16. Analisi del Flusso dei Dati

L'analisi del flusso dei dati (Data-flow Analysis, DFA) è un insieme di tecniche che consentono di ragionare, durante la fase di compilazione, sul flusso dei valori in fase di esecuzione. Questo approccio non rappresenta un fine a sé stante, ma i suoi risultati vengono utilizzati per garantire la sicurezza del codice e identificare opportunità di ottimizzazione.

L'analisi richiede quasi sempre la costruzione di un grafo, che può essere un grafo del flusso di controllo, un grafo delle chiamate o loro derivati. Per migliorare l'efficienza, si possono utilizzare grafi di valutazione sparsi per modellare il flusso dei valori.

Generalmente, l'analisi viene formulata come un sistema di equazioni simultanee, dove insiemi vengono associati ai nodi e agli archi del grafo. Spesso, questi insiemi utilizzano strutture come reticoli o semireticoli.

Il risultato desiderato è solitamente una soluzione *meet over all paths*, ovvero:

- "Cosa è vero su ogni possibile percorso a partire dall'ingresso?"
- "È possibile che un determinato evento si verifichi lungo uno qualsiasi dei percorsi dall'ingresso?"

Due Problemi di Analisi del Flusso dei Dati: *Dom* e *Live*

Il problema del calcolo dei dominatori (*Dom*) si basa sull'analisi dei nodi all'interno del grafo di flusso. Questo si traduce in un insieme di semplici equazioni di analisi del flusso dei dati, che possono essere risolte utilizzando qualsiasi risolutore di equazioni per analisi del flusso.

Si definiscono le condizioni iniziali per i dominatori:

- $DOM(n_0) = \{n_0\}$
- $DOM(n) = N$, $\forall n \neq n_0$, dove N è l'insieme dei nodi del grafo di flusso.

L'equazione che definisce i dominatori è

$$DOM(n) = \{n\} \cup \bigcap_{p \in preds(n)} DOM(p)$$

dove $preds(n)$ rappresenta l'insieme dei predecessori del nodo n.

Il dominio per il calcolo delle variabili vive (*Live Variables*) è costituito dall'insieme dei nomi delle variabili nella procedura in analisi. Le equazioni del flusso di dati per le variabili vive sono più complesse e si basano sui seguenti insiemi:

- $UEVAR(b)$: l'insieme dei nomi di variabili utilizzate nel blocco b prima di essere definite nello stesso blocco.
- $VARKILL(b)$: l'insieme dei nomi di variabili definite nel blocco b.

Le condizioni iniziali sono

$$LIVEOUT(n) = \emptyset, \forall n$$

Le equazioni del punto fisso per calcolare le variabili vive sono:

1. Per il flusso in uscita (*live-out*): $LIVEOUT(b) = \bigcup_{s \in succ(b)} LIVEIN(s)$, dove $succ(b)$ rappresenta l'insieme dei successori del blocco b.
2. Per il flusso in ingresso (*live-in*): LIVEIN(b) = $UEVAR(b) \cup (LIVEOUT(b) \setminus VARKILL(b))$

Algoritmo Classico: Algoritmo Iterativo Round-Robin

L'algoritmo iterativo round-robin è un metodo semplice ed efficace per calcolare soluzioni al punto fisso per problemi di flusso di dati. Si basa su aggiornamenti successivi degli insiemi associati ai nodi del grafo fino a raggiungere la stabilità, ovvero quando gli insiemi non cambiano più.

Calcolo dei Dominatori (DOM)

L'obiettivo è calcolare gli insiemi dei dominatori $DOM(n)$ per ciascun nodo n. L'algoritmo procede come segue:

1. Inizializzare gli insiemi:
   - $DOM(n_0) \leftarrow \{n_0\}$ per il nodo iniziale.
   - $DOM(n) \leftarrow N$ (l'insieme di tutti i nodi) per ogni altro nodo n.
2. Eseguire iterazioni fino a stabilità:
   - Impostare una variabile $change$ a vero.
   - Mentre $change$ è vero:
     - Impostare $change$ a falso.
     - Per ciascun nodo n_i, calcolare un insieme temporaneo $TEMP$: $TEMP \leftarrow \{n_i\} \cup \bigcap_{p \in preds(n_i)} DOM(p)$
     - Se $DOM(n_i)$ è diverso da $TEMP$, impostare $change$ a vero e aggiornare $DOM(n_i) \leftarrow TEMP$.

L'algoritmo termina quando gli insiemi $DOM(n)$ smettono di cambiare, raggiungendo così il punto fisso.

Calcolo delle Informazioni di Vivacità (LIVEOUT e LIVEIN)

L'algoritmo può essere utilizzato anche per calcolare le informazioni di vivacità. L'obiettivo è determinare $LIVEOUT(n)$ e $LIVEIN(n)$ per ciascun nodo n. La procedura è simile al calcolo dei dominatori, ma segue il flusso all'indietro nel grafo.

1. Inizializzare gli insiemi:
   - $LIVEOUT(n_f) \leftarrow \emptyset$ per il nodo finale.
   - $LIVEIN(n) \leftarrow UEVAR(n)$ per ogni nodo n, dove $UEVAR(n)$ è l'insieme delle variabili utilizzate prima di essere definite nel nodo n.
2. Eseguire iterazioni fino a stabilità:
   - Impostare una variabile $change$ a vero.
   - Mentre $change$ è vero:
     - Impostare $change$ a falso.
     - Per ciascun nodo n, calcolare:
       $TEMP \leftarrow \bigcup_{s \in succ(n)} LIVEIN(s)$
       Se $LIVEOUT(n)$ è diverso da $TEMP$, impostare $change$ a vero e aggiornare $LIVEOUT(n) \leftarrow TEMP$.
       Aggiornare $LIVEIN(n)$ come:
       $LIVEIN(n) \leftarrow UEVAR(n) \cup (LIVEOUT(n) \cap \neg VARKILL(n))$

La funzione di trasferimento per $LIVEOUT$ può essere incorporata direttamente nel calcolo per eliminare la necessità di rappresentare esplicitamente $LIVEIN$, ma questa formulazione esplicita è preferita per maggiore chiarezza.

Operazione di Convergenza

Entrambi i calcoli sfruttano una "meet operation" (intersezione per i dominatori, unione per la vivacità) che garantisce la monotonia e la convergenza verso il punto fisso. La semplicità dell'algoritmo lo rende una scelta comune per problemi di analisi del flusso di dati.

Algoritmo Iterativo con Worklist

L'algoritmo iterativo basato su una *worklist* rappresenta un'alternativa ottimizzata all'algoritmo round-robin per problemi di flusso di dati. L'idea principale è ricalcolare gli insiemi $LIVEOUT$ e $LIVEIN$ solo per i nodi che richiedono un aggiornamento, riducendo così il numero di operazioni necessarie.

Procedura dell'Algoritmo

1. **Inizializzazione**:
   - La *worklist* viene inizializzata con tutti i nodi del grafo di controllo.
   - Per il nodo finale n_f, si imposta $LIVEOUT(n_f) \leftarrow \emptyset$.
   - Per tutti gli altri nodi n, si imposta $LIVEIN(n) \leftarrow UEVAR(n)$, dove $UEVAR(n)$ è l'insieme delle variabili utilizzate prima di essere definite nel nodo n.
2. **Ciclo Iterativo**:
   - Finché la *worklist* non è vuota:
     - Rimuovere un nodo n dalla *worklist*.
     - Calcolare un insieme temporaneo $TEMP$ come:
       $TEMP \leftarrow \bigcup_{s \in succ(n)} LIVEIN(s)$
     - Se $LIVEOUT(n) \neq TEMP$, aggiornare $LIVEOUT(n) \leftarrow TEMP$.
       Inoltre, aggiornare $LIVEIN(n)$ come:
       $LIVEIN(n) \leftarrow UEVAR(n) \cup (LIVEOUT(n) \cap \neg VARKILL(n))$
     - Aggiungere i predecessori di n alla *worklist* se $LIVEOUT(n)$ è cambiato.

Vantaggi dell'Algoritmo con Worklist

Questo algoritmo evita di ricalcolare $LIVEOUT(n)$ e $LIVEIN(n)$ per i nodi in cui gli insiemi non sono cambiati, riducendo il numero di operazioni sugli insiemi rispetto all'approccio round-robin. Tuttavia, ciò introduce un costo aggiuntivo dovuto alla manipolazione della *worklist*. Per minimizzare questo costo, la *worklist* deve essere implementata come un insieme per evitare duplicati.

Confronto con l'Algoritmo Round-Robin

A differenza dell'algoritmo round-robin, che effettua passaggi completi attraverso tutti i nodi indipendentemente dai cambiamenti, l'algoritmo con *worklist* è più mirato. Ricalcola i valori solo per i nodi che hanno subito modifiche nei loro insiemi, garantendo una maggiore efficienza computazionale per problemi complessi.

Implementazione della Worklist

L'implementazione della *worklist* come un insieme garantisce che ogni nodo venga aggiunto una sola volta, evitando ricalcoli inutili. Questa scelta è cruciale per ottenere il massimo

beneficio dall'algoritmo, riducendo il numero complessivo di operazioni e migliorando le prestazioni.

Confronto tra Algoritmi Round-Robin e Worklist

Entrambi gli algoritmi, round-robin e basato su worklist, condividono la stessa logica fondamentale, ma si differenziano nel modo in cui elaborano i nodi e ottimizzano le operazioni. Di seguito viene presentata una panoramica delle differenze e delle caratteristiche principali.

Termination, Correttezza e Complessità

Le dimostrazioni di terminazione, correttezza e complessità derivano originariamente dalla versione round-robin dell'algoritmo. Poiché il worklist algorithm apporta modifiche negli stessi punti e nello stesso ordine dell'algoritmo round-robin, tali argomentazioni possono essere estese al worklist algorithm, purché questo utilizzi un ordine di elaborazione appropriato.

Efficienza del Worklist Algorithm

Il worklist algorithm è più efficiente rispetto al round-robin perché visita un numero inferiore di nodi, limitandosi ai nodi che necessitano di aggiornamenti. Questo approccio riduce significativamente il numero di operazioni sugli insiemi $LIVEOUT$ e $LIVEIN$, soprattutto in problemi con ampie regioni di stabilità.

Una versione del worklist algorithm con due worklist (*current* e *next*) può essere implementata per riprodurre esattamente il comportamento del round-robin:

1. I nodi vengono estratti dalla worklist *current* in Reverse Postorder (RPO).
2. I nodi che necessitano di ulteriori elaborazioni vengono aggiunti alla worklist *next*.
3. Quando *current* si svuota, le due worklist vengono scambiate, corrispondendo così a un passaggio del round-robin.

Questa variante effettua le stesse modifiche nell'ordine del round-robin, garantendo che gli argomenti di terminazione e correttezza rimangano validi.

Ordine di Elaborazione

Sebbene l'ordine RPO sia una scelta naturale per molti problemi, potrebbe non essere sempre il migliore. Iterare attorno a un ciclo fino alla stabilizzazione può risultare più rapido in alcuni casi. La scelta dell'ordine dipende spesso dal problema specifico e dall'implementazione:

- La struttura dati utilizzata per la worklist può influenzare l'efficienza.
- Diversi ordini di elaborazione possono avere impatti misurabili sulla velocità di convergenza.

Il Worklist Algorithm come Versione Sparsa

Il worklist algorithm può essere considerato una versione "sparsa" dell'algoritmo round-robin. Mentre il round-robin esamina tutti i nodi ad ogni passaggio, il worklist algorithm si limita ai nodi che necessitano di aggiornamenti, riducendo il lavoro complessivo. Tuttavia, questa efficienza richiede un costo aggiuntivo per gestire la struttura della worklist, che deve essere implementata con attenzione per evitare duplicati e garantire che l'algoritmo rimanga efficace.

Conclusione

Il worklist algorithm offre una maggiore efficienza in molti casi rispetto al round-robin, specialmente in grafi grandi o complessi. Tuttavia, l'ordine di elaborazione e la struttura dati utilizzata per la worklist possono influenzare significativamente le prestazioni. La scelta tra i due algoritmi dipende dal contesto e dalle priorità del problema da risolvere.

Proliferazione dei Problemi di Analisi del Flusso di Dati (GDFAPs)

Negli anni '60 e '70, molti problemi di analisi del flusso di dati (*Generalized Data-Flow Analysis Problems*, GDFAP) vennero proposti come strumenti per ottimizzare i compilatori. I GDFAP divennero rapidamente lo standard per dimostrare la sicurezza di una trasformazione. L'introduzione di una nuova trasformazione implicava la formulazione di un nuovo problema GDFAP, rendendo l'analisi del flusso di dati centrale nel tempo di compilazione.

Durante questo periodo, i compilatori passavano una grande parte del tempo di compilazione risolvendo problemi GDFAP. Ciò era complicato dalla lentezza dei computer (1-10 MIPS) e dalla loro limitata capacità di memoria (16-32 MB). Per affrontare questa complessità, vennero sviluppati framework per semplificare e standardizzare l'analisi del flusso di dati.

Esempio: Variabili Vive e Trasformazioni

L'analisi delle variabili vive (*Live Variables Analysis*) è un classico problema di flusso di dati che viene utilizzato, ad esempio, per eliminare gli *store* inutili. Un valore in un registro viene considerato "morto" se:

- È stata raggiunta la sua ultima definizione.
- Non verrà mai più utilizzato lungo nessun cammino del grafo di controllo.

Gli *store* verso variabili morte possono essere eliminati dal compilatore, tranne nei casi in cui servano per il debugging.

Formulazione del Problema

L'analisi delle variabili vive è un problema di flusso **all'indietro**. L'obiettivo è calcolare l'insieme $LIVEOUT(b)$ per ogni blocco b, che rappresenta le variabili vive in uscita dal blocco. La formulazione è la seguente:

- $LIVEOUT(b) = \bigcup_{s \in succ(b)} (UEVAR(s) \cup (LIVE(s) \cap \neg VARKILL(s)))$
- $LIVEOUT(n_f) = \emptyset$ per il nodo finale.

Dove:

- $LIVE(b)$ è l'insieme delle variabili vive in uscita dal blocco b.
- $VARKILL(b)$ è l'insieme delle variabili ridefinite nel blocco b.
- $UEVAR(b)$ è l'insieme delle variabili utilizzate nel blocco b prima di essere ridefinite.

Ruolo dell'Analisi delle Variabili Vive

L'analisi delle variabili vive è fondamentale in due ambiti principali:

1. Allocazione dei registri, dove le variabili vive determinano il momento in cui un registro può essere liberato.
2. Costruzione di una pruned-SSA (Static Single Assignment) ottimizzata, dove solo le variabili vive vengono mantenute nei nodi ϕ.

Formulazione Standard

La formulazione standard per il calcolo degli insiemi nell'analisi del flusso di dati utilizza un'operazione "meet" del tipo:

$f(x) = a \cup (b \cap c)$

Questa formulazione si applica a una singola equazione, rendendola efficiente da risolvere iterativamente fino al raggiungimento del punto fisso.

Conclusione

La proliferazione dei problemi GDFAP e il loro ruolo nei compilatori degli anni '60 e '70 ha portato a un'evoluzione significativa nell'ottimizzazione del flusso di dati. L'analisi delle variabili vive, uno dei problemi più comuni, rimane un pilastro nell'ottimizzazione dei programmi e nell'allocazione efficiente delle risorse.

Calcolo delle Espressioni Disponibili (Available Expressions)

Un'espressione è considerata *disponibile* all'ingresso di un blocco b se, lungo ogni cammino dal nodo iniziale del grafo di controllo al blocco b, l'espressione è stata calcolata e nessuna delle sue sottoespressioni è stata successivamente modificata.

Definizione Formale

L'insieme delle espressioni disponibili per un blocco b si calcola come segue:

$$AVAIL(b) = \bigcap_{x \in preds(b)} (DEEXPR(x) \cup (AVAIL(x) \cap \neg EXPRKILL(x)))$$

Dove:

- $DEEXPR(b)$ è l'insieme delle espressioni definite in b che non vengono successivamente "uccise".
- $EXPRKILL(b)$ è l'insieme delle espressioni uccise in b, ovvero quelle che non sono più valide perché uno dei loro componenti è stato modificato.
- $preds(b)$ rappresenta i predecessori del blocco b nel grafo di controllo.

Condizioni Iniziali

1. Per il nodo iniziale n_0, $AVAIL(n_0) = \emptyset$, poiché nulla è stato calcolato prima dell'inizio del programma.
2. Per tutti gli altri nodi n, $AVAIL(n)$ è inizializzato come l'insieme di tutte le espressioni, dato che l'operazione di intersezione ($\cap$) restringerà progressivamente il risultato.

Proprietà delle Espressioni Disponibili

L'analisi delle espressioni disponibili è un problema di flusso di dati **in avanti**, il che significa che l'informazione si propaga dai predecessori ai successori. La dimensione dell'insieme $AVAIL$ dipende dal numero di espressioni nel programma, $|AVAIL| = |expressions|$.

Utilizzo delle Espressioni Disponibili

Le espressioni disponibili costituiscono la base per la **eliminazione globale delle sottoespressioni comuni (Global Common Subexpression Elimination, GCSE)**. Questo tipo di ottimizzazione consente al compilatore di eliminare calcoli ridondanti, migliorando l'efficienza del programma.

Se un'espressione come $x + y$ appartiene a $AVAIL(b)$, il compilatore può sostituire ogni nuova occorrenza di $x + y$ in b con un riferimento al valore precedentemente calcolato. Questo processo segue alcune precauzioni standard per garantire la correttezza:

- Calcolare un *hash* di $x + y$ e assegnare un nome temporaneo all'espressione.
- Copiare tutte le occorrenze di $x + y$ al nome temporaneo.
- Camminare all'indietro nel codice per trovare il numero minimo di copie necessarie.

Il risultato è simile all'algoritmo di *value numbering*, che si basa sull'identità testuale delle espressioni piuttosto che sulla loro identità di valore. Questo metodo riconosce anche le ridondanze portate attraverso i branch che chiudono i cicli.

La Forma Classica della GCSE

La tecnica descritta rappresenta una delle forme più classiche e antiche di ottimizzazione basata sull'eliminazione delle sottoespressioni comuni. Essa migliora le prestazioni eliminando le ridondanze globali nei programmi, mantenendo al contempo l'integrità del comportamento originale.

Very Busy Expressions

Un'espressione e è considerata *very busy* in uscita da un blocco n se viene valutata e utilizzata lungo **ogni cammino** che parte da n e arriva al nodo finale n_f, e se la sua valutazione alla fine di n produrrebbe lo stesso risultato della sua prossima valutazione lungo quei cammini.

Scopo dell'Analisi delle Very Busy Expressions

L'analisi delle *very busy expressions* viene utilizzata per identificare opportunità di ottimizzazione del codice attraverso una trasformazione chiamata *hoisting*. Lo scopo principale di questa ottimizzazione è ridurre lo spazio occupato dal codice, eliminando valutazioni ridondanti e spostando le espressioni in posizioni migliori.

Piano

L'approccio per ottimizzare il codice si articola come segue:

- Annotare ogni blocco n con un insieme $VERYBUSY(n)$ che contiene le espressioni *very busy* alla fine del blocco.
- Eseguire un'analisi per identificare opportunità di ottimizzazione.
- Se un'espressione e appartiene a $VERYBUSY(n)$, inserire una valutazione di e alla fine del blocco n e rimuovere le valutazioni successive che essa copre.

Per ottimizzare ulteriormente:

- Se e appartiene a $VERYBUSY(n)$ per blocchi successivi, è importante inserirla nel blocco "giusto". Questo può essere:
  - L'ultimo blocco, per minimizzare la domanda di registri.
  - Il blocco meno frequentemente eseguito, per minimizzare il numero dinamico di valutazioni.
  - Altri criteri basati sulle priorità specifiche dell'ottimizzazione.

Trasformazione: *Hoisting*

La trasformazione di *hoisting* consiste nel valutare un'espressione e in un blocco b se:

- e è definita in ogni successore di b.
- La valutazione di e in b produce lo stesso risultato della sua valutazione successiva.

Questa tecnica consente di risparmiare spazio nel codice senza accorciare alcun cammino nel grafo di controllo.

Problema di Flusso di Dati: Very Busy Expressions

L'insieme $VERYBUSY(b)$ per un blocco b può essere calcolato tramite la seguente equazione:

$$VERYBUSY(b) = \bigcap_{s \in succ(b)} (UEEXPR(s) \cup (VERYBUSY(s) \cap \neg EXPRKILL(s)))$$

Dove:

- $UEEXPR(b)$ è l'insieme delle espressioni utilizzate in b prima di essere "uccise".
- $EXPRKILL(b)$ è l'insieme delle espressioni uccise in b prima di essere utilizzate.
- $succ(b)$ rappresenta i successori di b.

Per il nodo finale n_f, $VERYBUSY(n_f) = \emptyset$.

Proprietà delle Very Busy Expressions

L'analisi delle very busy expressions è un problema di flusso di dati **all'indietro**, simile all'analisi delle variabili vive (*LIVE*). La dimensione dell'insieme $VERYBUSY$ dipende dal numero di espressioni nel programma, ovvero $|VERYBUSY| = |expressions|$.

La forma della funzione di trasferimento è simile a quella utilizzata per l'analisi delle variabili vive:

$$f(x) = a \cup (b \cap c)$$

Conclusione

L'ottimizzazione tramite *hoisting* basata sull'analisi delle very busy expressions è particolarmente utile per ridurre lo spazio occupato dal codice, mantenendo invariata la semantica del programma. L'analisi backward garantisce che le espressioni identificate siano effettivamente utilizzate lungo tutti i cammini, evitando valutazioni ridondanti e migliorando l'efficienza complessiva del codice.

Propagazione delle Costanti (Classic Formulation)

La **propagazione delle costanti** è un'ottimizzazione globale che consente di determinare valori costanti per le variabili lungo i cammini del programma e di specializzare le computazioni in base a tali valori. Questa trasformazione, nota anche come **global constant folding**, permette di semplificare il codice riducendo le operazioni eseguite durante il tempo di esecuzione.

Trasformazione: Global Constant Folding

Un'ottimizzazione di constant folding consiste nel sostituire le variabili con valori noti quando è garantito che tali valori siano costanti lungo ogni cammino fino al punto di utilizzo. In questo modo, le computazioni vengono specializzate sulla base dei valori noti, riducendo il lavoro dinamico.

Problema di Flusso di Dati: Propagazione delle Costanti

La propagazione delle costanti è formulata come un problema di flusso di dati **in avanti**. Il dominio dell'analisi è costituito dall'insieme delle coppie $\langle v_i, c_i \rangle$, dove v_i è una variabile e $c_i \in C$ rappresenta il valore costante noto (se esiste).

L'insieme delle costanti per un blocco b è calcolato come:

$$CONSTANTS(b) = \bigcap_{p \in preds(b)} f_p(CONSTANTS(p))$$

Dove:

- $\bigcap$ rappresenta un'operazione di *meet* (intersezione) eseguita coppia per coppia sugli insiemi $\langle v_i, c_i \rangle$ provenienti dai predecessori di b.
- $f_p(x)$ è una funzione specifica del blocco p che modella gli effetti del blocco p sugli insiemi di costanti x.

La propagazione delle costanti si occupa di analizzare come le variabili e i loro valori cambiano lungo il flusso del programma, garantendo che i valori propagati siano coerenti con le operazioni eseguite in ciascun blocco.

Proprietà della Propagazione delle Costanti

1. **Problema di Flusso In Avanti**: L'informazione scorre dai predecessori ai successori nel grafo di controllo.

2. **Dominio delle Costanti**: $|CONSTANTS| = |variables|$, poiché ogni variabile può essere associata a un singolo valore costante.
3. **Funzione di Trasferimento (f)**: La funzione di trasferimento per i blocchi è più complessa rispetto ad altri problemi di flusso di dati. Essa tiene conto delle definizioni e degli usi delle variabili per aggiornare l'informazione relativa alle costanti.

Catene di Informazioni (Information Chains)

Con la proliferazione delle trasformazioni di flusso di dati, è emersa la necessità di un nuovo paradigma che semplificasse l'implementazione e riducesse il tempo speso nell'analisi. Questo ha portato allo sviluppo delle **catene di informazioni (information chains)**, un unico problema di flusso di dati generalizzato (GDFAP) che può essere applicato a molteplici trasformazioni.

Caratteristiche delle Catene di Informazioni

Una catena è una **tupla** che collega due eventi di flusso di dati, rappresentando direttamente le relazioni tra definizioni e usi delle variabili. Le catene forniscono una rappresentazione grafica e semplificano la ricerca saltando il codice non correlato.

Le catene possono essere costruite in modo efficiente e consentono di esprimere quattro tipi principali di relazioni di dipendenza:

- **DEF → USE**: Dipendenza vera o di flusso, in cui una definizione è utilizzata successivamente.
- **USE → DEF**: Dipendenza anti, in cui un uso precede una ridefinizione.
- **DEF → DEF**: Dipendenza in uscita, in cui una definizione segue un'altra definizione della stessa variabile.
- **USE → USE**: Dipendenza in ingresso, in cui un uso dipende da un altro uso.

Tra queste, le **catene DEF-USE** sono le più comuni e utili nell'ottimizzazione.

Notazione e Costruzione delle Catene

Per ogni operazione i e ogni variabile v, definiamo:

- $DEFS(v, i)$ come l'insieme delle operazioni che potrebbero aver definito v più recentemente prima dell'operazione i, lungo qualche cammino nel grafo di flusso di controllo (CFG).

- $USES(v, i)$ come l'insieme delle operazioni che potrebbero usare il valore di v calcolato in i, lungo qualche cammino nel CFG.

La relazione tra $DEFS$ e $USES$ è simmetrica:

$$x \in DEFS(A, y) \iff y \in USES(A, x)$$

Costruzione delle Catene DEF-USE

La costruzione delle catene DEF-USE si basa sulla risoluzione del problema delle **definizioni raggiungibili** (reaching definitions), che è un caso di GDFAP.

Definizione Raggiungibile

Una definizione d di una variabile v raggiunge un'operazione i se e solo se:

1. L'operazione i legge il valore di v.
2. Esiste un cammino v-clear da d a i, dove v-clear indica che non ci sono altre definizioni di v sul cammino.

Considerazioni Speciali

- Se esiste una definizione precedente di v nello stesso blocco, si ha $|DEFS(v, i)| = 1$.
- Se non c'è una definizione precedente, si ha $|DEFS(v, i)| \geq 1$, indicando che più definizioni possono raggiungere i.

Applicazioni delle Catene DEF-USE

Le catene DEF-USE sono alla base di numerose ottimizzazioni:

- **Propagazione delle costanti**: Utilizzare

Reaching Definitions

Le **Reaching Definitions** rappresentano un problema di flusso di dati fondamentale per analizzare le definizioni di variabili che raggiungono determinati punti nel programma. Questo problema è essenziale per ottimizzazioni come la costruzione di catene DEF-USE, l'eliminazione di codice morto e la rilevazione di anomalie.

Definizione Formale

L'insieme $REACHES(n)$ rappresenta le definizioni che raggiungono l'inizio del blocco n nel grafo di flusso di controllo (CFG). Le equazioni per calcolare $REACHES$ sono:

$$REACHES(n) = \bigcup_{p \in preds(n)} (DEDEF(p) \cup (REACHES(p) \cap \neg DEFKILL(p)))$$

Dove:

- $DEDEF(n)$ è l'insieme delle definizioni nel blocco n che raggiungono la fine del blocco.
- $DEFKILL(n)$ è l'insieme delle definizioni oscurate da nuove definizioni nel blocco n.
- $preds(n)$ rappresenta i predecessori del blocco n nel CFG.

Condizioni Iniziali

Per ogni nodo $n \in N$, inizialmente si assume:

- $REACHES(n) = \emptyset$, poiché nessuna definizione raggiunge alcun blocco all'inizio dell'analisi.

Calcolo di REACHES

L'insieme $REACHES$ può essere calcolato utilizzando qualsiasi metodo di analisi del flusso di dati, come i framework iterativi rapidi. Un approccio più efficiente, basato sull'algoritmo di Zadeck (*Incremental Data-Flow Analysis*), consente un calcolo in tempo lineare rispetto al numero di operazioni nel programma.

Il dominio dell'analisi è l'insieme delle definizioni, proporzionale al numero di operazioni nel programma.

Costruzione degli Insiemi DEFS

L'insieme $DEFS(v, i)$ identifica le definizioni della variabile v che possono raggiungere l'operazione i nel blocco corrente. Il calcolo procede come segue:

1. Identificare i blocchi base e costruire il CFG.
2. Per ogni blocco b, calcolare $REACHES(b)$.
3. Per ogni operazione i in b e ogni variabile referenziata v:
    - Se esiste una definizione precedente d di v nello stesso blocco, allora $DEFS(v, i) \leftarrow \{d\}$.
    - Altrimenti, $DEFS(v, i) \leftarrow \{d | d \text{ definisce } v \text{ e } d \in REACHES(b)\}$.

Costruzione degli Insiemi USES

Gli insiemi $USES$ possono essere costruiti invertendo $DEFS$ o risolvendo le *reachable uses* in modo analogo al calcolo di $REACHES$. Questo processo stabilisce i legami tra operazioni che utilizzano i valori prodotti da altre definizioni.

Applicazioni e Rilevazione delle Anomalie

Costruzione delle Catene DEF-USE

Le catene DEF-USE si costruiscono direttamente da $DEFS$ e $USES$, fornendo una rappresentazione compatta delle dipendenze tra definizioni e usi.

Rilevazione delle Anomalie

- Se $DEFS(v, i) = \emptyset$, l'operazione i utilizza una variabile mai inizializzata.
- Per rilevare un set più ampio di anomalie, è possibile aggiungere una definizione iniziale per ogni variabile v al nodo iniziale n_0. Se questa definizione è presente in $DEFS(v, i)$, significa che esiste un cammino verso i senza inizializzazione di v.

Considerazioni Finali

L'analisi delle **Reaching Definitions** è cruciale per ottimizzazioni e rilevazione di errori nei programmi. La sua implementazione efficiente e la compatta rappresentazione degli insiemi $DEFS$ e $USES$ ne fanno un pilastro dell'ottimizzazione nei compilatori moderni.