependency graph ddg

Specifico per multicore e parallelismo, usato per dare una rappresentazione tra le dipendenze dei **dati** - tipicamente i loop

per esempio, (loop innestati lavorano tipicamente su str dati complesse e multidimensionali (matrici, immagini, ecc.))

```
for (i = 0, i < n, i++)
  A[i]=init();
```

Ogni it. resta indipendente dalle altre (uso il solo indice del loop) - basta aggiungere ad es. `A[i-1] = A[i]` per generare dipendenze tra le iterazioni → il loop non è più parallelizzabile, e deve per forza essere eseguito in sequenza

Data la complessità delle casistiche reali, esistono vari modi per rappresentare lo spazio delle iterazioni di un loop e le dipendenze che ne derivano:
all'oggi usiamo il **polyhedral model** → rappresento lo spazio delle it. come un poliedro (a seconda del numero di loop innestati), che permette di capire se esiste qualche permutazione dei loop (direzione di attraversamento dello spazio delle iterazioni; ovvero ad esempio scambiare l'ordine dei loop) non soggetta a dipendenze

2.4.9 call graph

rappresetnazione grafica a grafo usata per ragionare sulle relazione tra le funzioni della translation unit del file (insieme delle potenziali chiamate tra funzioni)

rappr. gerarchica, utile soprattutto a livello di analisi **interprocedurale** (la maggior parte delle ottimizzazioni avvengono a livello *intraprocedurale*) (recupera questo concetto) (lavora sui file che abbiamo chiamato translation units, ovvero quelli da cui poi genero i file oggetto)

→ evidente come il compilatore abbia visibilita solo fino a livello dei singoli moduli: posso estendere le ottimizzazioni al massimo fino ai legami tra funzioni dello stesso modulo - le ottimizzazioni piu ampie si spostano a framework di ottimizzazione che agiscono a livello di linker per esempio

3 Ottimizzazione locale e basic value numbering - 24 marzo

3.1 scope dell'ottimizzazione

finora abbiamo lavorato in locale, ovvero all'interno di un singolo BB → un'ott. locale non si preoccupa del flusso di controllo di un programma, mentre una globale lavora a livello dell'intero CFG → lo scope viene influenzato da come viene gestito il flusso di controllo in un programma

- ott locale: entro un sibgolo bb, non si preoccupa del flusso
- ott globale: lavora a livello dell'intero CFG
- ott interprocedurale: lavora a livello del call graph, e quindi sui CFG di più funzioni

dead code elimination

es di ott locale

```
main {
  int a = 4;
  int b = 2;
  int c = 1;
  int d = a + b;
  print d;
}
```

evidentemente dobbiamo togliere la def di c, ma: solo quello? ricorda che rappresentano dead code le istr che definiscono (assegnano un valore a) una var che non è mai utilizzata → poiche la print non definisce nulla, stando a questa def è dead code! invece, estendiamo la definizione dicendo che lo sono le istr **prive di side effects** che definiscono ...

proviamo a costruire un algoritmo per implementare la dce così definita:

1. ∀ istruzione in BB
   - aggiungi operandi ad un metadato array `used`
2. ∀ istr in BB
   - se istr non ha destinazione e non ha side effects rimuovila
   - altrimenti se la destinazione non corrisponde a nessuno degli elem di `used` rimuovila

dall'esempio vediamo che funziona, ma iterativamente potremmo eliminare poi altre istr → soluzione semplice è aggiungere la parte iterativa all'algoritmo, che dunque esegue fino a convergenza

Esempio

altro esempio: ridefinizione di una variabile (slide 6-22) - ovvero il nostro algoritmo corrente non gestisce le dead stores

in generale, capiamo che scrivere un passo significa prima definire bene cosa deve fare...

per estendere l'algoritmo, dobbiamo essere in grado di rilevare gli assegnamenti multipli di una variabile, e quindi anche preoccuparci dell'ordine delle istruzioni! (più complicato)

analizzando pseudocodice a 6-29: lista di variabili definite; itero sulle istruzioni e rimuovo gli operandi da `last_def` (sono variabili usate da qualche parte) ; poi controllo le definizioni: se la destinazione dell'istruzione (LHS) si trova ancora in lastdef, la elimino da li e al suo posto metto l'istruzione stessa (??????????) (vedi da 6-29 a 6-38 per spiegazione iterativa fatta benino)

3.2 Local value numbering

tecnica utilizzata per considerare il concetto di ordine delle definizioni in assenza di proprietà di tipo SSA (che ti permette di non avere questo problema)

osserviamo 3 pattern che forniscono opportunità di eliminazione di codice ridondante:

- dead code elimination: 1 variabile e piu valori
- copy propagation: 1 valore e piu variabili
- common subexpression elimination: 1 valore (in forma di espressione) e piu variabili

```
main {
  int a = 100;
  int a = 42;
  print a;
}
```

```
main {
  int x = 4;
  int copy1 = x;
  int copy2 = copy1;
  int copy3 = copy2;
  print copy3;
}
```

```
main {
  int a = 4;
  int b = 2;
  int sum1 = a + b;
  int sum2 = a + b;
  int prod = sum1 * sum2;
  print prod;
}
```

sono tutti modelli di computazione che si focalizzano sulle **variabili** → focalizzandoci sui **valori** possiamo eliminare tutte le forme di ridondanza

soluzione ai tempi in cui ancora non era in uso il modello ssa: local value numbering

tecnica che ci aiuta nelle ott. proprio per il cambio focus da variabili a valori

costruisco un metadato in forma di tabella che riscrive le espr (istruzioni) in funzione dei valori gia osservati → evitando di riassegnare lo stesso valore a piu var si evita la ridondanza

esempio da 6-46 a 6-56

analizzo le istruzioni in termini del value, e in caso questo coincida con entry della tabella gia presenti punto direttamente a quella

semplice variante del programma:

```
#...
int sum1 = a + b;
int sum2 = b + a;
```

evidentemente nascono problemi, poiche il nostro algoritmo non conosce le proprieta aritmetiche delle operazioni → non vado a rimuovere l'istruzione

→ soluzione semplice: **canonicalizzare** l'algoritmo, ovvero imporre un ordine numerico tra le entry (i valori) e usarle sempre in ordine crescente per le op. commutative (di fatto è la tecnica usata da tutti i compilatori, essendo al piu semplice?)

4 data flow analysis

step successivo dell'analisi, da immaginare come un framework di analisi alla base per costruire la forma ssa (?) ; o come una metodologia di analisi

4.1 cos'è la dfa

- analisi locale: si focalizza sull'effetto di ogni istruzione → posso comporre gli effetti di tutte el istr per derivare informazione dall'inizio del bb ad ogni istruzione
- analisi globale - dataflowanalysis: simile, ma molto più complessa → analizza l'effetto di ogni BB, e poi ha una metodologia per comporre l'effetto dei BB ai CONFINI degli stessi per derivare informazione

skip 7-6
esempio 7-7 struttura ad amaca (hammock) - statement di tipo if che dirama su due possibili flussi di controllo → per ogni variabile x consente di derivare

- valore di x?
- quale "definizione" definisce x?
- la definizione è ancora valida (*live*)?

osserviamo che queste risposte noi le abbiamo gia ottenute in maniera molto semplice, grazie alla forma SSA delle istruzioni! e alla struttura di llvm
nota: stiamo "costruendo" la forma SSA, considerando un caso in cui non abbiamo ancora quel liverllo?

4.2 rappresentazione del programma statica o dinamica

statica: programma finito, un pezzo di codice → molto facile da analizzare dinamica: può avere infiniti percorsi di esecuzione, rappresenta una possibile esecuzione reale! → es. loop che si basa sull'analisi di un input
sono condizioni che un compilatore non può analizzare, soprattutto non in maniera statica e "finita" in termini di possibili istanze

capacita di analisi dei compilatori: limitata alle condizioni analizzabili e determinabili **staticamente**
motivo per cui spesso si passano al compilatore informazioni di "profiling" , misurate durante ripetute esecuzioni del programma prima di darlo in pasto al compilatore e i relativi passi di ottimizzazione

con la dfa siamo in grado di dire **per ogni punto del programma** qualcosa; combinando informazioni relative a tutte le possibili istanze dello stesso p.to
esempio di problema dfa: quale def definisce il valore usato nello statement `b=a`? vedi da slide 7-8 il codice

4.2.1 effetti di un bb

effetti di un'istruzione:

- uses : delle variabili
- kills : una precedente definizione
- defines : una variabile

combinando gli effetti delle singole istr si definiscono gli effetti di un BB:

- uso localmente esposto (locally exposed use): in un bb è un uso di una var che non è preceduto nei bb da una definizione della stessa variabile
- ogni definizione di una variabile nel BB killa tutte le definizioni delal stessa variabile che **raggiungono** il BB

- definizione localmente disponibile (locally available definition): ultima definizione di una variabile nel bb

esempio: 7-15

4.2.2 reaching definition

ogni istruzione di assegnamento è una definizione una definizione d raggiunge (reaches) un punto it$p$ se esiste un percorso da d a p tale per chu d non è uccisa (sovrascritta) lungo quel percorso

definizione del problema: determinare per ogni punto del programma se ogni definizione nel programma raggiunge quel punto - come lo facciamo? → usiamo un **bit vector** per ciascun punto del programma (ogni istruzione), con lunghezza del vettore pari al numero di definizioni ⇒ diventa una sorta di matrice con righe le istruzioni, colonne le definizioni

4.2.3 schema dfa

consideriamo un flow graph che ha sempre un BB entry e uno exit (single-entry e single-exit → sempre possibile)

come faccio a stabilire l'effetto del codice in ciascun BB? uso quelle che si chiamano funzioni di trasferimento: fz che correlano input e output tra loro per un dato bb

qual è l'effetto del flusso di controllo? lo stabilisco in base alla vicinanza dei blocchi: correlo outp e inp di blocchi adiacenti

alla fine dobbiamo solo risolvere queste equazioni

4.2.4 effetti di uno statement

partiamo dalle fz di trasf di statement (astrazione per indicare un istr? di assegnamento?)

l'output di uno statement ... rec blabla

ricorda che stiamo lavorando su bit vector!!!!!!!!!!!!!! una fz di trasferimento riempie out[s] a partire da in[s] e applicando qualche tipo di calcolo

alla fine posso combinare queste funzioni di trasferimento in maniera lineare! (non lo dim ma si potrebbe)

quindi la funz di tr di un BB compone linearmente le fz dei suoi statements

rec da carta e slide

4.2.5 effetti degli archi aciclici

caso predecessori multipli: unisco l'informazione con che criterio? join

oggi fino slide 7-42

ricordiamo che in generale stiamo usando un "approccio" bit-vector, che non possiede nozione di ordine di esecuzione - quando calcolo i kill, li calcolo sempre relativamente a **tutti gli altri blocchi!** es. slide 7-36: $Kill[B_1] = \{0, 2, 3, 4, 6\}$

riprendiamo la questione sugli effetti degli archi aciclici, nel caso di un nodo con predecessori multipli:

- $out[b] = f_b(in[b])$
- nodo di unione (join): nodo con predecessori multipli
- operatore di unione (meet): $in[b] = out[p_1] \cap out[p_2] \cap \ldots \cap out[p_n]$ con $p_1, \ldots, p_n$ tutti predecessori di b
- caso entry block: $out[\text{entry}] = \emptyset \implies in[B_1] = \emptyset$ **boundary condition**
- caso exit dall'esempio: $in[\text{entry}] = out[B_2] \cap out[B_3] = \{2, 3, 4, 5, 6\}$

notiamo come i due modi (bit-vector oppure usare join e meet) sono equivalenti (?)

4.2.6 effetti degli archi ciclici

notiamo un problema con gli archi **ciclici** - posso arrivare a condizioni in cui non ho ancora calcolato l'out di un qualche bb, ma mi serve per un arco ciclico appunto che punta a un blocco precedente

soluzione: devo iterare fino a convergenza, ma soprattutto definire delle condizioni iniziali (come per l'out dell'entry block) che ci dicono che in uscita ad ogni blocco (all'inizio) trovo l'insieme vuoto (necessaria per avere qualcosa su cui operare alla prima iterazione che considera dei backedge)

4.3 liveness analysis

4.3.1 live variable analysis

definizioni: variabile live o dead: una var è viva in un punto p del programma se il valore di v è usato da quel punto in avanti da qualche parte nel programma - altrimenti è morta

altri possibili motivi di questa analisi (oltre alla dce): register allocation: nel momento in cui devo fare i conti con un'architettura specifica, devo decidere come allocare i registri (non sono illimitati) - register allocation è l'op. che cerca di evitare le spill in cache e in memoria, e che cerca i casi in cui risulta possibile riutilizzare un certo registro: dipende evidentemente dalla liveness di una variabile!

definizione del problema in termini di dfa: $\forall$ BB devo stabilire se ogni variabile è viva in ciascuno di essi $\rightarrow$ bit vector di lunghezza pari al numero di variabili (evidentemente...)

4.3.2 funzione di trasferimento

recupera questione su forward analysis

che informazione sto cercando per capire se una definizione raggiunge o meno un punto del programma? devo analizzare "il passato" - gli statement tra la definizione e il punto di interesse per cercare eventuali kill (analizzo dal punto all'entry point, a ritroso)

nel contesto della liveness analysis invece, devo analizzare il "futuro" - dal punto p all'exit point cerco gli usi della variabile

(abbiamo spiegato la differenza tra forward e backward analysis, nelle sezioni prima abbiamo implicitamente considerato backward analysis per il problema di reaching definition)

per la formulazione più tipica della liveness analysis (ce ne sono diverse, a seconda delle fonti) usiamo

- l'insieme delle variabili vive che può generare un BB (use)
- l'insieme delle variabili definite nel BB (def)
- le variabili vive in ingresso che il bb può propagare (out - def)
- fz di trasferimento quindi è: in = use U out - def

> nota che di fatto stiamo riscrivendo quello che abbiamo detto per la reaching definition! solo al contrario - in questo momento stiamo risalendo al contrario per calcolare gli insiemi che ci servono e le funzioni di trasferimento, e sarebbe sufficiente "sostituire" come nomenclatura use e def con gen e kill (non mi ricordo se questo è l'ordine giusto)

anche qui abbiamo quindi un operatore di meet, applicabile ai join node (nodi con **successori** multipli)

esempio 7-56 e 7-57, recupera appunti da dani

algoritmo iterativo a 7-58

boundary condition: input dell'exit block è l'insieme vuoto, così come le starting conditions per tutti i bb tranne l'entry block

da cosa deriva questa scelta delle condizioni iniziali? ovviamente dall'operatore di meet - se stessimo usando per qualsiasi motivo un meet op di tipo intersezione, usare l'insieme vuoto "pialla tutto" e impedisce la propagazione dell'informazione

7-73 : a convergenza si arriva indipendentemente dall'ordine in cui calcolo le funzioni dei bb (cambia al massimo il numero di iterazioni per arrivare a convergenza)

7-74 : tabella di confronto tra i due problemi visti, volendoli formalizzare tramite il framework della dfa

4.4 available expressions

altro problema modellabile tramite dfa

esempio:

```
if (...) {
  x = m + n;
} else {
  y = m + n;
}
z = m + n;
```

dunque risulta utile ad es per ottimizzazioni come la common subexpr elimination

ma cosa succede se l'espressione non viene calcolata nel ramo else invece? → la possibile soluzione a livello di sorgente sarebbe calcolare l'espressione prima dell'if, e usarlo poi nelle definizioni successive che la richiedono

concetto di available expressions

4.4.1 available expressions

Il concetto di available expr. è necessario come maniera rigorosa di ragionare sulla ridondanza

Consideriamo come dominio l'insieme di tutte le espressioni del programma (solo espr binarie del tipo x + cerchiato y) - (stiamo considerando match ... ? recupera)

OCIO CHE FORWARD E BACKWARD SI RIFERISCONO (PRIMA ERANO INVERTITI MI PARE) ALLA DIREZIONE CHE VADO A CONSIDERARE RISPETTO ALLA DIREZIONE CLASSICA DEL FLUSSO ENTRY → EXIT - fw va da entry a punto p, backward da exit a p

recupera tutto bene fino a 7-79

7-80: esempio

7-81 fino a 7-87 spiega come mai stiamo facendo fw analysis → per capire quali expr. eliminare dobbiamo guardare al passato - elimino quelle che non sono state calcolate in tutti i blocchi predecessori (available appunto in questo caso appena descritto) → è proprio il motivo per cui sto usando come operatore di meet l'**intersezione**

7-88 fino 7-92 : condizioni al contorno e condizioni iniziali - non posso usare l'insieme vuoto con l'intersezione, devo usare quello che in insiemistica chiamo insieme universo (o universale), dunque in termini di bit vector un vettore di tutti 1

7-93 : tabella riassuntiva , ricordiamo che il dominio sono tutte le operazioni con due operandi del programma e quindi lunghezza vettore pari alla sua cardinalità

Esempio

immagine 7-94

risolvere il problema di dfa delle available expr per il cfg in figura

dominio:

- `a-b`
- `a*b`
- `a-1`

condizione di partenza: out[entry] = out[bb1] = emptyset

recupera da foglio

essendo la dfa per natura statica, l'esempio mostra come l'essere statici è di per se pessimistico - considero sempre il caso peggiore (stiamo parlando dell'espressione condizionale?)

il risultato dell'analisi sarebbe diverso se conoscessi l'istanza specifica - sapendo ad esempio che bb4 sarà falso potrei fare delle assunzioni che, staticamente, non posso fare, e dunque devo conservativamente ritornare sempre al caso peggiore che il loop esegua almeno una volta ?

in ogni caso, sono informazioni aggiuntive relative al comportamento dinamico (a esecuzione) del programma che il compilatore non conosce a priori - posso al massimo fornirgliele io dall'esterno

5 1 aprile - 2o assignment su dfa

per ciascuno dei 3 problemi di analisi, definire dominio direzione fz di trasf meet op e condizioni di boundary e iniziali

inoltre per il cfg di esempio fornito bisogna compilare una tabella per descrivere le iterazioni che servono a raggiungere convergenza

per validare la formulazione, fate domande del tipo es. liveness aanalysis: questa variabile è viva in questo punto? eccetera

5.1 very busy expressions

very busy se indipendentemente dal percorso preso, l'espr. viene usata prima che uno dei suoi operandi venga definito (viene usata) ci interessa conoscere le espressioni disponibili .. ?

utile perche permette code hoisting (tecnica usata in loop invariant code motion, vedi inizio, recupera)

sposto prima del loop una certa espressione: posso farlo solo se sono sicuro che qualsiasi espr che usa il valore dell'espr da spostare ... rec

capire se posso evitare di ripetere un'op in maniera ridondante eseguendola una voltra asola in un punto dove il control flow e comune a tutti i path?

in ogni caso ci ritorneremo, per ora ci limitiamo a ragionare in termini di dfa relativamente al sottoproblema delle vbexpr

5.2 dominator analysis

analisi dei dominatori - inizieremo a parlarne quando inizieremo a parlare dei loop, fondamentale per riconoscere l'esistenza di un loop

riconoscere l'esistenza di un loop significa in primis riconoscere la presenza di un ciclo

un blocco domina un altro blocco se il primo appare sempre in ogni percorso prima del secondo - in questo caso x domina y

questa era la condizione di dominanza; unendola alla condizione di "direzione" dell'arco, posso rilevare la presenza di cicli (ne parleremo in dettaglio)

come impostare il framework di dfa per capire quali blocchi sono dominanti rispetto agli altri?

da esempio: i blocchi che dominano f sono a c f, non d ed e perche sono opzioni "mutualm escl" ovvero non necessariamente devo passare attraverso uno dei due; inoltre f perche un blococ e sempre dominatore di se stesso

5.3 constant propagation

per poter trovare i punti del programma in cui le var hanno valore costante bisogna usare un dominio speciale: devo associare sia nome variabile che valore della costante - se una var viene nel frattempo riassegnata l'informazione di constant pr cambia

dominio fatto da coppie del tipo var,val-costante `(x,c)`

dunque garantisco che per coppie x,c ad ogni uso di x sia associata c (il suo valore)

questa analisi riesce a determinare il valore costante di espr binarie dove 1 o entrambi gli operandi son ocsotanti note - nel determinare le equazioni possiamo tenere conto di questa info, e quindi eventualmente fare constant folding (non obbligatorio?)

6 sezione 9 2 aprile

6.1 cos e un loop

slide 9-4: obiettivo → definire un loop in termini di teoria dei grafi (cfg), ovvero indipendentemente dalla loro sintassi e dal tipo (for, while, goto, eccetera ... o anche costrutti stile assembly con salti condizionati e non)

accordiamoci sulla terminologia: cosè un loop e con quale terminologia lo definiamo

generalmente, non tutti i cicli sono un "loop" da un pto di vista dell'ottimizzazione → es. 9-5 a-c-d è un ciclo, ma anche un loop? e c-d ?

elementi chiave per riconoscere un loop:

- gli archi devono formare almeno un ciclo
- (fondamentale) deve avere un **singolo entry point** (tutti gli archi, se multipli, devono entrare nello stesso punto)

6.2 definizioni formali

dominator

un nodo d domina un nodo n in un grafo (d dom n) se ogni percorso dall'ENTRY node a n passa per d

come facciamo ad usare e rappresentare comodamente questa proprietà?

dominator tree

modo per rappresentare la proprietà di dominanza in forma di albero

- a $\rightarrow$ b nel dominator tree $\iff$ a domina immediatamente b
- non compaiono le relazioni di "auto-dominazione" (ogni nodo domina sempre se stesso, ma nell'albero non serve rappresentarlo)
- il nodo entry è la radice
- ogni nodo d domina solo i suoi discendenti nell'albero

immediate dominator

l'ultimo dominator di n su qualsiasi percorso da entry a n
m domina immediatamente (strettamente) n (m sdom n) iff m dom n AND m != n
esempio 9-9 di dominator tree
altro esempio 9-10 e 9-11
gli esempi ci mostrano come si formano le gerarchie di dominator
note queste definizioni, possiamo iniziare a provare a definire i loop

6.3 loop naturali

pur specificandoli in molti modi diversi nei src, dal punto di vista dell'analisi importa solo che la rappr abbia proprietà che facilitino l'ottimizzazione, ovvero

- singolo entry point: header $\rightarrow$ l'header **domina tutti i nodi nel loop**
- back edge, ossia arco la cui testa domina la propria coda (tail $\rightarrow$ head): un back edge **deve far parte di almeno un loop**

6.3.1 identificare i loop naturali

1. trovare le relazioni di dominanza
2. identificare i back edges
3. trovare il loop naturale associato al back edge

trovare i dominatori

lo impostiamo in termini di dfa (segue esempio di prima)

trovare i back edges

algoritmo usato ad es. dal dragon book (sezione 2.3.4): depth-first traversal

- inizio alla radice e visito ricorsivamente i figli di ogni nodo in qualsiasi ordine
- importante la "velocità di discesa": prima scendo ed esploro in profondità, a prescindere dall'ordine

il percorso della visita deinisce un depth-first spanning tree (DFST):

- archi solidi: struttura dell'albero
- archi tratteggiati: altri archi del cfg

come categorizzo gli archi:

- advancing (A) edges: da antenato a discendente, ovvero gli archi detti *proper* - gli archi solidi sono tutti A

- retreating (R) e.: da discendente a antenato (non necessariamente proper → da un nodo a se stesso) - solo archi tratteggiati
- cross (C) edges: archi tali per cui nessuno dei due nodi è antenato dell'altro

se disegniamo il dfst in modo che i figli siano aggiunti da sx a dx nell'ordine di visita, allora i cross edges vanno sempre da dx a sx

definizione: 9-20 algoritmo: 9-20, si basa su concetto di retreating edge, e controlla se la testa è nella lista dei dominatori della coda (usa h head e t tail)

dunque di fatto in questo modo vado a riconoscere i back edges come questi casi specifici di retreating edges su cui eseguo il controllo della dominanza

trovare il loop naturale

il loop naturale di un back edge è il più piccolo insieme di nodi che include head e tail del back edge e non ha predecessori fuori da questo insieme

algoritmo: 9-22

dunque genericamente rappresento i loop in maniera "header-centrica"

6.4 preheader

spesso succede che sia necessario (anche propedeuticam ad altre ottimizzazioni) dare la garanzia che quando arrivo all'header block, ci arrivo con un arco di tipo falltrough nel grafo - es per la loop invariant code motion (caso di code hoisting), quindi quando devo spostare un'istruzione in un punto comune al control flow di tutte le iterazioni del loop → tipico inserire un blocco detto preheader appena prima del loop, fatto apposta per inserire le istruzioni da eseguire una volta sola in seguito alla code hoisting (operazioni preliminari, eccetera)

IMG 9-23

questo vuol dire che per manipolare le str dati di tipo loop in llvm ci sono a disposizione primitive per recuperare proprio questi blocchi fondamentali (poi per navigare nel resto dei blocchi possiamo usare i classici iteratori, sia seguendo l'ordine del cfg sia non)

7 use-def e def-use chains

per come funziona l'ottimizzazione, abbiamo capito che serve la capacità di ricollegare in maniera efficiente la definizione di una variabile a tutti i suoi usi (per esempio per propagare un risultato) → per questo vengono previsti questi riferimenti nell'IR llvm

7.1 dove viene definita o usata una variabile

la definizione di def use e use def chain è precedente a quella di ssa → lo scope lessicale del programma in cui si potevano trovare gli usi della variabile era molto più ampio

es. loop invariant code motion e copy propagation: 9-25

es licm se le variabili usate nell'espr che definisce a sono definite all'interno del loop evidentemente non posso spostare la definizione fuori dal loop

es cp riprendo il concetto di reaching definition ...

notando l'utilità di poter scorrere in maniera agevole le relazioni def usi, vogliamo un'IR che la preveda → permette una forma di analisi "sparsa", ovvero che ignora tutte le istruzioni che non sono casi di uso o definizione specifici all'istruzione correntemente analizzata

9-26: il discorso sulle catene può essere ulteriormente semplificato se si riesce a ridefinire completamente anche il nome della variabile ad ogni sua ridefinizione (ci avviciniamo a ssa?) (permette catene evidentemente sensibilmente più corte)

9-28: mostra come le catene possono essere onerose