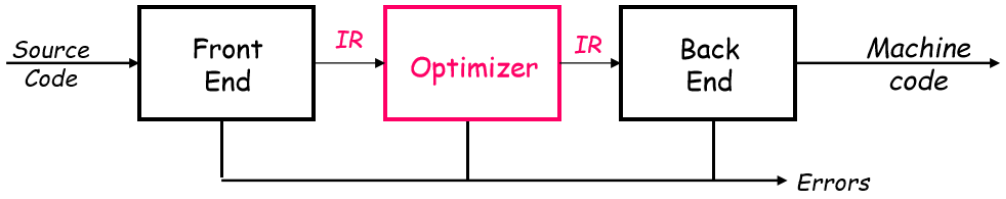
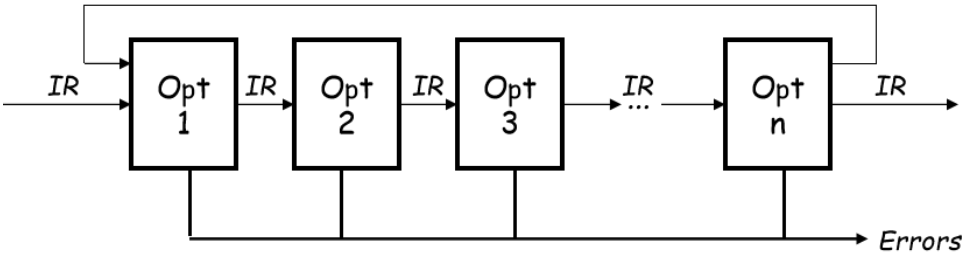
13. Code Optimization

L'ottimizzazione (o miglioramento del codice) analizza e riscrive (o trasforma) il codice intermedio (IR).



L'obiettivo principale è ridurre il tempo di esecuzione del codice compilato, ma può anche migliorare altri aspetti come lo spazio di memoria occupato, il consumo di energia e altri fattori di efficienza. È però fondamentale che l'ottimizzazione preservi il "significato" del codice originale, il quale è misurato in base ai valori delle variabili nominate.

L'ottimizzatore



Modern optimizers are structured as a series of passes

Le trasformazioni tipiche effettuate dall'ottimizzatore includono la scoperta e la propagazione di valori costanti, lo spostamento di calcoli in posizioni meno frequentemente eseguite (e.g. *loop invariant code motion*, spostamento di codice invariante al di fuori di un ciclo, per evitare di ricalcolarlo ad ogni iterazione) e la specializzazione di calcoli in base al contesto. L'ottimizzatore individua anche calcoli ridondanti per rimuoverli e elimina codice inutile o non raggiungibile, oltre a codificare determinate espressioni in una forma particolarmente efficiente.

Il compilatore può implementare una procedura in molti modi differenti, e l'ottimizzatore cerca di trovare una versione "migliore" in termini di velocità, dimensione del codice, spazio di dati e altri fattori di efficienza. Per raggiungere questo scopo, analizza il codice per ottenere conoscenza sul comportamento durante l'esecuzione, utilizzando tecniche come l'analisi del flusso dei dati e la disambiguazione dei puntatori; questi metodi rientrano nella

cosiddetta “analisi statica”. Questa conoscenza viene poi utilizzata per tentare di migliorare il codice. Esistono letteralmente centinaia di trasformazioni proposte, con una grande sovrapposizione tra loro. Tuttavia, non esiste nulla di realmente “ottimale” nell’ottimizzazione, poiché le dimostrazioni di ottimalità si basano su condizioni restrittive e poco realistiche.

Esempio Importante. Ridondanza di un'espressione

Un’espressione $x + y$ è considerata ridondante se e solo se, lungo ogni percorso dall'entrata della procedura, è stata già valutata e i suoi sotto-elementi costitutivi x e y non sono stati ridefiniti. Se il compilatore può dimostrare che un’espressione è ridondante, può conservare i risultati di valutazioni precedenti e sostituire la valutazione corrente con un riferimento.

Il problema si divide in due parti:

1. Dimostrare che $x + y$ è ridondante, o disponibile
2. Riscrivere il codice per eliminare la valutazione ridondante

Una tecnica che permette di ottenere entrambi questi obiettivi è chiamata **value numbering**.

Value Numbering

Concetto chiave. Assegnare un numero identificativo, $V(n)$, a ciascuna espressione, in modo tale che $V(x + y) = V(j)$ se e solo se $x + y$ e j hanno sempre lo stesso valore (all'interno di un blocco di base; la definizione diventa più complessa quando si considerano più blocchi). Per rendere efficiente questo processo, si utilizza una tecnica di hashing sui numeri di valore.

Uso dei numeri di valore per migliorare il codice:

- **Sostituzione delle espressioni ridondanti:** se due espressioni hanno lo stesso numero di valore (VN), è possibile fare riferimento alla valutazione precedente anziché ricalcolarla.
- **Semplificazione delle identità algebriche** e scoperta delle espressioni a valore costante per eseguire folding e propagazione.

Questa tecnica è progettata per IR (Intermediate Representations) a basso livello e lineari; esistono metodi simili per strutture ad albero, come la costruzione di un DAG (Directed Acyclic Graph).

Algoritmo.

Per ciascuna operazione $o = \langle \text{operatore}, o_1, o_2 \rangle$ all'interno di un blocco, eseguita in ordine:

1. Ottieni i numeri di valore degli operandi tramite una ricerca hash.
2. Calcola un numero di valore per o attraverso l'hash di $\langle \text{operatore}, VN(o_1), VN(o_2) \rangle$.
3. Se o ha già un numero di valore, sostituiscilo con un riferimento.
4. Se o_1 e o_2 sono costanti, valuta l'operazione e sostituiscila con un'istruzione di caricamento immediato `loadI`.

Quando il comportamento dell'hashing è ottimale, l'algoritmo opera in tempo lineare; altrimenti, si possono utilizzare tecniche di discriminazione dei multi-insiemi o automi DFA aciclici.

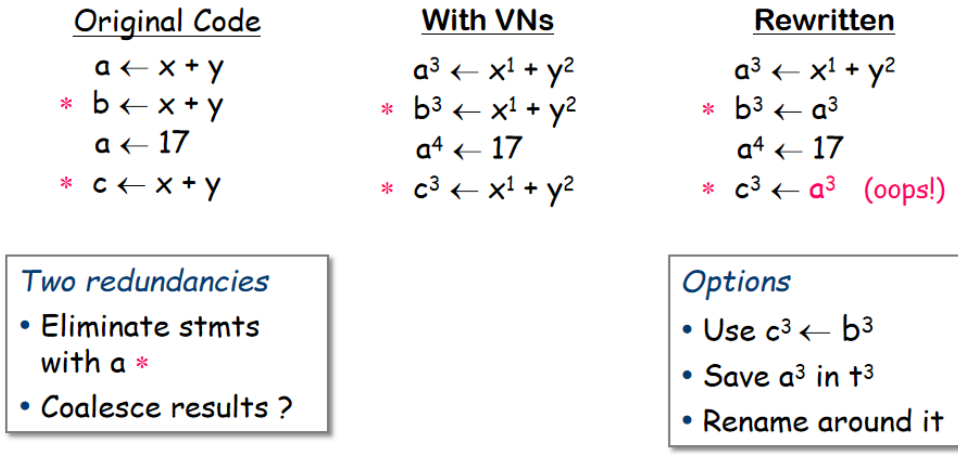
Gestione delle identità algebriche: attraverso una struttura condizionale che esamina il tipo di operatore, trattando i casi speciali per ciascun operatore.

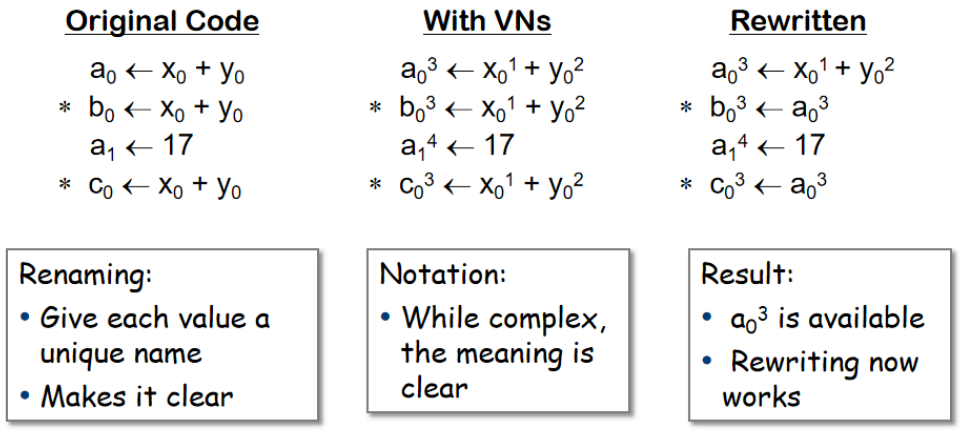
Complessità (Asintotica) e Problemi di Velocità (Costanti)

Nell'algoritmo, ci sono vari punti critici in termini di complessità e velocità. Il comando "ottenere i numeri di valore" può implicare una ricerca lineare oppure un'operazione di hashing, influenzando il tempo di esecuzione. Similmente, l'operazione di hashing su $\langle \text{operatore}, VN(o_1), VN(o_2) \rangle$ può essere una ricerca lineare o basarsi su un hash diretto.

Copy folding. Questa tecnica implica l'assegnazione del numero di valore direttamente al risultato, riducendo la necessità di ricalcoli.

Operazioni commutative. Queste richiedono di scegliere tra un doppio hashing o l'ordinamento degli operandi per garantire coerenza; il metodo scelto incide sull'efficienza dell'algoritmo.





Estensioni Semplici del Value Numbering

Constant Folding. Aggiungere un flag che registri se un valore è costante consente di valutare tali valori a tempo di compilazione. Questi possono essere sostituiti con un'istruzione di caricamento immediato `load immediate` o un operando immediato. Non esiste un algoritmo locale più efficace di questo per la gestione dei valori costanti.

Identità Algebriche. È necessario verificare numerosi casi speciali, sostituendo il risultato con il numero di valore dell'input, se applicabile. Per ottimizzare il processo, è utile costruire un albero decisionale basato sul tipo di operazione.

Ambito dell'ottimizzazione

Nel contesto di scansione e parsing, il termine "ambito" si riferisce a una regione di codice che corrisponde a uno spazio dei nomi distinto. Nell'ottimizzazione, invece, "ambito" rappresenta una regione di codice soggetta ad analisi e trasformazione. Sebbene i concetti siano collegati, la loro connessione non è necessariamente intuitiva. Diversi ambiti introducono sfide e opportunità uniche. Storicamente, l'ottimizzazione è stata eseguita su vari ambiti distinti, ciascuno con le proprie peculiarità e obiettivi.

Un *blocco base* è una sequenza di codice lineare di lunghezza massima.

Ottimizzazione Locale. Agisce interamente all'interno di un singolo blocco di base. Le proprietà di un blocco isolato permettono ottimizzazioni particolarmente efficaci.

Ottimizzazione Regionale. Si applica a una regione nel grafo di controllo del flusso (CFG) che include più blocchi, come loop, alberi, percorsi o blocchi di base estesi.

Ottimizzazione dell'Intera Procedura (Intraprocedurale). Opera sull'intero CFG di una procedura. La presenza di percorsi ciclici richiede prima un'analisi approfondita, seguita

dalla trasformazione del codice.

Ottimizzazione dell'Intero Programma (Interprocedurale). Interviene su parte o sull'intero grafo delle chiamate (composto da più procedure), affrontando anche le complessità introdotte dalle chiamate di funzione, dai ritorni e dal binding dei parametri.

Esempio. Comp 412

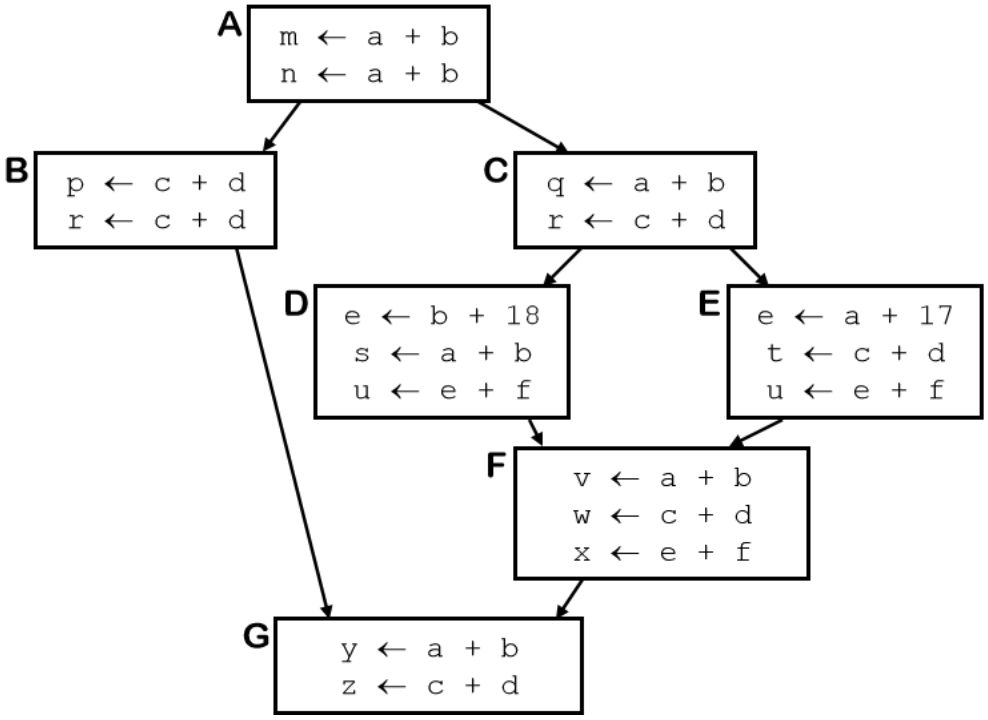
Sarebbe bello pensare che l'ottimizzazione si sia sviluppata in modo ordinato: prima i metodi locali hanno portato ai metodi regionali, i metodi regionali hanno condotto ai metodi globali e infine i metodi globali hanno aperto la strada ai metodi interprocedurali. Tuttavia, le cose non sono andate così. Il primo compilatore, FORTRAN, utilizzava sia metodi locali sia metodi globali, e lo sviluppo dell'ottimizzazione è stato frammentario e spesso simultaneo. L'ambito dell'ottimizzazione sembra essere più legato al tipo di inefficienza che si vuole affrontare, piuttosto che a una progressiva evoluzione da parte degli inventori.

Ecco la traduzione e riscrittura migliorata in formato Obsidian:

Superlocal Value Numbering

La **Superlocal Value Numbering** è una tecnica regionale applicata agli **Extended Basic Blocks** (EBB). Un EBB è definito come un insieme massimo di blocchi $B_1, B_2, \ldots, B_n$ in cui ogni blocco B_i, eccetto B_1, ha esattamente un predecessore, e quel predecessore è contenuto all'interno dell'EBB stesso.

Ad esempio, l'insieme di blocchi $\{A, B, C, D, E\}$ costituisce un EBB.



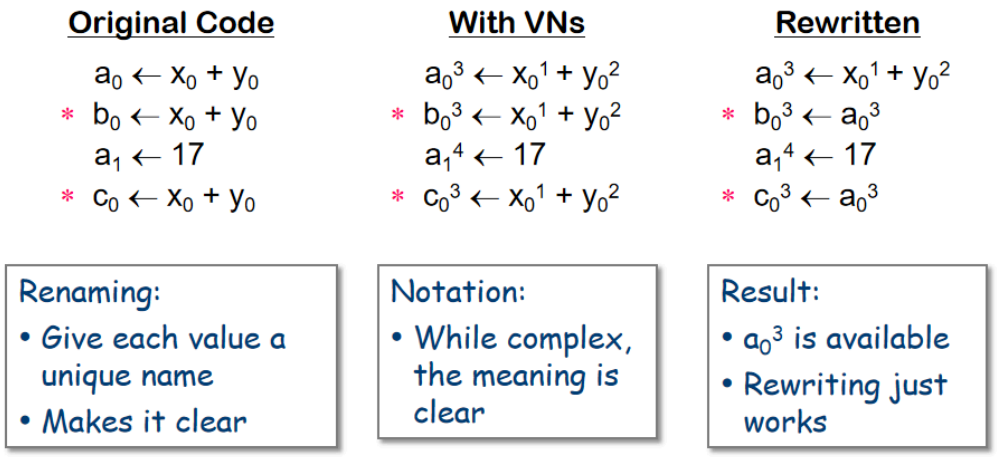
Questo EBB presenta tre percorsi possibili: (A, B), (A, C, D)e (A, C, E). In alcuni casi, è possibile trattare ciascun percorso come se fosse un singolo blocco. I blocchi $\{F\}$ e $\{G\}$ sono esempi di EBB degenerati.

La tecnica **superlocale** si riferisce all'applicazione della value numbering a un EBB. Il concetto fondamentale consiste nell'applicare il metodo locale ai percorsi attraverso gli EBB, come nei percorsi $\{A, B\}$, $\{A, C, D\}$ e $\{A, C, E\}$. Questo approccio consente di riutilizzare i risultati dagli antenati comuni, evitando di rianalizzare i blocchi A e C. Tuttavia, questa tecnica non apporta benefici ai blocchi degenerati F o G.

Efficienza

Per rendere il processo efficiente, si può usare la tabella di A per inizializzare le tabelle di B e C. Per evitare duplicazioni, è utile adottare una tabella hash con ambito, organizzata come A, AB, A, AC, ACD, AC, ACE, F, G. È inoltre necessario mantenere una mappatura da **VN** a nome per gestire le eliminazioni (un “kill” rappresenta una ridefinizione di un nome) e poter ripristinare la mappatura in base all’ambito. Questo approccio aggiunge complessità al sistema senza incidere sui costi.

Per semplificare ulteriormente, è utile disporre di un nome unico per ciascuna definizione, il che rende il nome direttamente associabile a **VN**. A tale scopo, si utilizza lo **spazio dei nomi SSA** (Static Single Assignment), dove i nomi con apici o indici rappresentano versioni uniche di ogni variabile, come nell’esempio degli indici subscript precedenti, che costituiscono un’istanza dello spazio dei nomi SSA.



Spazio dei Nomi SSA

Nel **SSA Name Space** (Static Single Assignment), si seguono due principi fondamentali:

1. Ogni nome viene definito esattamente da un'unica operazione.
2. Ogni operando si riferisce esattamente a una sola definizione.

Per conciliare questi principi con il codice reale, è necessario inserire le **funzioni** ϕ nei punti di unione del flusso di controllo, consentendo di unificare lo spazio dei nomi. Inoltre, si aggiungono pedici ai nomi delle variabili, garantendo così l'unicità di ciascun nome.

Loop Unrolling

La **Loop Unrolling** è una tecnica regionale che si basa sull'osservazione che molte applicazioni trascorrono gran parte del tempo di esecuzione all'interno di loop. Riducendo il sovraccarico del loop mediante lo srotolamento, è possibile eliminare operazioni aggiuntive, test e ramificazioni, rendendo così il codice risultante suscettibile di ottimizzazioni locali più efficaci. Questa tecnica funziona solo quando i limiti del loop sono fissi e il numero di iterazioni è relativamente ridotto.

Il principio alla base è valido e, se i limiti sono impostati correttamente, l'unrolling risulta sempre sicuro.

Srotolamento parziale del loop

Lo srotolamento del loop con fattori più piccoli permette di ottenere gran parte dei benefici dell'unrolling completo, riducendo l'aumento della dimensione del codice. Questo approccio consente di ridurre test e ramificazioni fino al 25%, limitando l'overhead per ogni operazione utile. Inoltre, la **Local Value Numbering** (LVN) può eliminare le aggiunte duplicate e le

espressioni ridondanti, migliorando ulteriormente l'efficienza. Tuttavia, questa tecnica dipende dalla conoscenza precisa dei limiti del loop.

Srotolamento con limiti sconosciuti

Quando si effettua lo srotolamento di un loop con limiti sconosciuti, è necessario generare dei **guard loops** (loop di protezione) per garantire la corretta esecuzione. Questo approccio consente di ottenere la maggior parte dei risparmi, riducendo i test e le ramificazioni fino al 25%. La **Local Value Numbering** (LVN) continua a funzionare all'interno del corpo del loop, contribuendo a eliminare espressioni ridondanti. Sebbene il guard loop occupi una certa quantità di spazio, questo metodo può essere generalizzato per supportare limiti superiori e inferiori arbitrari, così come diversi fattori di srotolamento.

Un altro trucco per lo srotolamento

Un'altra tecnica di srotolamento consiste nell'eliminare le copie alla fine di un loop. Questo si può ottenere srotolando il loop in base al **minimo comune multiplo (MCM)** dei cicli di copia. In questo modo, le copie, che rappresentavano semplicemente un artificio di denominazione, vengono eliminate, ottenendo alcuni dei vantaggi dello srotolamento: minore overhead e blocchi più lunghi, che sono più facili da ottimizzare localmente. Questa situazione si verifica in più casi di quanto si possa sospettare, offrendo benefici significativi in termini di efficienza.

Ricerca di Variabili Non Inizializzate

La ricerca di variabili non inizializzate è una tecnica globale che sfrutta l'analisi del flusso di dati per individuare variabili che potrebbero essere utilizzate prima di essere definite. Una variabile v è considerata **live** in un punto p se e solo se esiste un percorso nel grafo di controllo del flusso (CFG) da p a un uso di v lungo il quale v non viene ridefinita. Qualsiasi variabile live nel blocco di ingresso di una procedura può essere usata prima di essere definita, rappresentando un potenziale errore logico nel codice.

L'analisi del flusso di dati è una forma di ragionamento a tempo di compilazione sul flusso dei valori a runtime. Il compilatore dovrebbe individuare e segnalare questa condizione per aiutare nella rilevazione di errori logici. Esaminiamo questa problematica poiché ci offre l'opportunità di introdurre il calcolo della **liveness**.

Il compilatore costruisce un CFG e calcola $LiveOut(n)$ per ogni nodo n. $LiveOut$ è un problema classico nell'analisi del flusso di dati e rappresenta uno strumento fondamentale per identificare variabili potenzialmente non inizializzate.

Variabili live

I problemi di flusso di dati possono essere espressi attraverso un sistema di equazioni simultanee su insiemi associati ai nodi di un grafo. Nel caso delle variabili live, le equazioni sono definite come segue:

$$\mathrm{LiveOut}(n_f) = \emptyset$$

$$\mathrm{LiveOut}(n) = \bigcup_{m \in \mathrm{succ}(n)} (\mathrm{UEVar}(m) \cup (\mathrm{LiveOut}(m) \cap \mathrm{VarKill}(m)))$$

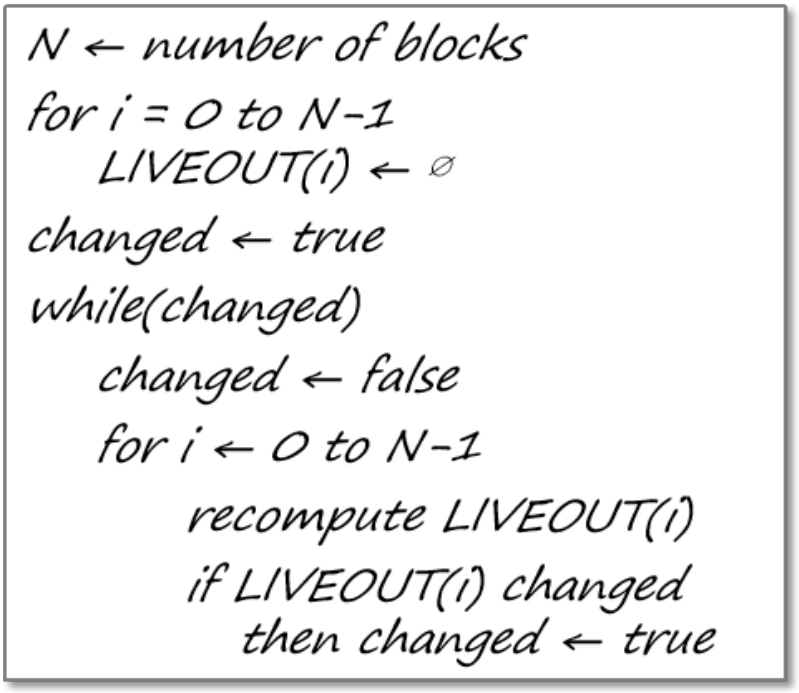
Dove:

- $\mathrm{UEVar}(n)$ è l'insieme dei nomi utilizzati prima di essere definiti nel blocco corrispondente al nodo n nel grafo di controllo del flusso (CFG).
- $\mathrm{VarKill}(n)$ è l'insieme dei nomi definiti nel blocco corrispondente al nodo n nel CFG.

Queste equazioni permettono di annotare ciascun nodo n del CFG con un insieme $LiveOut$, rappresentando le variabili che sono ancora live all'uscita del nodo.

Calcolo delle Informazioni di Liveness per una Procedura

Per calcolare le informazioni di **liveness** di una procedura, è necessario seguire alcuni passaggi chiave. Innanzitutto, si costruisce il **grafo di controllo del flusso** (CFG). Successivamente, si calcolano gli insiemi **UEVar** e **VarKill** per ogni blocco nel CFG. Infine, si utilizza un risolutore iterativo a punto fisso per calcolare gli insiemi **LiveOut**.



Il risolutore iterativo a punto fisso funziona nel seguente modo:

- $LiveOut$ è un sottoinsieme di 2^{Names}, ovvero l'insieme delle possibili combinazioni delle variabili.
- $UEVar$ e $VarKill$ sono insiemi costanti, una volta calcolati.

- L’equazione è monotona crescente, il che significa che i valori non possono diminuire durante l’iterazione.

Poiché gli insiemi sono finiti e le equazioni sono monotone, l'algoritmo deve terminare. La teoria dell'analisi del flusso di dati garantisce che questa equazione abbia una soluzione a punto fisso unica, permettendo di calcolare correttamente gli insiemi $LiveOut$.

Considerazioni

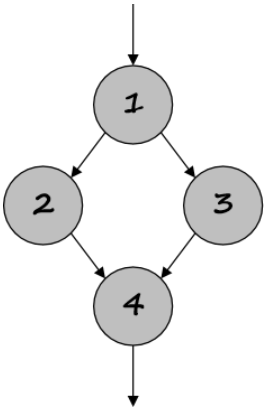
Gli insiemi $LiveOut$ possono essere utilizzati per individuare variabili non inizializzate e per eliminare memorizzazioni superflue. Se una variabile $x \in \mathrm{LiveOut}(n_0)$, significa che x è non inizializzata in qualche punto di utilizzo. Inoltre, è possibile utilizzare gli insiemi $LiveOut$ per costruire lo stato di $Live$ per ogni operazione. Se una variabile $x \notin \mathrm{Live}$ in un’operazione di memorizzazione, significa che il valore non viene mai ricaricato, rendendo quella memorizzazione non necessaria.

Le equazioni di $LiveOut$ hanno una soluzione unica a punto fisso, e l'algoritmo trova questa soluzione a punto fisso; poiché la soluzione è unica, l'algoritmo individua la soluzione corretta. La velocità di convergenza dipende dall'ordine di calcolo scelto, e un ordine adeguato può consentire di raggiungere rapidamente il punto fisso.

Analisi data-flow

L'ordine di calcolo influenza la velocità di convergenza dell'algoritmo, in particolare perché $Live$ è un problema di flusso di dati all’indietro. Gli insiemi $LiveOut$ per un nodo n vengono calcolati a partire dagli insiemi dei successori di n nel grafo di controllo del flusso (CFG).

Ad esempio, $LiveOut(1)$ dipende da $LiveOut(2)$ e $LiveOut(3)$, che a loro volta dipendono da $LiveOut(4)$.

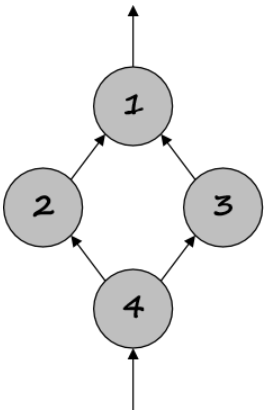


Di conseguenza, il risolutore dovrebbe visitare prima i nodi 2 e 3 rispetto al nodo 1, e il nodo 4 prima dei nodi 2 e 3.

L'idea generale è di aggiornare quante più "sorgenti" possibile prima di visitare un nodo specifico, permettendo così a una modifica nell'insieme $LiveOut$ di propagarsi il più lontano possibile nel CFG in un singolo passaggio del ciclo $while$. In questo modo, l'algoritmo raggiunge più rapidamente il punto fisso, ottimizzando la velocità di convergenza.

L'ottimizzazione del codice è strettamente legata alla teoria dei grafi. Il calcolo dell'insieme $LiveOut$ viene effettuato, concettualmente, sul **grafo di controllo del flusso inverso** (RCFG). Pertanto, l'ordine per il risolutore è definito dal RCFG.

Per un problema di propagazione all'indietro, l'ordine desiderato di visita dei nodi sarebbe: prima il nodo 4, poi i nodi $\{2, 3\}$, e infine il nodo 1.



L'ordine posticipato (PostOrder) sul RCFG sarebbe: $1, \{2, 3\}, 4$. L'ordine desiderato per il calcolo è quindi il **reverse postorder** (RPO), ovvero l'inverso del postorder, risultando in: $4, \{3, 2\}, 1$.

La formula per calcolare l'RPO è: $\mathrm{RPO}(i) = |N| + 1 - \mathrm{PO}(i)$, dove $|N|$ è il numero di nodi e $\mathrm{PO}(i)$ è il postorder del nodo i. In questo contesto, non importa l'ordine specifico tra i nodi 2 e 3.

Per i problemi di propagazione in avanti, utilizziamo l'RPO direttamente sul CFG. Per i problemi di propagazione all'indietro, come in questo caso, applichiamo l'RPO sul RCFG. È importante notare che l'RPO sul RCFG non è lo stesso dell'ordine preorder inverso sul CFG.

Riepilogo: variabili live

Definizione del problema: una variabile v è considerata **live** in un punto p se e solo se esiste un percorso nel **grafo di controllo del flusso** (CFG) da p a un uso di v lungo il quale v non viene ridefinita.

Risoluzione delle equazioni per $LiveOut$

$$\mathrm{LiveOut}(n_f) = \emptyset$$

$$\mathrm{LiveOut}(n) = \bigcup_{m \in \mathrm{succ}(n)} (\mathrm{UEVar}(m) \cup (\mathrm{LiveOut}(m) \cap \mathrm{VarKill}(m)))$$

Per risolvere queste equazioni, si utilizza un **risolutore iterativo a punto fisso**. La teoria garantisce che questo problema abbia una soluzione unica, e la scelta di un ordine di calcolo efficiente consente di raggiungere rapidamente questa soluzione.

Gli insiemi $LiveOut$ risultanti possono essere usati per vari scopi, tra cui identificare variabili non inizializzate, eliminare memorizzazioni inutili, individuare i range di vita delle variabili e molto altro ancora.