

# Optimierungen in Übersetzern: Verfahren

Marco Ammon (my04mivo)

12. August 2020

## Inhaltsverzeichnis

<b>1</b>	<b>Kontrollflussanalyse</b>	<b>2</b>
1.1	Kontrollflussgraph . . . . .	2
1.2	Dominanz . . . . .	2
1.2.1	Berechnung der Dominatoren $D(n)$ eines Knoten $n$ . . . . .	2
1.2.2	Dominanzgrenze . . . . .	3
1.3	Schleifenerkennung . . . . .	3
<b>2</b>	<b>Datenflussanalyse</b>	<b>4</b>
2.1	Berechnung von Datenflusswissen . . . . .	4
2.2	Typische Datenflussprobleme . . . . .	5
<b>3</b>	<b>Wertnummerierung in Grundblock</b>	<b>6</b>
<b>4</b>	<b>Static Single Assignment (SSA)</b>	<b>7</b>
4.1	Konstruktion der SSA-Form . . . . .	7
4.1.1	Iteratives Dominanzgrenzenverfahren nach Cytron ( $\mathcal{O}(N^2)$ ) . . . . .	7
4.1.2	Wertnummerierungsverfahren nach Click . . . . .	7
4.2	Optimierungen auf SSA-Form . . . . .	7
4.3	Rücktransformation aus SSA-Form . . . . .	8
<b>5</b>	<b>Alias-Analyse</b>	<b>8</b>
5.1	DFA-Ansatz zur intraprozeduralen, fluss-sensitiven may-Analyse nach Muchnick	9
5.2	Standard-Verfahren zur interprozeduralen, fluss-insensitiven may-Analyse . . . . .	10
5.3	Steensgards Algorithmus zur interprozeduralen, fluss-insensitiven may-Analyse . . . . .	11
<b>6</b>	<b>Schleifen</b>	<b>11</b>
6.1	Schleifeninvariante Anweisungen . . . . .	11
6.2	Induktionsvariablen . . . . .	12
6.3	Datenabhängigkeiten in Schleifen und Arrays . . . . .	13
6.4	Indexanalyse . . . . .	15
6.5	Entfernung von schleifengetragenen Datenabhängigkeiten . . . . .	15
6.5.1	Schleifentransformationen . . . . .	16
6.5.2	Schleifenrestrukturierungen . . . . .	16

# 1 Kontrollflussanalyse

## 1.1 Kontrollflussgraph

- Gerichteter Graph
- Knoten: Grundblöcke (meist maximal)
- Kante zwischen zwei Blöcken  $A$  und  $B$  wenn  $B$  direkt nach  $A$  ausgeführt werden kann (etwa [un-]bedingter Sprung oder Fallthrough)
- Synthetische Ergänzung um Entry- und Exit-Knoten, die mit Kante verbunden sind
- Kontrollflussabhängigkeit: Bei Verzweigungsknoten  $v$  mit direkten Nachfolgern  $a$  und  $b$ :  $y$  kontrollflussabhängig von  $v \Leftrightarrow$  mindestens ein Pad von  $a$  zum Exit-Knoten ohne  $y$  und jeder Pfad von  $b$  zum Exit-Knoten über  $y$

## 1.2 Dominanz

- Knoten  $x$  dominiert  $y$  ( $x \geq y$ ), wenn jeder Pfad von Wurzel zu  $y$  durch  $x$  laufen muss
- Strikte Dominanz  $x \gg y$ , falls zusätzlich  $x \neq y$  gilt
- $\text{ImmDom}[y]$  ist strikter Dominator von  $y$ , der  $y$  am Nächsten ist
- Dominatorbaum enthält jeden Knoten als Kind seines  $\text{ImmDomms}$   $\rightarrow$  Pfad zwischen  $x$  und  $z$  in Dominatorbaum  $\Leftrightarrow x \gg z$

### 1.2.1 Berechnung der Dominatoren $D(n)$ eines Knoten $n$

#### Iterativer Fixpunkt-Algorithmus (Lengauer)

- mit  $\mathcal{O}(|E||N|^2)$
- Zunächst Überapproximation der Dominatorenmenge
- Initialisierung aller  $D(n) \in N$  mit  $N$  außer Startknoten  $S$  mit  $D(S) = S$
- Bis Fixpunkt erreicht ist: alle  $D(n)$  zu  $D'(n) = \{n\} \cup \bigcap_{(p,n) \in E} D(p)$
- $n$  am besten in Tiefensuchereihenfolge durchlaufen

#### Verfahren mit Spannendem Tiefenbaum $T$

- Besuch des KFG in Tiefensuchereihenfolge mit zugehöriger Nummerierung:
  - „Spannende“ Kanten gehen zu frisch nummerierten Knoten
  - Rückschreitende Kanten gehen zu Vorgänger (kleinere DFS-Nummer) in  $T$
  - Fortschreitende Kanten gehen zu Nachfolger (größere DFS-Nummer) in  $T$
  - Kreuzkanten führen in früher besuchten Ast in  $T$
- Dominatoren  $D(n)$  liegen auf jeden Fall „über“  $n$  in  $T$
- Berechnung der Semidominatoren  $\text{SemDom}[w]$  in Reihenfolge fallender DFS-Nummern:
  - Direkte Vorgänger auf  $T$  sind Kandidaten
  - $\min_{u \in \text{Pred}(w)} \text{SemDom}[u]$  ist Kandidat
  - Minimum der Kandidaten ist  $\text{SemDom}[w]$

- Berechnung von  $\text{ImmDom}[w]$  durch Durchlaufen in Tiefenordnung von  $\text{SemDom}[w]$  nach  $w$ :

- Jeweils alle Vorgänger  $u$  untersuchen und  $u$  mit kleinstem  $\text{SemDom}[u]$  finden

–

$$\text{ImmDom}[w] = \begin{cases} \text{SemDom}[u] & \text{falls } \text{SemDom}[w] = \text{SemDom}[u] \\ \text{ImmDom}[u] & \text{sonst} \end{cases}$$

### 1.2.2 Dominanzgrenze

- Dominanzgrenze  $DG[x]$  enthält Knoten  $y$ , die einen von  $x$  dominierten Vorgänger besitzen, aber nicht von  $x$  streng dominiert werden
- Berechnung der  $DG[x]$ :

$$DG[x] = DG_{\text{local}}[x] \cup \bigcup_{z \in N, \text{ImmDom}[z]=x} DG_{\text{up}}[x, z]$$

$$DG_{\text{local}}[x] = \{y \in \text{Succ}(x) \mid \text{ImmDom}[y] \neq x\}$$

$$DG_{\text{up}}[x, z] = \{y \in DG[z] \mid \text{ImmDom}[y] \neq x\}$$

- Invertierung der Dominanzgrenzen liefert Kontrollflussabhängigkeiten

### 1.3 Schleifenerkennung

- Region:
    - Untergraph mit einem „Header“  $d$ , der (potentiell mehrere) Eingangskante von außerhalb besitzt
    - Wichtige Region: maximale Region mit  $d$  dominiert alle Knoten der Region
    - Hierarchischer Flussbaum: Baum der Regionen
  - Rückwärtskante: Kante  $(n, d)$  mit  $d \geq n$
  - Natürliche Schleife:
    - Rückwärtskante  $(n, d)$  sowie alle Knoten  $k$  mit  $d \geq k$  und es gibt einen Pfad von  $k$  nach  $n$  ohne  $d$
    - Bestimmung mit Worklist-Algorithmus, der bei  $n$  beginnt und rekursiv die Vorgänger bis  $d$  durchläuft und in Menge aufnimmt
  - Suche nach Rückwärtskanten und natürlichen Schleifen in wichtigen Regionen ausreichend
  - „Unsaubere“ Regionen:
    - ein Knoten dominiert nachgeordneten Zyklus
    - Erkennung durch Prüfung der Reduzierbarkeit des Graphs:
      - \* Entfernung der Rückwärtskanten aus KFG  $\rightarrow$  azyklischer Graph, in dem jeder Knoten von der Wurzel erreicht werden kann  $\Leftrightarrow$  KFG frei von unnatürlichen Schleifen
      - \* Alternative mit Transformationen: Am Ende Graph aus einem einzigen Knoten  $\Leftrightarrow$  KFG reduzierbar (ohne Zyklen)
- T1-Transformation** Selbstschleifen aus Graph löschen
- T2-Transformation** Knoten mit eindeutigem Vorgänger mit diesem zusammenfassen

## 2 Datenflussanalyse

- Datenabhängigkeiten verhindern Umordnen:
  - Schreiben vor Lesen: Flussabhängigkeit  $\delta^f$
  - Schreiben vor Schreiben: Ausgabeabhängigkeit  $\delta^o$
  - Lesen vor Schreiben: Antiabhängigkeit  $\delta^a$
- Starke Variablendefinition: sichere Zuweisung zu einer Variablen
- Schwache Variablendefinition: mögliche Zuweisung zu einer Variablen (etwa über Zeiger oder Referenz, die möglicherweise auf Variable zeigen)

### 2.1 Berechnung von Datenflusswissen

- Funktion pro Grundblock:
  - Eingabe: bei Vorwärtsproblem (Rückwärtsproblem) Datenflusswissen der Vorgängerknoten (Nachfolgerknoten)
  - Vorverarbeitung: logische Operationen oder Mengenoperationen
    - \* sicher (must): Eigenschaft muss auf allen Eingangskanten erfüllt sein
    - \* möglich (may): Eigenschaft muss auf mindestens einer Eingangskante erfüllt sein
  - Ausgabe: auf Eingabe und in Grundblock enthaltenen Befehlen basierendes, aktualisiertes Datenflusswissen
- Iterativer Fixpunkt-Algorithmus für Vorwärtsproblem: optimaler Besuch in Reihenfolge des spannenden Tiefenbaums

```
in(Entry) ← Init
for all  $n \in N \setminus \{\text{Entry}\}$  do
  in( $n$ ) ←  $\perp$ 
end for
WL ←  $N \setminus \{\text{Entry}\}$ 
while WL  $\neq \emptyset$  do
   $B \leftarrow \text{pop}(\text{WL})$ 
  out ←  $f_B(\text{in}(B))$ 
  for all  $B' \in \text{Succ}(B)$  do
    in( $B'$ ) ← in( $B'$ )  $\sqcup$  out
    if out  $\neq$  out( $B$ ) then
      WL ← WL  $\cup \{B'\}$ 
      out( $B$ ) ← out
    end if
  end for
end while
```
- Probleme bei Verwendung von Bitvektoren:
  - Transformation erfordert komplettes Neuberechnen
  - Bitvektoren oft zu groß für jeweilige Nutzungsstelle
- Alternative Datenstrukturen:
  - Definitions-Nutzungs-Graph für erreichbare Nutzungen
  - Nutzungs-Definitions-Graph für erreichenden Definitionen
  - Variablen-Netz als Vereinigung aller sich schneidenden DU-Graphen

- Single Static-Assignment (SSA)

## 2.2 Typische Datenflussprobleme

- Erreichende Definitionen:
  - mindestens ein Pfad, auf dem Variable nicht erneut schwach definiert wird
  - Optimierungen:
    - \* Keine erreichende Definition  $\rightarrow$  Variable nicht initialisiert
    - \* Genau eine oder gleiche Definition  $\rightarrow$  Konstantenweitergabe oder Kopienfortschreibung möglich
    - \* Alle Variablen nur außerhalb einer Schleife definiert  $\rightarrow$  Ausdruck schleifeninvariant
  - Umsetzung:
    - \* Eine Bitposition pro Variablendefinition
    - \* Setzen bei Definition der Variable
    - \* Zurücksetzen bei mindestens schwacher Definition
    - \* Anfangsbelegung: false
    - \* Vorverarbeitung: oder
- Konstantenweitergabe:
  - Eine Bitposition plus Wert pro Variable
  - Setzen von Bit und Wert bei konstanter Definition
  - Zurücksetzen von Bit bei mindestens schwacher oder nicht konstanter Definition
  - Anfangsbelegung: false, ?
  - Vorverarbeitung: und sowie Wertgleichheit
- Kopienfortschreibung:
  - Eine Bitposition pro aus Kopieren entstandener Wertgleichheitsbeziehung
  - Setzen bei Kopieroperation
  - Zurücksetzen wenn Original oder Kopie mindestens schwach definiert wird
  - Anfangsbelegung: false
  - Vorverarbeitung: und
- Verfügbare Ausdrücke:
  - auf allen Pfaden von Entry-Knoten aus wird Ausdruck bestimmt und verwendete Variablen anschließend nicht mehr schwach definiert
  - Optimierungen:
    - \* Elimination gleicher Teilausdrücke
    - \* Wiederverwendung statt Neuberechnung
  - Umsetzung:
    - \* Eine Bitposition pro wertnummeriertem (Teil-)Ausdruck
    - \* Setzen wenn Ausdruck ausgewertet wird
    - \* Zurücksetzen wenn vorkommende Variable mindestens schwach definiert wird

- \* Anfangsbelegung: false
- \* Vorverarbeitung: und sowie Ausdruck muss auf allen Pfaden verfügbar sein
- Lebendige Variablen:
  - lebendig in Knoten  $D$ , wenn es einen Pfad von  $D$  zum Exit-Knoten gibt, auf der die Variable ohne vorherige Redefinition schwach benutzt wird; sonst tot
  - Optimierungen:
    - \* Tote Variablen müssen nicht ausgerechnet werden
    - \* Weniger Registerdruck
    - \* Elimination redundanter Schleifenlaufvariablen nach der Schleife
  - Umsetzung:
    - \* Eine Bitposition pro Variable
    - \* Setzen bei Verwendung der Variable
    - \* Zurücksetzen bei starker Redefinition
    - \* Anfangsbelegung: false
    - \* Vorverarbeitung: oder
- Erreichbare Nutzungen:
  - Alle Nutzungen einer Variable durch Bestimmen der Pfade, auf denen Variable ohne vorherige Redefinition verwendet wird
  - Zweck: Hilfreich für Lebendigkeitsspannen und Registerallokation
- Vorhersehbare Ausdrücke:
  - auf allen Pfaden zum Exit-Knoten wird Ausdruck bestimmt und verwendete Variablen werden nicht vorher schwach redefiniert
  - Optimierungen:
    - \* Vorziehen der Ausdrucksberechnung zur Verkleinerung der Code-Größe
    - \* Geringerer Registerdruck in folgenden Zweigen
  - Umsetzung:
    - \* Eine Bitposition pro (Teil-)Ausdruck
    - \* Setzen bei Auswertung des Audrucks
    - \* Zurücksetzen bei mindestens schwacher Redefinition einer der verwendeten Variablen
    - \* Anfangsbelegung: false
    - \* Vorverarbeitung: und

### 3 Wertnummerierung in Grundblock

- Post-Order-Traversierung des Ausdrucksbaums
- Jede referenzierte Variable bekommt eindeutige Id
- Kombination aus Operator und beiden Argumenten bekommt entweder frische Id oder bei Duplikat (auch kommutativ) bereits verwendete
- Gleiche Ids  $\Leftrightarrow$  gleicher Ausdruck

## 4 Static Single Assignment (SSA)

- Jede Variable hat exakt eine starke und keine schwache Definition
- erfordert Verwendung von  $\phi$ -Funktionen, die verschiedene Variablen zusammenführt

### 4.1 Konstruktion der SSA-Form

- In isoliertem Grundblock trivial: Erzeugung einer neuen Variable  $x_i$  bei jeder Definition
- $\phi$ -Funktionen sollten möglichst sparsam nur wenn notwendig eingefügt werden

#### 4.1.1 Iteratives Dominanzgrenzenverfahren nach Cytron ( $\mathcal{O}(N^2)$ )

- Dominanzgrenzenkriterium: Aus Definition von  $v$  in Grundblock  $X$  folgt eine  $\phi$ -Funktion für  $v$  in jedem Grundblock der Dominanzgrenze von  $X$
- Eingefügte  $\phi$ -Funktionen werden Variablen zugewiesen, für die das Dominanzgrenzenkriterium erneut angewendet werden muss

#### 4.1.2 Wertnummerierungsverfahren nach Click

- Wertnummerierung für jeden Grundblock  $Z$  durchführen:
  - $\text{wn}_Z(\text{const}) = \text{const}$
  - Wenn  $Z$  Entry-Knoten, alle Wertnummern der Tupel in  $Z$  auf ungültig setzen
  - Bei zwei Vorgängern  $X, Y$  von  $Z$ , welche beide bereits besucht sind:
    - \* Wenn  $\text{wn}_X(t) \neq \text{wn}_Y(t)$ , dann neue Wertnummer  $x$  für  $\text{wn}_Z(t)$  einführen und  $\text{wn}_Z(t) = \phi(\text{wn}_X(t), \text{wn}_Y(t))$  generieren
    - \* Ansonsten:  $\text{wn}_Z(t) = \text{wn}_X(t)$
  - Bei einem Vorgänger  $X$  mit undefiniertem  $\text{wn}_X(t)$  diesen rekursiv initialisieren
  - Bei zwei Vorgängern  $X, Y$  von  $Z$ , wobei nur  $X$  noch unbesucht ist:
    - \* Neue Wertnummer für  $\text{wn}_Z(t)$  einführen
    - \* Neue besondere Wertnummer für  $\text{wn}_X(t)$  einführen
    - \* Vorläufige  $\phi'$ -Funktion  $\text{wn}_Z(t) = \phi'(\text{wn}_X(t), \text{wn}_Y(t))$  generieren
- Wertnummerierung für alle Tupel  $t$  in  $Z$  durchführen
- Nach Durchführung für alle Blöcke,  $\phi'$ -Funktionen korrigieren:
  - Besondere Wertnummern in  $X$  durch letzte in  $X$  gültige Wertnummer ersetzen
  - Offene  $\phi'$ -Funktionen durch abgeschlossene  $\phi$ -Funktion mit ersetzten Werten ersetzen
  - Falls  $t$  in unbesuchten Blöcken nicht geändert wurde,  $\text{wn}_Z(t) = \phi'(\text{wn}_X(t), \text{wn}_Y(t))$  löschen und alle Verwendungen von  $\text{wn}_Z(t)$  durch  $\text{wn}_X(t)$  ersetzen

### 4.2 Optimierungen auf SSA-Form

- Konstantenweitergabe:
  - Aus  $v_i = c$  können alle Verwendungen von  $v_i$  durch die Konstante  $c$  ersetzt werden
  - Aus  $v_i = \phi(c_1, c_2, \dots)$  kann  $v_i$  durch  $c_1$  ersetzt werden, wenn alle  $c_i$  gleich sind
  - Datenflussanalyse muss nach Ersetzung nicht wiederholt werden, Worklist-Algorithmus reicht aus

- Kopienfortschreibung: Aus  $v_i = y_j$  kann jede Verwendung von  $v_i$  durch  $y_j$  ersetzt werden
- Lebendigkeit einer Variable lässt sich direkt aus weiterem lesenden Zugriff erkennen
- Gemeinsame Teilausdrücke, verfügbare Ausdrücke und vorhersehbare Ausdrücke lassen sich durch Wertnummerierung in gesamter Prozedur leicht erkennen
- Elimination toten Codes:
  - alles, was nicht als lebendig markiert ist, ist tot
  - Markierung von Anweisungen als lebendig, falls sie
    - \* Seiteneffekte wie I/O, Funktionsaufrufe oder die Rückgabe von Werten hat
    - \* eine später von einer lebendigen Anweisung benutzte Variable schreibt
    - \* eine Verzweigung ist, von der eine lebendige Anweisung kontrollflussabhängig ist

### 4.3 Rücktransformation aus SSA-Form

- $\phi$ -Funktion nicht in Hardware abbildbar, Umwandlung notwendig
- Aus  $v_3 = \phi(v_1, v_2)$  wird
  - $v_3 = v_i$  in Vorgängerblock  $i$  falls sich Lebensspannen der  $v_i$  nicht paarweise überlappen (konventionelle SSA-Form)
  - $v_3 = t$  ( $t$  frische Temporärvariable) in Block mit  $\phi$ -Funktion sowie  $t = v_i$  in Vorgängerblock  $i$
- Sequentialisierung paralleler Kopieroperationen (TODO VL-06)
- Variablen können auch nur schwach definiert sein, etwa durch Zeiger auf Variablen oder Seiteneffekte von Prozeduren auf globale Variablen  $\rightarrow$  Hilfsfunktion  $\text{isAlias}(p, v)$ 
  - $$\text{isAlias}(p, v) = \begin{cases} *p & \text{falls } p = \&v \\ v & \text{sonst} \end{cases}$$
  - Für jede Zuweisung  $*p = \dots$  für jede Variable  $v_i$ , auf die  $p$  zeigen könnte, neue Zuweisung  $v_{i+1} = \text{isAlias}(p, v_i)$  einfügen

## 5 Alias-Analyse

- Aliase: Verschiedene Möglichkeiten auf gleiche Speicherstelle zuzugreifen
- Sprachabhängige Quellen von Aliasen:
  - dynamisch allokierte Datenstrukturen
  - Zeiger
  - überlappende Speicherbereiche
  - Referenzen auf Arrays, -Abschnitte, -Dimensionen und -Elemente
  - Prozedurparameter
  - Zeigerarithmetik
- Sprachunabhängige Quellen wie Transitivität ( $a$  zeigt auf  $b$  und  $b$  zeigt auf  $c \rightarrow c$  ist von  $a$  erreichbar)



- TODO: Aliase in Fortran, Pascal, C und Java (VL-07)
- mögliche Aliase (may) müssen nur in einem Pfad Aliase sein
- sichere Aliase (must) müssen in allen Pfaden Aliase sein
- Fluss-ignorierende Analyse bestimmt, ob Namen irgendwo im KFG Aliase sein können:
  - vergleichsweise einfach berechenbar
  - oft für interprozedurale Analyse eingesetzt
  - Relation  $\text{Alias}(a, b)$  gibt an, dass  $a$  und  $b$  für ganze Prozedur Aliase sind, ist reflexiv, symmetrisch und bei must-Analyse auch transitiv
- Fluss-sensitive Analyse bestimmt, ob Namen in einem Basisblock Aliase sein können:
  - aufwändige Berechnung
  - oft für intraprozedurale Analyse eingesetzt
  - Funktion  $\text{Alias}(P, v) = S$  gibt an, dass  $v$  an Programmpunkt  $P$  auf Speicherstelle  $S$  zeigen kann
  - Bei may-Analyse:
    - \*  $\text{Alias}(P, v_1) \cap \text{Alias}(P, v_2) \neq \emptyset \Leftrightarrow v_1, v_2$  sind in  $P$  Aliase
    - \* nicht transitiv
  - Bei must-Analyse:
    - \*  $|\text{Alias}(P, v)| = 1$ , Aliase wenn Speicherstelle gleich
    - \* transitiv
- intraprozedurale Analyse:
  - Betrachtung einer einzelnen Prozedur
  - Worst-Case-Annahmen bei Aufruf anderer Prozedur
  - häufig auf Datenflussproblem zurückgeführt
- interprozedurale Analyse:
  - Standard-Verfahren
  - Andersens Algorithmus
  - Steensgards Algorithmus

## 5.1 DFA-Ansatz zur intraprozeduralen, fluss-sensitiven may-Analyse nach Muchnick

- Vorwärtsproblem mit minimalen Grundblöcken
- Sprachabhängige lokale Flussfunktionen für unmittelbare Auswirkungen pro Instruktion (Sammler/Gatherer):
  - TODO (VL-07)
- Sprachunabhängiger Fixpunktpunktalgorithmus (Verbreiter/Propagator):
  - TODO (VL-07)

## 5.2 Standard-Verfahren zur interprozeduralen, fluss-insensitiven may-Analyse

- Laufzeit:  $\mathcal{O}(n^2 + n \cdot e)$
  - Alias-Quellen in Sprachen ohne &-Operator:
    - Übergabe globaler Variable an Funktion
    - Übergabe der gleichen Variable an mehrere formale Parameter einer Funktion
    - Zugriff auf umgebende Variablen/formale Parameter bei geschachtelten Prozeduren (vgl. globale Variablen)
  - Prozeduraufwurfgraph (PCG):
    - Knoten: Prozeduren des Programms
    - Gerichtete Kante von  $p$  nach  $q$  wenn  $p$   $q$  aufrufen kann
  - Variablenbindungsgraph:
    - Knoten: formale Parameter
    - Gerichtete Kante von  $x$  nach  $y$  wenn  $(p, q)$  in PCG,  $x$  formaler Parameter von  $p$  und  $x$  Argument für formalen Parameter  $y$  von  $q$
    - transitiver Abschluss liefert Alias-Menge der formalen Parameter (aber globale Variablen sind nicht berücksichtigt)
  - Paarbindungsgraph:
    - Knoten: alle Paare formaler Parameter
    - Gerichtete Kante von  $(x, y)$  nach  $(X, Y)$ , wenn
      - \*  $(x, y)$  bei einem Anruf an  $(X, Y)$  gebunden werden oder
      - \*  $(x, y)$  bei einem Aufruf im Inneren einer geschachtelten Prozedur an  $(X, Y)$  gebunden werden, wobei ggf. Aliase von  $x$  oder  $y$  verwendet werden
1. Aliase globaler Variablen:
    - a) Bindungen von globalen Variablen an formalen Parametern entdecken
    - b) Aus Pfaden in Variablenbindungsgraph zusätzliche formale Parameter, an die globale Variable weitergegeben werden können, bestimmen (in topologischer Reihenfolge durchlaufen; bei Zyklen bis Fixpunkt erreicht)
    - c) Aliase der globalen Variablen ablesen
  2. Parameterpaare als Aliase: Zwei formale Parameter sind Aliase, wenn
    - die gleiche Variable an beide übergeben wird
    - eine globale Variable und einer ihrer Aliase übergeben wird
    - a) Betrachtung aller Paare formaler Parameter
    - b) Markierung der Paare, die beim Aufruf zu Alias-Paaren werden können
    - c) Propagation von Alias-Informationen durch den PCG
    - d) Ablesen der Aliase formaler Parameter

## 5.3 Steensgards Algorithmus zur interprozeduralen, fluss-insensitiven may-Analyse

- Speichergraph:
    - Knoten: eine oder mehrere Speicherstellen
    - gerichtete Kante: „zeigt (möglicherweise) auf“-Beziehung, damit Alias: (\*start, ziel)
  - approximiert Speichergraph:
    - Maximal eine abgehende Kante pro Knoten
    - Zusammenfassung mehrerer Speicherstellen, auf die Variable zeigen kann, zu einer Speicherstelle
1. Einführung eines Knotens im Speichergraph pro Variable, wobei Kanten Zeiger repräsentieren
  2. Fluss-ignorierendes, lineares Abarbeiten aller Anweisungen:
    - $a = \&b$ : Kante zwischen  $a$  und  $b$  einfügen; sofern bereits Kante zwischen  $a$  und  $x$  existiert, rekursives Verschmelzen von  $x$  und  $b$
    - $a = b$ :
      - $b$  ist Zeiger: Verschmelzen der Ziele von  $a$  und  $b$ , anschließend zeigen beide auf diesen Knoten
      - $b$  ist kein Zeiger bzw. noch nicht erkannt: Annotiere  $b$  mit  $(a : b)$  (Falls später Zeigerziel  $y$  von  $b$  entdeckt wird, muss Kante von  $a$  nach  $y$  ergänzt werden)
    - $a = *b$ :
      - $b, *b$  haben bereits ausgehende Kanten: Kante zwischen  $a$  und  $**b$  ergänzen
      - sonst: analog zu  $a = b$  mit adäquaten Annotationen (TODO: Übung)
    - $*a = b$ :
      - $a, *a$  mit ausgehenden Kanten: rekursives Verschmelzen von  $*b$  und  $**a$
      - sonst: TODO (Übung)
    - $a = b \oplus c$  mit  $\oplus$  binärer Operation:
      - $a, b, c$  keine Zeiger: Annotation von  $b$  und  $c$  mit  $(a : b)$  bzw.  $(a : c)$
      - $b$  oder  $c$  Zeiger: Kante von  $a$  nach  $*b$  hinzufügen,  $c$  mit  $(a : c)$  annotieren
    - $x = p(y_1, \dots, y_n)$ :
      - ggf. Speichergraph für  $p$  berechnen
      - Zuweisungsregeln für  $x$  und alle  $y_i$  verwenden
    - Funktionszeiger: Bei Verschmelzen auf Typen achten

## 6 Schleifen

### 6.1 Schleifeninvariante Anweisungen

- schleifeninvariante Instruktion: alle Operanden  $v$  sind
  - konstant oder
  - nur außerhalb der Schleife berechnet oder
  - innerhalb der Schleife in *einer* schleifeninvarianten Instruktion berechnet
- Auffinden schleifeninvarianter Instruktionen:

1. Identifikation natürlicher Schleifen
  2. Berechnung der erreichenden Definitionen
  3. Konstantenfaltung
  4. Markierung aller konstanten Ausdrücke und Variablen, sowie alle Markierung, deren erreichende Definitionen außerhalb der Schleife liegen, als schleifeninvariant
  5. Solange neue Markierungen auftreten: alle Instruktionen/Variablen als schleifeninvariant markieren, wenn
    - sie konstant sind oder
    - wenn alle erreichenden Definitionen außerhalb der Schleife liegen oder
    - wenn genau eine erreichende Definition existiert und diese als schleifeninvariant markiert ist
- Herausziehen von schleifeninvarianten (Teil-)Ausdrücken vor den Schleifenkopf unter Beachtung von Seiteneffekten/Fehlern immer möglich
  - Auswahl von herausziehbaren Zuweisungen:
    1. Berechnung der Dominanz, erreichenden Definitionen und Anwendung von Konstantenfaltung
    2. Berechnung der invarianten Instruktionen
    3. Bestimmung der Schleifenausgänge, also Knoten, die KFG-Nachfolger außerhalb der natürlichen Schleife besitzen
    4. Alle Anforderungen an Zuweisungen müssen für Herausziehbarkeit gelten:
      - schleifeninvariant
      - in Grundblock, der alle Schleifenausgänge dominiert und alle die Variable nutzenden Blöcke strikt dominiert
      - einzige Zuweisung zu Variable in der ganzen Schleife
    5. Herausziehen vor den Schleifenkopf in Breitensuchreihenfolge zur Bewahrung der Datenabhängigkeiten zwischen herausgezogenen Zuweisungen
  - Sonderfall while-Schleife: Schleifenausgang wird von keinem anderen Grundblock in der Schleife dominiert, also keine herausziehbaren Zuweisungen in der Schleife → erst Umbau zu do-while-Schleife
  - Optimierung bei invariantem Test: Generierung einer eigenen Schleife für jeden Fall

## 6.2 Induktionsvariablen

- Induktionsvariable: Variable, die bei jeder Schleifeniteration um konstanten Wert inkrementiert oder dekrementiert wird
- Einfache Induktionsvariable: Variable  $x$ , die nur durch  $x = x \pm c$  mit schleifeninvariantem  $c$  verändert wird
- Abhängige Induktionsvariable: Variable  $y$ , die in der Schleife durch  $y = a \cdot x \pm b$  mit schleifeninvarianten  $a$  und  $b$  als lineare Funktion aus Induktionsvariable  $x$  berechnet wird
- Berechnung der Familie einer einfachen Induktionsvariable  $x$ : Menge der Induktionsvariablen  $y$ , die mit linearer Funktion  $y = c_1 \cdot x \pm c_2$  von  $x$  mit schleifeninvarianten  $c_1$  und  $c_2$  abhängen:

- Zuweisung der Form  $y = c_1 \cdot x \pm c_2$
- Zuweisung der Form  $y = c_1 \cdot z \pm c_2$ , wobei  $z \in \text{Familie}(x)$  und
  - \* keine Veränderung von  $x$  zwischen Zuweisung zu  $z$  und Zuweisung zu  $y$
  - \* keine von außerhalb der Schleife erreichende Definition von  $z$  erreicht Zuweisung zu  $y$
- Berechnungen der Tripel  $(x, c_1, c_2)$  für alle Induktionsvariablen  $y$  einer Schleife:
  1. Berechnung von Schleifeninvarianz und erreichenden Definitionen
  2. Bestimmung aller einfachen Induktionsvariablen durch Code-Inspektion („Pattern Matching“)
  3. Bestimmung aller Variablen  $y$  mit einziger (oder identischen) Zuweisung der Form  $y = x \pm c$  oder  $y = c \cdot x$  mit  $x$  Induktionsvariable und  $c$  schleifeninvariant:
    - $x$  ist einfache Induktionsvariable:

$$\text{Tripel}(y) = \begin{cases} (x, 1, c) & \text{falls } y = x + c \text{ oder } y = c + x \\ (x, 1, -c) & \text{falls } y = x - c \\ (x, c, 0) & \text{sonst } (y = x \cdot c \text{ oder } y = c \cdot x) \end{cases}$$

- $x$  ist abhängige Induktionsvariable mit Tupel  $(u, d_1, d_2)$ :

$$\text{Tripel}(y) = \begin{cases} (u, d_1, d_2 + c) & \text{falls } y = x + c \text{ oder } y = c + x \\ (u, d_1, d_2 - c) & \text{falls } y = x - c \\ (u, d_1 \cdot c, d_2 \cdot c_2) & \text{sonst } (y = c \cdot x \text{ oder } y = x \cdot c) \end{cases}$$

- Vereinfachung der Berechnungsvorschrift von Schleifenlaufvariablen pro Iteration (Reduktion der Ausdrucksstärke, „Strength Reduction“) für jede einfache Induktionsvariable  $x$ :
  1. Für jede von  $x$  abhängige Induktionsvariable  $y$  mit Tripel  $(x, a, b)$ :
    - a) Einführung neuer Variable  $y'$
    - b) Ersetzung der Zuweisung zu  $y$  durch  $y = y'$
    - c) Einfügen der Zuweisung  $y' = y' \pm (a \cdot c)$  nach jeder Zuweisung  $x = x \pm c$
    - d) Einfügen der Zuweisung  $y' = x \cdot a + b$  vor dem Schleifenkopf
- Modifikation von bedingten Sprüngen ggf. möglich
- Verringerung der Anzahl von Array-Bereichstests ggf. möglich

### 6.3 Datenabhängigkeiten in Schleifen und Arrays

- Schleifenunabhängige Datenabhängigkeiten: existieren auch ohne Schleife (etwa wenn im Rumpf)
- Schleifengetragene Datenabhängigkeiten: resultieren ausschließlich aus Schleife (etwa zwischen verschiedenen Iterationen)
- Iterationsvektor  $i = (i_1, i_2, \dots, i_d)$  für  $d$  geschachtelte Schleifen
  - eindeutig für jede Iteration
  - lexikographische Ordnung auf Iterationsvektoren
  - $i \angle j \Leftrightarrow$  Iteration  $i$  zur Laufzeit vor Iteration  $j$

- Iterationsraum: Menge aller möglichen Iterationsvektoren
- Abhängigkeitsdistanz (bei geschachtelten Schleifen Vektor)  $d$ : Bei Datenabhängigkeit zwischen Iteration  $i_{\text{source}}$  und  $j_{\text{target}}$  und  $i_{\text{source}} < j_{\text{target}}$ , dann  $i_{\text{source}} + d = j_{\text{target}}$
- Abhängigkeitsrichtung: Vergrößerung der Abhängigkeitsdistanz mit folgenden Ersetzungen an jeder Position des Vektors:

$$r_i = \begin{cases} = & \text{falls } 0 = d_i \\ < & \text{falls } 0 < d_i \\ > & \text{falls } 0 > d_i \end{cases}$$

- Richtungsvektor: Zusammenfassung aller Abhängigkeitsrichtungen einer Schleife:
  - „übliche“ Zusammenfassungen (z.B. = und < zu  $\leq$ )
  - keine Aussage \* bei < und >
- Die Abhängigkeit tragende Schleife  $p$  folgt aus der Struktur der Abhängigkeitsdistanzen bei Inkrementeschleifen:

$$\underbrace{(d_1, d_2, \dots, d_p)}_0, \underbrace{\dots}_{>0}, \underbrace{\dots, d_n}_{\text{beliebig}}$$

- Parallele Ausführbarkeit aller Iterationen einer Schleife  $p$  gegeben, wenn
  - diese keine Abhängigkeit trägt (alle Abhängigkeitsdistanzen haben  $d_p = 0$ ) oder
  - jede schleifengetragene Abhängigkeit von einer  $p$  umgebenden Schleife getragen wird (für jede Abhängigkeitsdistanz  $\exists q < p : d_q > 0$ )
- Parallele Ausführung nicht immer gewinnbringend, da auf Cache-Ebene Abhängigkeiten auftreten können („False Sharing“)
- Optimierungsmöglichkeiten:
  - Skalare Ersetzung von Array-Ausdrücken („register pipelining“), wenn:
    - \* Abhängigkeit wird von innerster Schleife getragen
    - \* Konstante Abhängigkeitsdistanz  $d$
    - \*  $d + 1$  freie Register
    1. Vor Schleife Initialisierung der Register und „Abschälen“ der ersten  $d$  Iterationen
    2. In Schleife Benutzung der Register bei Lesezugriff und Speicherung der Registerinhalte bei Speicherzugriff, sowie am Anfang des Rumpfs „Aufrücken“ der Register
  - Skalar-Vervielfachung: Einführung eines Arrays zur Elimination von unnötigen Abhängigkeiten einer Temporärvariable
  - Verbesserung der zeitlichen Lokalität durch optimierte Registerverwendung
  - Verbesserung der räumlichen Lokalität durch optimierte Cache-Verwendung
  - Parallelisierung/Vektorisierung von Schleifen
  - Daten-/Prozessplatzierung zur Kommunikationsoptimierung

## 6.4 Indexanalyse

- Bestimmung, ob 2 Array-Zugriffe selbe bzw. unterschiedliche Elemente ansprechen: Grundannahme pessimistisch (gleiches Element)
- Für 2 Array-Zugriffe  $S1 : A[f_1(i_1, \dots, i_d), \dots, f_m(i_1, \dots, i_d)]$  und  $S2 : A[f_1(i_1, \dots, i_d), \dots, f_m(i_1, \dots, i_d)]$  gilt  $S1 \delta S2 \Leftrightarrow$ :
  - mindestens ein Schreibzugriff
  - $\exists I, J : I = (i_1, \dots, i_d) \angle J = (j_1, \dots, j_d)$  mit  $I, J$  innerhalb der Schleifengrenzen (Ungleichungssystem mit Nebenbedingungen)
  - $\forall p : f_p(I) = g_p(J)$  (Gleichungssystem mit Schleifenlaufvariablen als Variablen und Konstanten aus linearem Index-Ausdruck als Koeffizienten)
- Aus Laufzeitgründen lediglich Tests, sonst pessimistische Grundannahme:
  - GGT-Test:
    1. Gleichsetzen der beiden Array-Ausdrücke
    2. Konstante auf eine Seite umformen
    3. Abhängigkeit nur dann möglich, wenn der größte gemeinsame Teiler der Koeffizienten die Konstante teilt
    4. Für verschiedene Arten der Abhängigkeit in obige Kriterien einsetzen
  - Ungleichungstest:
    1. Ungleichungen für verwendete Variablen aufstellen
    2. Gleichung für Zugriff aufstellen und so umstellen, dass 0 auf einer Seite steht
    3. In Gleichung jeweils untere und obere Grenzwerte einsetzen und damit ein Intervall bestimmen
    4. Sofern Intervall nicht 0 enthält, keine Abhängigkeit
  - Fourier-Motzkin-Test:
    - \* Ungleichungssystem in kanonische Form überführen:
      - $\beta < b \cdot z$  untere Schranke für  $z$  mit  $\beta > 0$
      - $a \cdot z < \alpha$  obere Schranke für  $z$  mit  $\alpha > 0$
    - \* Mit Fourier-Motzkin-Elimination Variable  $z$  eliminieren:
$$a \cdot \beta \leq a \cdot z \cdot b \leq b \cdot \alpha \rightarrow a \cdot \beta \leq b \cdot \alpha$$
    - \* Kleineres Problem  $a \cdot \beta \leq b \cdot \alpha$  rekursiv lösen:
      - keine Lösung  $\rightarrow$  unabhängig
      - Lösung existiert  $\rightarrow$  Prüfung, ob auch ganzzahlige Lösung für  $a \cdot z \cdot b$  existiert

## 6.5 Entfernung von schleifengetragenen Datenabhängigkeiten

- Legalität: Für jede Datenabhängigkeit muss die relative Reihenfolge auch nach Anwendung der Transformation bzw. Restrukturierung erhalten bleiben, die entstehenden Abhängigkeitsdistanzvektoren nicht lexikographisch sein

### 6.5.1 Schleifentransformationen

- Erhalten Durchlaufreihenfolge
- Neuausrichtung:
  - Verschiebung der Ausführung einer Anweisung um  $d$  Iterationen (wobei  $d$  konstante Abhängigkeitsdistanz ist)
  - Bedingte Anweisungen für die ersten und letzten  $d$  Anweisungen zur Erhaltung der Semantik
- Ausrollen um Faktor  $f$ :
  - Pro Iteration der neuen Schleife gleich  $f$  Iterationen der alten Schleife ausführen
  - Reduktion des Schleifenoverheads
  - Vergrößerung des Codes
- Bereichsteilen:
  - Iterationsbereich wird auf zwei Schleifen mit jeweils gleichem Rumpf aufgeteilt
  - ermöglicht weitere Optimierungen (etwa mit DFA oder Vektorisierung)
  - Vergrößerung des Codes
- Schälen (Spezialfall des Bereichsteilens):
  - erste (oder letzte) Iteration der Schleife werden herausgezogen um spezielle Datenabhängigkeiten zu eliminieren
  - ermöglicht oft Parallelisierung der Restschleife
  - ergänzt Neuausrichtung
- Produktschleifenbildung:
  - Kombination geschachtelter Schleifen zu einer Schleife durch Multiplikation der Bereiche
  - Produktgröße oft besser geeignet für Parallelisierung/Vektorisierung
  - fügt ggf. abhängige Induktionsvariablen ein
  - verhindert möglicherweise andere Optimierungen
- Streifenschneiden:
  - Aufteilen des Schleifenbereichs in mehrere Bereiche (Blöcke, Streifen) bestimmter Größe
  - verbessert Parallelisierung/Vektorisierung der Schleife
  - oft nach Produktschleifenbildung
  - erhöht Schleifenoverhead durch zusätzliche Abhängigkeitsdimension

### 6.5.2 Schleifenrestrukturierungen

- Ändern Durchlaufreihenfolge
- Verschmelzung:
  - Zusammenfassen der Rümpfe mehrerer Schleifen
  - Voraussetzungen:
    - \* beide Schleifen haben gleichen Indexbereich



- \* Namensgleichheit beider Laufvariablen kann durch Umbenennung erreicht werden
- \* Bei keiner Skalarnutzung in zweiter Schleife darf es erreichende Definition aus anderer Schleife geben
- Legalität: Es darf keine Abhängigkeit zwischen einer Instruktion aus der zweiten Schleife mit einer aus der ersten in einer späteren Iteration entstehen
- Reduktion des Schleifenoverheads
- größere Grundblöcke erlauben effizientere lokale Optimierungen
- ggf. verbesserte zeitliche Lokalität
- Rumpfteilen:
  - Aufteilen des Rumpfs einer Schleife auf mehrere Schleifen
  - Gegenteil der Verschmelzung
  - Legalität:
    - \* Anweisungen, die in starken Zusammenhangskomponenten des Abhängigkeitsgraphs der zu teilenden Schleife liegen, dürfen nicht auf verschiedene Schleifen verteilt werden
    - \* Reihenfolge der Schleifen muss Abhängigkeiten vor dem Teilen widerspiegeln
  - kleinere Grundblöcke verringern ggf. Registerdruck
  - kleinere Rumpfe passen besser in den Instruktionscache
  - ggf. kann eine der Schleifen parallelisiert/vektoriert werden
- Kachelschneiden

### **Lineare Schleifenrestrukturierungen**

- Schleifenvertauschung
- Richtungsumkehr
- Schleifenneigen