Compilerkonstruktion

Wintersemester 2015/16

Prof. Dr. R. Parchmann

15. Dezember 2015

Lokale Optimierungen

In diesem Abschnitt werden einfache Blöcke isoliert betrachtet. Der Drei-Adress-Code in den Blöcken wird optimiert, dabei werden gemeinsame Teilausdrücke im Block entfernt und einfache algebraische Umformungen durchgeführt.

Ausgehend von einem einfachen Block wird ein gerichteter azyklischer Graph (directed acyclic graph - DAG) aufgebaut, der die Berechnungen im Block darstellt. Dieser DAG kann dann als Eingabe für eine Maschinencode- Erzeugung verwendet werden oder aber in einen Block zurücktransformiert werden. Bei der Konstruktion werden gemeinsame Teilausdrücke erkannt und berücksichtigt.

Konstruktion eines DAGs für einen einfachen Block

Die Knoten des DAGs enthalten die folgende Information:

- interne Knoten enthalten einen Operator als Markierung
- Blätter enthalten einen Variablennamen oder eine Konstante als Markierung
- den Knoten ist eine Liste angehängt, in der Null oder mehr Variablennamen auftreten.

Jeder Knoten des DAGs korrespondiert mit einem berechneten Zwischenergebnis und die in der Liste aufgeführten Variablen „enthalten“ diesen Wert.

Man benötigt weiterhin eine Tabelle, in der ein Verweis auf den jeweils „neuesten“ Knoten zu finden ist, der den Wert einer Variablen bzw. einer Konstanten repräsentiert.
`node`(*identifier*) gibt jeweils diesen Knoten zurück.

Algorithmus

Durchlaufe den einfachen Block von Anfang bis Ende.
Für jeden Drei-Adress-Befehl der Form `x := y op z` führe die folgenden Schritte aus:

1. Befindet sich `y` nicht in der Tabelle, so erzeuge einen neuen Knoten mit Markierung `y` und vermerke dies in der Tabelle. (d.h. `node(y)` gibt diesen Knoten zurück!)
   Befindet sich `z` nicht in der Tabelle, so führe dieselben Schritte für `z` aus.
2. Bestimme, ob es im DAG einen Knoten α gibt, der als linken Nachfolger `node(y)`, als rechten Nachfolger `node(z)` und als Markierung `op` hat. Existiert ein derartiger Knoten nicht, so erzeuge einen Knoten α mit dieser Eigenschaft.
3. Ist die Variable `x` in der dem Knoten `node(x)` angehefteten Liste, so lösche `x` aus der Liste. Füge `x` in die Liste des Knotens α ein und ändere die Tabelle so, dass `node(x)` den Wert α hat.

Für jeden Drei-Adress-Befehl der Form `x := op y` führe die folgenden Schritte aus:

1. Befindet sich `y` nicht in der Tabelle, so erzeuge einen neuen Knoten mit Markierung `y` und vermerke dies in der Tabelle.
2. Bestimme, ob es im DAG einen Knoten α gibt, der als Nachfolger `node(y)` und als Markierung `op` hat. Existiert ein derartiger Knoten nicht, so erzeuge einen Knoten α mit dieser Eigenschaft.
3. Ist die Variable `x` in der dem Knoten `node(x)` angehefteten Liste, so lösche `x` aus der Liste. Füge `x` in die Liste des Knotens α ein und ändere die Tabelle so, dass `node(x)` den Wert α hat.

Für jeden Drei-Adress-Befehl der Form `x := y` führe die folgenden Schritte aus:

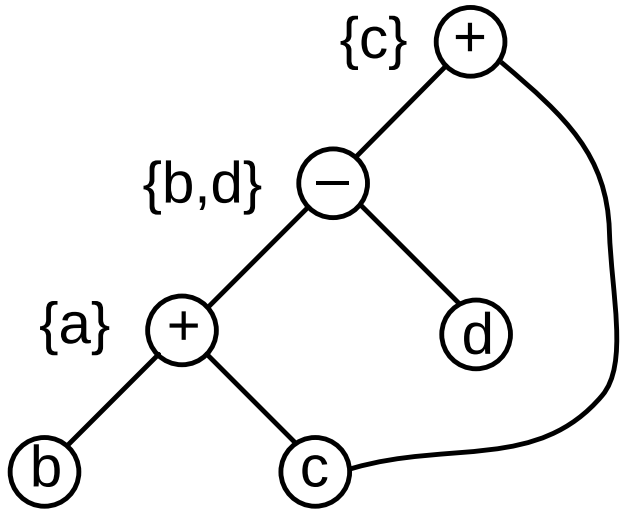
1. Befindet sich `y` nicht in der Tabelle, so erzeuge einen neuen Knoten mit Markierung `y` und vermerke dies in der Tabelle.
2. Sei $\alpha =$ `node(y)`.
3. Ist die Variable `x` in der dem Knoten `node(x)` angehefteten Liste, so lösche `x` aus der Liste. Füge `x` in die Liste des Knotens α ein und ändere die Tabelle so, dass `node(x)` den Wert α hat.

Beispiel

Für den einfachen Block

```
a := b + c
b := a - d
c := b + c
d := a - d
```

erhielte man den DAG



Ist die Variable `b` nicht lebendig am Ende des Blocks, könnte man den einfachen Block in

```
a := b + c
d := a - d
c := d + c
```

umschreiben.

Sind `b` und `d` am Ende lebendig, muss der Befehl `b := d` angefügt werden.

Darstellung von Feldzugriffen im DAG

Auf den ersten Blick sieht es so aus, als könne man Zugriffe auf Feldelemente wir die anderen Operatoren bearbeiten.

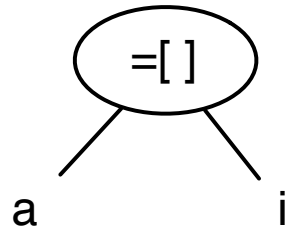
Beispiel

Betrachte die Drei-Adress-Befehle

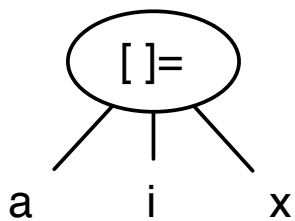
```
x := a[i]
a[j] := y
z := a[i]
```

Interpretiert man den Feldzugriff `a[i]` als eine Operation mit Operanden `a` und `i`, dann wäre `a[i]` im Beispiel ein gemeinsamer Teilausdruck. Allerdings könnten `i` und `j` den gleichen Wert haben und damit repräsentieren die beiden Auftreten von `a[i]` eventuell unterschiedliche Werte!

Man übersetzt einen Feldzugriff der Art `x := a[i]` in einen Knoten mit zwei Nachfolgern der Art



Eine Zuweisungen zu Feldelementen wie etwa `a[i] := x` wird durch einen Knoten mit einer dreifach-Verzweigung dargestellt:

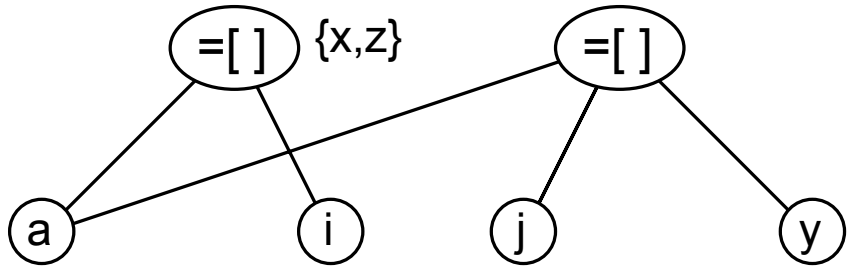


Beim Einsetzen eines Knotens dieser Art in den entstehenden DAG werden alle Knoten, deren Wert vom Knoten `node(a)` abhängen, eingefroren, da angenommen werden muss, dass *jedes* Element des Feldes verändert wurde.

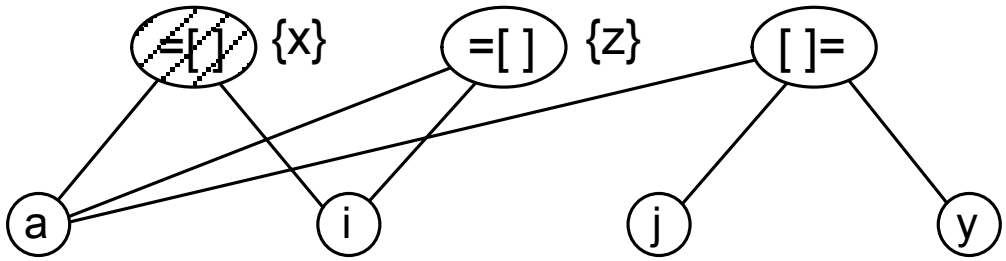
Dies bedeutet, dass ihre Listen nicht mehr verändert werden dürfen!

Eingefrorene Knoten werden in den Abbildungen schraffiert gezeichnet.

Ohne diese Regel würde man aus dem obigen Block den (falschen) DAG



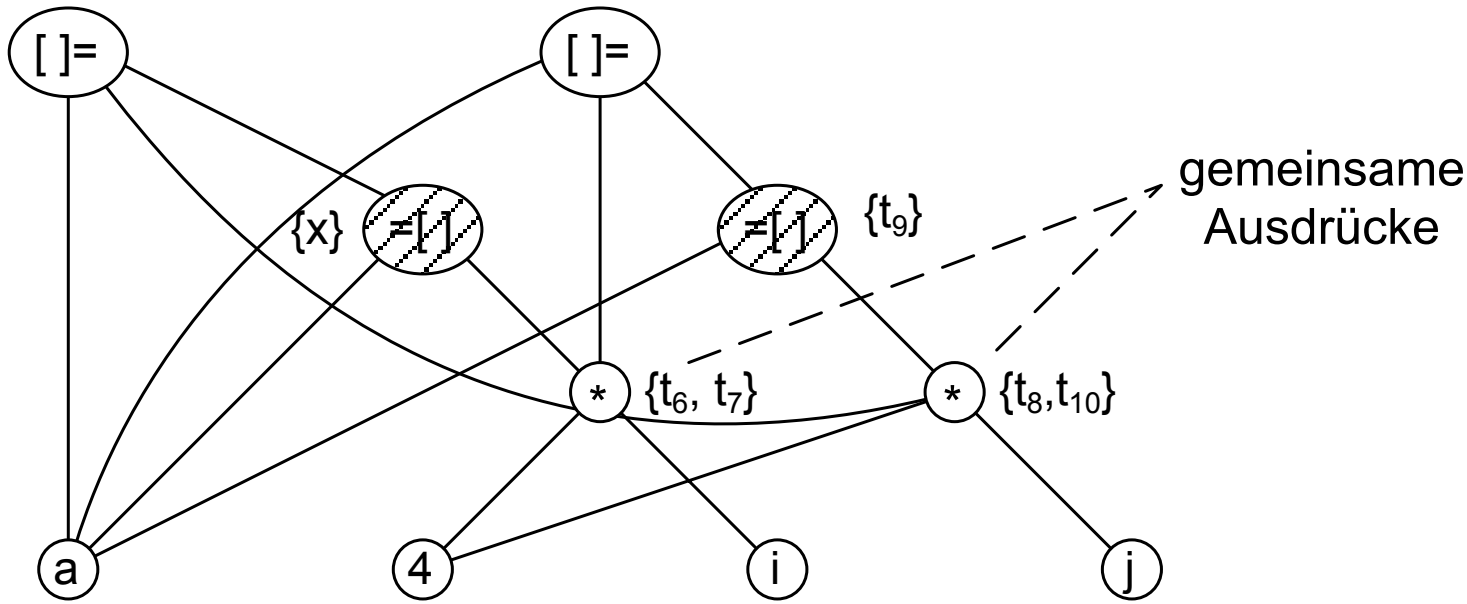
erhalten. Beachtet man jedoch die zusätzliche Regel, so erhält man:



Entsprechende Regeln müssen auch bei Zeiger-Operationen und bei Prozeduraufrufen beachtet werden. Im Zweifelsfall bleibt bei Sprachen, die etwa beliebige Zeigeroperationen ermöglichen nichts anderes über, als alles Vorherige einzufrieren!

Beispiel

Der ursprüngliche Block B_5 im Quicksort-Beispiel ergäbe den folgenden DAG:



Was kann man nun mit dem so gewonnenen DAG machen?

1. Erkennen, welche Variablen und Konstanten in einem einfachen Block benötigt werden (Blätter) und welchen Variablen neue Werte zugewiesen werden (use/def- Werte).
2. Rückwandlung in einen einfachen Block, wobei gemeinsame Teilausdrücke entfernt werden.
3. Maschinencode-Erzeugung direkt vom DAG aus.