Optimierungen in Übersetzern: Verfahren

Marco Ammon (my04mivo)

11. August 2020

Inhaltsverzeichnis

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

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 SemDom[w] in Reihenfolge fallender DFS-Nummern:
  - Direkte Vorgänger auf T sind Kandidaten
  - $\min_{u\in \mathrm{Pred}(w)} \mathrm{SemDom}[u]$ ist Kandidat
  - Minimum der Kandidaten ist SemDom[w]

- Berechnung von ImmDom[w] durch Durchlaufen in Tiefenordnung von SemDom[w] nach w:
  - Jeweils alle Vorgänger u untersuchen und u mit kleinstem SemDom[u] finden
  - $$\text{ImmDom}[w] = \begin{cases} \text{SemDom}[u] & \text{falls SemDom}[w] = \text{SemDom}[u] \\ \text{ImmDom}[u] & \text{sonst} \end{cases}$$

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 $d \geq\geq n$
- Natürliche Schleife:
  - Rückwärtskante (n, d) sowie alle Knoten k mit $d \geq\geq k$ 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:
  - Schreiben vor Lesen:
  - Schreiben vor Schreiben: Ausgabeabhängigkeit
  - Lesen vor Schreiben: Anti-Abhängigkeit
- 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: optimal Besuch in Reihenfolge des spannenden Tiefenbaums

```
in(Entry) ← Init
for all n ∈ N \ {Entry} do
    in(n) ← ⊥
end for
WL ← N \ {Entry}
while WL ≠ ∅ do
    B ← pop(WL)
    out ← f_B(in(B))
    for all B' ∈ Succ(B) do
        in(B') ← in(B') ⊔ out
        if out ≠ out(B) then
            WL ← WL ∪ {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 → Variable nicht initialisiert
    * Genau eine oder gleiche Definition → Konstantenweitergabe oder Kopienfortschreibung möglich
    * Alle Variablen nur außerhalb einer Schleife definiert → 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 Wergleichtheitsbeziehung
  - 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 redudanter 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 ϕ-Funktionen, die verschiedene Variablen zusammenführt

4.1 Konstruktion der SSA-Form

- In isoliertem Grundblock trivial: Erzeugung einer neuen Variable x_i bei jeder Definition
- ϕ-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 ϕ-Funktion für v in jedem Grundblock der Dominanzgrenze von X
- Eingefügte ϕ-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 ϕ'-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, ϕ'-Funktionen korrigieren:
  - Besondere Wertnummern in X durch letzte in X gültige Wertnummer ersetzen
  - Offene ϕ'-Funktionen durch abgeschlossene ϕ-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

- ϕ-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 ϕ-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 isAlias(p, v)
  - $$\text{isAlias}(p, v) = \begin{cases} *p & \text{falls } p = \&v \\ v & \text{sonst} \end{cases}$$
  - Für jede Zuweisung $*p =$ 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 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 Alias$(P, v) = S$ gibt an, dass v an Programmpunkt P auf Speicherstelle S zeigen kann
  - Bei may-Analyse:
    - Alias$(P, v_1) \cap$ 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 mit $\mathcal{O}(n^2 + n \cdot e)$

- 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)
- Prozeduraufrufgraph (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 für interprozedurale, fluss-insensitive 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
   - $x = p(y_1, \ldots, y_n)$:
     - ggf. Speichergraph für p berechnen
     - Zuweisungsregeln für x und alle y_i verwenden
   - Funktionszeiger: Bei Verschmelzen auf Typen achten

6 Induktionsvarianten und schleifeninvarianter Code

7 Schleifen und Arrays

8 Schleifentransformationen

9 Schleifenrestrukturierungen