Dominatorbasierte Optimierung
Erweiterung, Neuordnung und Reduzierung

Sascha Schlegel
Matrikelnummer: 7678991

Sommersemester 2014

Sascha Schlegel
Sophie-Scholl-Str. 22
41540 Dormagen

sascha-schlegel@web.de

Inhaltsverzeichnis

Vorwort

Diese Ausarbeitung entstand im Rahmen meiner Seminararbeit im Sommersemester 2014 an der FernUniversität Hagen. Betreut wurde das Seminar 01928 von apl. Prof. Dr. Robert Rettinger.

Das Thema des Seminars lautete „Code-Optimierung“, ein Teilbereich des Compilerbaus. Dabei orientierten sich die Themen aller Seminarteilnehmer - auch in ihrer Reihenfolge - an dem Buch Building an optimizing Compiler von Bob Morgan [Morgan(1998)].

In diesem Rahmen bearbeitete ich das Thema „Dominatorbasierte Optimierung - Erweiterung, Neuordnung und Reduzierung“, welches grob durch die Kapitel 8.4 bis 8.7 des genannten Buches umrissen wird.

Neben dieser Ausarbeitung erarbeitete ich einen Vortrag mit dem gleichen Thema. Die Präsenzphase vom 19. bis 20. Juli diente dabei der Vorstellung aller Präsentationen, deren Dauern etwa 60 Minuten zuzüglich Diskussionsrunde für jeden Seminarteilnehmer betrugen.

Kapitel 1

Einordnung

Zu Beginn der Ausarbeitung meiner Seminararbeit möchte ich klar machen, wo mein Thema anzusiedeln ist. Dabei gebe ich zunächst einen Überblick über die Gesamtthematik, bevor ich mich der Einordnung meines Themas in den Themenkomplex widme. Ohne zu sehr auf Details einzugehen, werden am Ende dieses Kapitels einige Begriffe erläutert und klargestellt. Dies beschränkt sich in seiner Notwendigkeit jedoch auf diejenigen Terme, die zum Verständnis des eigentlichen Themas dieser Arbeit wichtig sind.

1.1 Überblick über die Gesamtthematik

Ein „Übersetzer“ oder „Compiler“ ist ein technisches Hilfsmittel, um aus einem als Zeichenkette vorliegendes Computerprogramm in maschinenles- und ausführbare Form zu bringen. Um einen Maschinencode zu erzeugen, steht die Analyse des als Zeichenkette vorliegenden Quellcodes im Mittelpunkt. Zunächst wird auf dieser eine lexikalische Analyse durchgeführt, deren Ergebnis eine Token-Kette ist. Den genauen Ablauf stellt Abbildung 1.1 grafisch dar.

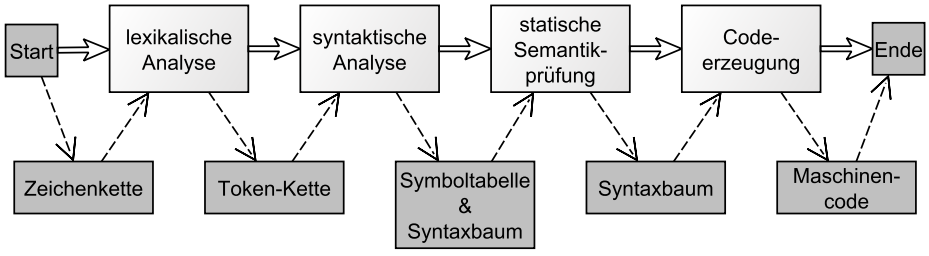

Abbildung 1.1: Einfacher Compiler

Dabei ist zu beachten, dass dies eine relativ einfache und triviale Realisierung eines Compilers darstellt. Diese ist natürlich auch voll funktionsfähig, birgt aber noch einiges Potential, denn dieser Compiler ist zwar schnell, erzeugt jedoch keinen effizienten Maschinencode. Darüber hinaus ist dieser Compiler abhängig von einem bestimmten Maschinentypen und kann schwerlich auf andere Architekturen übertragen werden.

Wünschenswert ist also eine möglichst maschinenunabhängige Realisierung eines Compilers, die darüber hinaus eine „Optimierungseinheit“ besitzt. Genau dies bietet ein optimierender Compiler. Die Idee des Konzeptes besteht darin, nicht - wie der einfache Compiler - direkt Maschinencode zu erzeugen, sondern diesen Schritt durch eine Zwischencodeerzeugung zu ersetzen.

Auf Basis dieses Zwischencodes lassen sich nun Maschinencode-unabhängige Optimierungen vornehmen, bevor der Maschinencode erzeugt wird. Als weiterer Optimierungsschritt kann dieser erzeugte Code nochmals Verbesserungen unterzogen werden, die abhängig von der spezifischen Maschinenarchitektur sind (siehe Abbildung 1.2).

Ziele der Optimierungen sind zum Einen die Vermeidung oder Reduzierung von Redundanzen und teuren Operationen, genauso wie die Vermeidung von Ineffizienzen. Realisiert werden soll dies - grob gesagt - durch die Entfernung unnötiger Befehle beziehungsweise deren Ersetzung durch effizientere Äquivalenzen.

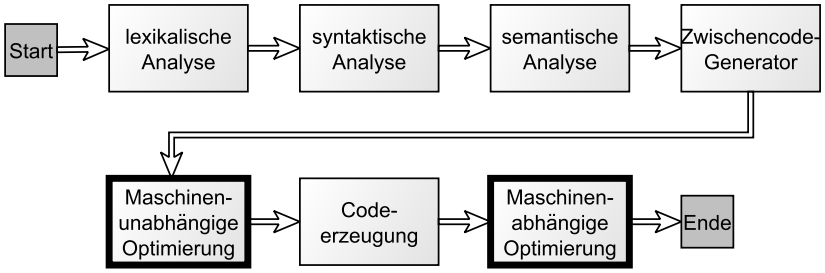

Abbildung 1.2: Optimierender Compiler

Effizienz bedeutet in diesem Zusammenhang, dass alle Ressourcen, die der Zielcomputer bietet, effizient genutzt werden.

Möglichkeiten zur Verbesserung des Codes bestehen zum Beispiel in der Wiederverwendung globaler gemeinsamer Teilausdrücke, Kopiepropagation, Eliminierung von totem Code, Reduzierung der Kosten oder durch die Verschiebung von Codeteilen, zum Beispiel aus Schleifen, um nur einige der wichtigsten Verfahren aufzuzählen.

1.2 Einordnung in den Kontext

Das Thema dieses Seminars beschäftigt sich mit dem Bereich der maschinenunabhängigen Optimierung. Die Grafik aus Abbildung 1.3 zeigt übersichtlich alle Einzelschritte, die dafür durchlaufen werden.

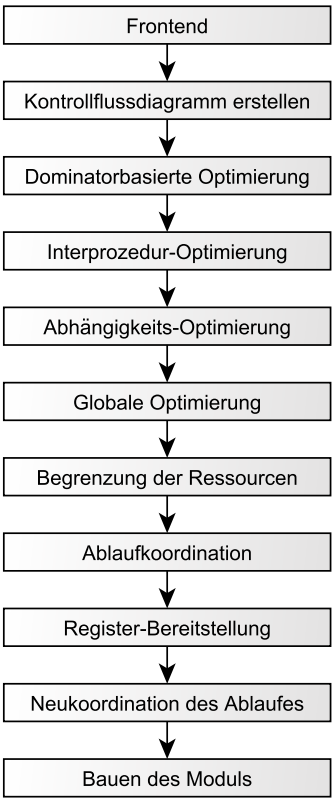

Abbildung 1.3: Compilerstruktur

Im ersten Moment mag der Begriff „Frontend“ verwundern, jedoch stellt dieser nur einen Sammelbe-

griff für alle vorangegangenen Schritte dar. Im Einzelnen sind dies die Schritte, die in Abbildung 1.2 bis zum Schritt „Zwischencode-Optimierung“ gezeigt werden. Eingangspunkt für die hier gezeigte Compilerstruktur ist also ein Syntaxbaum, auf dem dann optimiert wird.

In diesem Kontext nehmen Bereiche der Graphentheorie, Kontrollflussgraphen, lokale Optimierungen, Alias-Analyse und statische Zuordnungen wichtige Rollen ein, auf deren Basis die vorliegende Arbeit aufbaut.

1.3 Wichtige Ausdrücke und deren Bedeutung

Nachdem die vorherigen Abschnitte einen Überblick über das im Seminar bearbeitete Thema des Compilerbaus gegeben haben, widmen sich die folgenden Abschnitte meinem eigentlichen Thema. Dazu werde ich zunächst einige Begriffe einführen und deren genaue Bedeutung erörtern. Dabei habe ich mich auf die wichtigsten Kernbegriffe beschränkt, welche für das weitere Verständnis der Themenbereiche Schleifeninvarianten (Abschnitt 2.1 auf Seite 11), Erweiterung (Abschnitt 2.2 auf Seite 13), Neuordnung (Abschnitt 2.3 auf Seite 17) und Aufwandsreduzierung (Abschnitt 2.4 auf Seite 19) unumgänglich sind.

1.3.1 Kontrollflussgraph

Ein derartiger spezieller, gerichteter Graph ist ein Hilfsmittel, mit dem in der Informatik der Kontrollfluss von Programmen dargestellt werden kann. Er wird vor allem im Bereich der Programmoptimierung eingesetzt. Dabei wird der Eintrittspunkt in ein Computerprogramm durch den Wurzelknoten repräsentiert, während der Endpunkt durch einen Endknoten realisiert wird.

Auf triviale Art und Weise lassen sich in einem Kontrollflussgraphen auch Verzweigungen und Schleifen darstellen:

> „Wenn von einem Knoten mehrere Kanten wegführen (der Knoten also Quelle mehrerer gerichteter Kanten ist), so entspricht das einer Verzweigung. Schleifen finden sich als Zyklen in Kontrollflussgraphen wieder.“ [Kon(2014)]

Allgemein gesprochen muss in einem Graphen bestehend aus einer Menge von Knoten V, einem Wurzelknoten $r \in V$ und einer Menge gerichteter Kanten E jeder Knoten von der Wurzel aus erreicht werden. Anders formuliert: innerhalb eines gerichteten Kontrollflussgraphen $G = \langle V, E, r \rangle$ muss es einen Pfad von r zu jedem beliebigen $u \in V$ geben. Der in Abbildung 1.4 gezeigte Graph erfüllt diese Bedingungen.

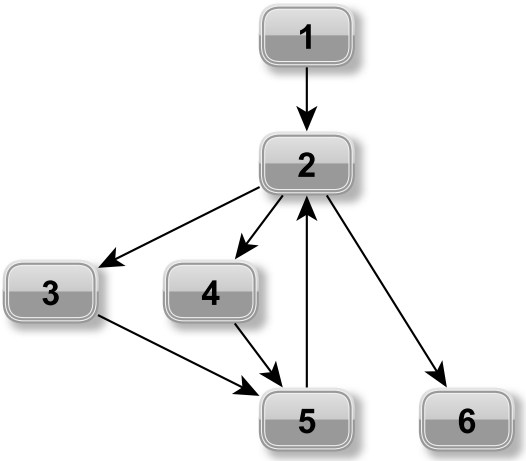

Abbildung 1.4: Kontrollflussgraph $G_1 = \langle V, E, 1 \rangle$

Neben der beschriebenen Form eines Kontrollflussgraphen existieren einige Erweiterungen und Abwandlungen, wie die Nennung eines bestimmten Endknotens $x \in V$. Dies führt jedoch für den hier behandelten Kontext zu weit. Als guter Einstiegspunkt zum Weiterlesen mit einem praktischen Beispiel eignet sich meines Erachtens Kapitel 8.3.2.1 ff. aus [Liggesmeyer(2009)].

1.3.2 Dominanzrelation

Eine derartige Relation beschreibt die Beziehung von Knoten innerhalb eines Kontrollflussgraphen zueinander. Innerhalb eines Graphen $G = \langle V, E, r\rangle$ dominiert ein Knoten u den Knoten v mit $u, v \in G$ genau dann, wenn jeder Pfad, der in r beginnt und in v endet, den Knoten u beinhaltet. Man schreibt dann auch $u \,\texttt{dom}\, v$.

Da jeder Knoten $w \in G$ sich selbst dominiert, ist die Dominanzrelation auch reflexiv. Ebenso ist die Dominanzrelation auch transitiv, denn aus $u \,\texttt{dom}\, v$ und $v \,\texttt{dom}\, w$ folgt, dass $u \,\texttt{dom}\, w$ gilt.

Man spricht genau dann von einer strikten Dominanz, wenn u den Knoten v dominiert, und zusätzlich $u \neq v$ gilt. Man notiert $u \,\texttt{stdom}\, v$ für diese Relation. Zur Verdeutlichung siehe auch [Dom(2014)].

1.3.3 Dominatorbaum

Will man die strikte Dominanzrelation grafisch darstellen, so lässt sich dies als Dominatorbaum umsetzen.

Als Beispiel soll hier wieder der Graph $G_1 = \langle V, E, 1\rangle$ aus Abbildung 1.4 dienen. Als erstes stellen wir fest, dass der Wurzelknoten mit dem Bezeichner 1 den Knoten 2 dominiert.

Ebenfalls offensichtlich ist, dass Knoten 2 den Knoten 5 dominiert, aber 3 dominiert 5 nicht, denn es gibt einen Pfad $1 \to 2 \to 4 \to 5$ der den dritten Knoten nicht beinhaltet. Entsprechend gilt insgesamt $1 \,\texttt{dom}\, 2$, $2 \,\texttt{dom}\, 3$, $2 \,\texttt{dom}\, 4$, $2 \,\texttt{dom}\, 5$, und $2 \,\texttt{dom}\, 6$. Weitere strikte Dominanzrelationen existieren nicht.

Aus dieser Aufstellung lässt sich der Graph aus Abbildung 1.5 zeichnen.

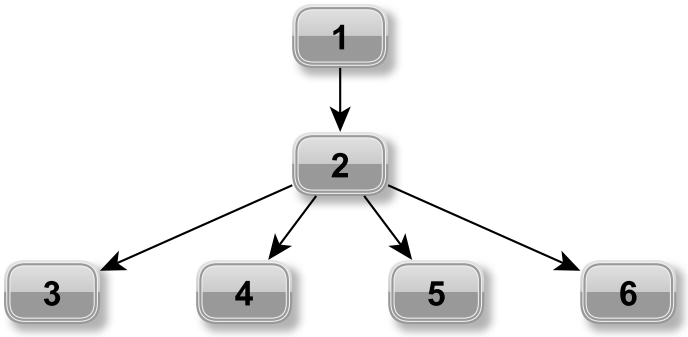

Abbildung 1.5: Dominatorbaum

1.4 Was werde ich zeigen?

Nachdem aus den vorangegangenen Abschnitten deutlich wurde, worum sich der Themenblock dreht, beschäftigt sich das nächste Kapitel mit der Umsetzung. Anhand der anstehenden Aufgaben und Tätigkeiten soll erreicht werden, dass ein vorliegender Zwischencode durch Neuordnung und Aufwandsreduzierung dominatorbasiert optimiert wird.

Hierzu werden im einzelnen zuerst Schleifeninvarianten bestimmt und Erweiterungen gefunden, die dann gesamtheitlich in einem zweiten Schritt neu angeordnet werden. Diese Neuanordnung führt dazu, dass der Aufwand zur Berechnung des Algorithmus im dritten Schritt reduziert werden kann.

Da dies nur einen Teilaspekt der Optimierung darstellt, mache ich auch auf potentielle Probleme und deren Abwägung aufmerksam.

Am Ende findet sich noch eine kurze Zusammenfassung dessen, was im Rahmen meiner Arbeit aufgezeigt wurde, und welche Themenbereiche einen guten Einstiegspunkt für weitere Recherchen im Zusammenhang mit optimierenden Compilern darstellen.

Kapitel 2

Dominatorbasierte Optimierung

2.1 Schleifeninvarianten

Ziel dieses Schrittes ist die Bestimmung der Schleifeninvarianten, um diese außerhalb der Schleife berechnen und vorhalten zu können. Das Prinzip sieht vor, diese in eine temporäre Variable auszulagern und während des Schleifendurchlaufes nicht zu benutzen. Dadurch kann der Aufwand der Schleifenberechnung deutlich reduziert werden.

Zentrale Aufgabe für diese „loop strength reduction“ (siehe Kapitel 8.4, [Morgan(1998)]) ist die Ersetzung von Multiplikationen durch wiederholte Additionen. Dafür werden drei verschiedene Operationen benötigt. Zuerst muss der Compiler diejenigen Variablen bestimmen, die sich während des Schleifendurchlaufes nicht ändern, also invariant sind.

Danach werden diejenigen Variablen bestimmt, die sich während des Schleifendurchlaufes ändern, um diese als neue temporäre Variablen einzufügen.

Zum Schluss müssen alle so erzeugten Operationen und Ausdrücke in eine sinnvolle Reihenfolge gebracht werden, sodass die Anzahl an Schleifeninvarianten möglichst groß wird. Gleichzeitig werden Multiplikationen der neu eingeführten Variablen durch wiederholte Addition von Schleifeninvarianten ersetzt.

2.1.1 Definition

> „Eine temporäre Variable `T` ist eine Schleifeninvariante der Schleife `L`, falls sie entweder nicht innerhalb der Schleife berechnet wird, oder wenn ihre Operanden Schleifeninvarianten sind.“ [Morgan(1998)], Seite 172, aus dem Englischen

In der Informatik ist eine Schleifeninvariante eine Sonderform der Invariante, die vor, während und nach der Ausführung einer Schleife in einem Algorithmus gültig ist. Sie ist damit unabhängig von der Zahl ihrer derzeitigen Durchläufe. [Sch(2014)]

Die Definition wurde so gewählt, damit diejenigen Schleifeninvarianten abgedeckt werden, die mehrere Operationen zur Berechnung benötigen. Falls zum Beispiel eine temporäre Variable `T` die Berechnung des Ausdrucks $(x + y) \cdot z$ repräsentiert, benötigt man zwei Operationen, um `T` zu berechnen. Falls die Definition anders aussähe, indem sie beschreiben würde, dass `T` nur genau dann eine Schleifeninvariante wäre, wenn seine Operanden außerhalb der Schleife ausgewertet werden, wäre `T` in diesem Beispiel keine Schleifeninvariante temporäre Variable, da $(x + y)$ innerhalb der Schleife ausgewertet würde.

Anders formuliert ist eine derartige temporäre Variable dadurch definiert, dass ihre Blätter im entsprechenden Ausdrucksbaum nicht innerhalb der Schleife ausgewertet werden.

Die unmittelbare Maßnahme ist demzufolge die Entfernung der entsprechenden Operationen aus der Schleife. Denn sobald diese Ausdrücke immer zum selben Ergebnis führen, sollten sie außerhalb berechnet werden. Allerdings ist dieses Vorgehen nicht sicher. Eine temporäre Variable ist unabhängig von ihrer Position in der Schleife eine Schleifeninvariante - sie könnte sich auch hinter einer Bedingung verbergen. Aber darum werden sich spätere Optimierungsschritte kümmern. Zum jetzigen Zeitpunkt ist es für den Compiler nur wichtig, welche Schleifeninvarianten existieren und welche nicht.

2.1.2 Berechnung

Um die Informationen, welche Schleifeninvarianten existieren, festzuhalten, wird ein Attribut `variant(T)` zu einer temporären Variable `T` hinzugefügt. Dieses Attribut beinhaltet die innerste Schleife des Schleifen-

baumes, in der `T` keine Schleifeninvariante ist. Falls `T` invariant in jeder Schleife ist, dann ist `variant(T)` die Wurzel des Schleifenbaumes. Falls `T` in keiner Schleife invariant ist, so ist `variant(T)` leer (bzw. `null`).

An dieser Stelle sei ein Querverweis auf einen anderen Vortrag dieses Seminars erlaubt (Thema: Static Single Assignment (SSA), vgl. Kapitel 7, [Morgan(1998)]), das die Basis dieser Berechnung bildet. Es gibt demnach, basierend auf der SSA-Form des Kontrollflussgraphen, eine einzige Definition für jede temporäre Variable, die wiederum die Nutzung der einzelnen Anweisungen dominiert (vgl.1.3.2).

Bevor wir mit der Beschreibung des Algorithmus zur Berechnung beginnen, betrachten wir zunächst jede Klasse von Instruktionen und ermitteln die Bedeutung der Schleifeninvarianz für jede dieser Arten.

Betrachte einen Knoten φ mit $\mathtt{T}_0 = \varphi(\mathtt{T}_1, ..., \mathtt{T}_n)$. Um festzustellen, dass $\mathtt{T}_0$ in jedem Schleifendurchlauf den selben Wert behält, muss der Compiler in Erfahrung bringen, welche die innerste Schleife ist, in der jeder der Operanden invariant ist. Außerdem muss er in Erfahrung bringen, welcher Teil des Kontrollflussdiagramms den Knoten φ beinhaltet und welcher Zweig dafür gewählt werden muss. Dabei ist die zweite Bedingung unmöglich zu berechnen. Daher wird der Compiler einfach annehmen, dass `variant` $(\mathtt{T}_0 = \varphi(\mathtt{T}_1, ..., \mathtt{T}_n))$ die innerste Schleife ist, in der der Knoten φ vorkommt.

Für Operationen, die einfache Funktionen sind (z.B. Addition, Multiplikation oder Disjunktion), verändern sich in der innersten Schleife, in der sich einer der Operanden ändert. Die (ebenfalls einfache) Kopierfunktion variiert in der gleichen Schleife, in der der Operand sich ändert. Dementsprechend variiert die `LOAD`-Anweisung in der tiefsten Schleife, in der eine `STORE`-Operation die selbe Position ändern könnte.

Der Compiler benötigt eine Hilfsfunktion, die den nächsten gemeinsamen Vorgänger zweier Knoten eines Baumes liefern kann. Diese Funktion liefert im hier beschriebenen Szenario den gesuchten Schleifenbaum. Dabei ist der Algorithmus 2.1 (entnommen aus [Morgan(1998)], Figure 8.11) relativ simpel. Falls ein Knoten ein Vorgänger des anderen Knotens ist, dann ist dieser Knoten der gesuchte. Anderenfalls wähle einen Knoten und beginne, den Baum aufwärts zu laufen. Vergleiche bei jedem Schritt, ob der neue Knoten der gesuchte ist.

Algorithmus 2.1 Finden des nächsten gemeinsamen Vorgängers

```
function Loop_Nearest_Ancestor(L1: Loop, L2: Loop) returns Loop
   if isancestor(L1, L2) then
      return L2;
   endif;
   L := L1;
   while NOT isancestor(L, L2) do
      L := LoopParent(L);
   endwhile;
   return L;
endfunction Loop_Nearest_Ancestor;
```

Für jede Anweisung berechnet der Compiler die innerste Schleife, in der die Berechnung variiert. Das benötigte Vorgehen stellt Algorithmus 2.2 (entnommen aus [Morgan(1998)], Figure 8.12) dar. Dieser Algorithmus bearbeitet einen Block, indem er zuerst über die φ-Knoten dieses Blockes iteriert. Dabei werden die Zielvariablen in dem Block verändert, sodass sie in der innersten Schleife variieren. Ziel ist es, die innerste Schleife des aktuellen Blockes `B` sowie den Punkt, an dem der Operand definiert wird, zu ermitteln. Dafür wird die in der Variablen `Varying` festgehaltene, partielle innerste Schleife mit der aktuellen verglichen. Falls der aktuelle Operand in einer weiter innen liegenden Schleife modifiziert wurde, wird diese Schleife diejenige, in der die Anweisung variiert. Nachdem alle Operanden betrachtet wurden, wird das Resultat für alle abhängigen Operanden festgehalten.

Algorithmus 2.2 Blockberechnung der Schleifeninvarianten

```
procedure Calculate_Loop_Invariants_Block(B: Block);
   foreach T0 = phi(T1,...,Tn) in phi(B) do
      variant(T0) = LoopParent(B);
   endfor;
   foreach I in B execution order do
      Varying = NULL;
      foreach T in Operands(I) do
         declare (T_varying: Loop = Loop_Nearest_Ancestor(variant(T), B));
         if Loop_Nearest_Ancestor(Varying, T_varying) then
            Varying = T_varying;
         endif;
      endfor;
      foreach T in Target(I) do
         variant(T) = Varying;
      endfor;
   endfor;
endprocedure Calculate_Loop_Invariants_Block
```

Die Prozedur zur Berechnung von Schleifeninvarianten in Algorithmus 2.3 (entnommen aus [Morgan(1998)], Figure 8.13) stellt sicher, dass die Varianz für jede temporäre Variable berechnet wurde, bevor diese benutzt wird. Ein Lauf durch den Dominatorbaum sorgt dafür, dass alle Operanden einen Wert für `variant(T)` haben.

Algorithmus 2.3 Berechnung von Schleifeninvarianten

```
procedure CALCULATE_LOOP_INVARIANTS;
   call CALCULATE_DOMINATOR_TREE;
   call CALCULATE_LOOP_TREE;
   foreach B in N preorder on dominator tree do
      call Calculate_Loop_Invariants_Block(B);
   endfor;
endprocedure CALCULATE_LOOP_INVARIANTS;
```

2.1.3 Beispiel

Betrachten wir zunächst den Kopf einer beispielhaften `while`-Schleife, um die Motivation der Bestimmung von Schleifeninvarianten zu verdeutlichen.

$$\texttt{while}\,(i \leqq \texttt{limit} - 2)$$

Obwohl sich das Ergebnis der Operation $\texttt{limit} - 2$ während der Schleifendurchläufe nicht ändert, würde diese ohne Optimierung dennoch bei jeder Iteration wieder ausgewertet. Für n Schleifendurchläufe würde diese Operation $n + 1$ mal ausgewertet werden müssen. Wenn wir also diesen Ausdruck ersetzen, ändert sich der Aufruf wie folgt:

$$t = \texttt{limit} - 2$$
$$\texttt{while}\,(i \leqq t)$$

Wie leicht zu sehen ist, wurde der invariante Teil der Schleife aus dieser entfernt. Außerdem wurde eine temporäre Variable t eingeführt, die innerhalb der Schleife nur noch gelesen und nicht kalkuliert werden muss. Für n Schleifendurchläufe muss die Kalkulation nur noch ein einziges Mal durchgeführt werden.

2.2 Erweiterung

Wie im Beispiel gerade gezeigt wurde, hilft die Identifikation der Schleifeninvarianten allein nicht weiter. Daher beschäftigen sich die folgenden Abschnitte mit der Auswahl passender Variablen, die eingefügt

werden sollen. Erst danach kann eine Restrukturierung im Rahmen der Optimierung stattfinden.

2.2.1 Reguläre Variabilität

In Schleifen können Variablen existieren, die ihren Wert regulär, also zum Beispiel per Inkrement oder Dekrement während des Durchlaufes verändern. Diese werden Induktionsvariablen oder Erweiterungsvariablen genannt. Operationen basierend auf derartigen Variablen, beispielsweise einfache Muliplikationen, können durch wiederholte Additionen ersetzt werden.

2.2.2 Kandidaten

Beginnen wir zunächst mit der Definition von Erweiterungskandidaten ([Morgan(1998)], Kapitel 8.5, aus dem Englischen):

In einer gegebenen Schleife L ist genau und ausschließlich dann eine Variable T ein Kandidat für L, wenn T in L ausgewertet wird und in einer der folgenden Formen vorliegt:

- $\mathtt{T} = \mathtt{T}_i \pm \mathtt{T}_j$, wobei einer der beiden Operanden ein Kandidat und der andere eine Schleifeninvariante von L ist
- $\mathtt{T} = \mathtt{T}_k$, wobei $\mathtt{T}_k$ ein Kandidat in L ist
- $\mathtt{T} = \pm\mathtt{T}_k$, wobei $\mathtt{T}_k$ ein Kandidat in L ist
- $\mathtt{T} = \mathtt{T}_i$, wobei $\mathtt{T}_i$ eine Schleifeninvariante von L ist
- $\mathtt{T} = \varphi(\mathtt{T}_1, ..., \mathtt{T}_m)$, wobei jeder der Operanden entweder ein Kandidat oder eine Schleifeninvariante von L ist

□

Ein Tupel von Erweiterungskandidaten wird dadurch berechnet, dass alle Anweisungen, die jede Variable auswerten, betrachtet werden. Dabei werden diejenigen entfernt, die nicht in der korrekten Form sind (also nicht einer der Formen aus der Definition entsprechen). Wenn die Anweisungen in ihrer Auswertungsreihenfolge betrachtet werden, kann daraus der Compiler erkennen, ob die Operanden Erweiterungskandidaten sind - sofern es sich nicht um φ-Knoten handelt.

Algorithmus 2.4 (aus: [Morgan(1998)], Figure 8.14) zeigt die programmatische Lösung dieser Aufgabe.

Algorithmus 2.4 Ermitteln von Erweiterungskandidaten

```
procedure CALCULATE_CANDIDATES(L: Loop)
   Candidates = NULL;
   Worklist = NULL;
   foreach B in L do
      foreach I in phi(B) UNION B where I is T=... do
         if Typeof(T) is integral then
            if T has form (see Definition) then
               add T to Candidates;
               add T to Worklist;
            endif;
         endif;
      endfor;
   endfor;
   while Worklist <> NULL do
      take T from Worklist;
      call CANDIDATE_PRUNE(T);
      if T not in Candidates then
         foreach I in Uses(T) where I is in L do
            foreach U in Targets (I) do
               if U in Candidates AND U not in Worklist then
                  add U to Worklist;
               endif;
            endfor;
         endfor;
      endif;
   endwhile;
endprocedure CALCULATE_CANDIDATES;
```

2.2.3 Kandidatenkürzung

Falls es sich um φ-Knoten handelt, die in Algorithmus 2.4 verarbeitet werden, muss zuvor die Liste der Kandidaten bereinigt werden. Es könnte nämlich passieren, dass einige der Operanden noch nicht bearbeitet wurden. Deshalb greift der Compiler in diesem Algorithmus eine Arbeitsliste zurück, in der zuerst angenommen wird, dass alle Variablen Erweiterungskandidaten sind. Sie werden erst aus der Arbeitsliste entfernt, wenn die Annahme widerlegt wurde. Für diesen Mechanismus sorgt Algorithmus 2.5 (aus: [Morgan(1998)], Figure 8.15)

2.2.4 Erweiterungstupel

Definition Erweiterungstupel ([Morgan(1998)], „Definition: Induction Sets and Temporaries“, aus dem Englischen):

> „Eine Induktionsvariable `T` in einer Schleife `L` ist ein Erweiterungskandidat mit den nachfolgenden Eigenschaften. Betrachte den Graphen mit Erweiterungskandidaten als Knoten und Kanten zwischen Kandidaten `T` und `U`, falls `T` dazu verwendet wird, den Wert von `U` zu berechnen. Eine Induktionsvariable ist ein Erweiterungskandidat, der ein Teil eines stark verknüpften Bereiches in diesem Graphen ist. Die Variablen innerhalb dieses stark verknüpften Bereiches werden Erweiterungstupel genannt.“

Mit anderen Worten wird die temporäre Variable `T` dazu genutzt, andere temporäre Variablen zu berechnen, auf deren Basis wieder andere berechnet werden. Dies geschieht so lange, bis in der nächsten Iterationsstufe der Wert von `T` berechnet wird. Schlussendlich wird der Wert von `T` benutzt, um den Wert von `T` selbst zu berechnen. Algorithmus 2.6 beschreibt dieses Vorgehen programmatisch.

Falls die betrachtete Schleife mehr als einen Einstiegspunkt hat, so halten wir uns nicht damit auf. Anderenfalls werden zuerst die Erweiterungskandidaten berechnet. Jeder stark verknüpfte Bereich mit mindestens zwei Elementen und einer temporären Variable, die Ziel eines φ-Knotens am Einstiegspunkt ist, ist ein Erweiterungstupel.

Algorithmus 2.5 Ausdünnung der Kandidatenliste

```
procedure CANDIDATE_PRUNE(T: temporary)
   case on form of I do
      T=phi(T1,...,Tm):
         for i=1 to m do
            if Ti not in Candidates AND Ti not loop invariant in L then
               remove T from Candidates;
               return;
            endif;
         endfor;
      T=Ti +- Tj:
         if Ti in Candidates AND Tj is loop invariant in L then
            return;
         elseif Tj in Candidates AND Ti is loop invariant in L then
            return;
         else
            remove T from Candidates;
            return;
         endif;
      T= +- T1 or T = T1:
         if Ti not in Candidates then
            remove T from Candidates;
            return;
         endif;
       otherwise:
         system error;
   endcase;
endprocedure CANDIDATE_PRUNE;
```

Algorithmus 2.6 Berechnung des Erweiterungstupels

```
procedure CALCULATE_INDUCTION(L: Loop)
   if L is of kind single_entry_loop then
      call CALCULATE_CANDIDATES(L)
      Consider graph with nodes being Candidates,
         the root begins with the phi-nodes in LoopEntry(L);
         an edge between T and T' if T is used as an
         operand of T'.
      Compute the strongly connected regions of this graph.
      Anchors = {T|T is a target of a phi-node in LoopEntry(L)}
      foreach strongly connected region S do
         if |S|>1 and (Anchors intersecting S) is not empty then
            add S to the collection of induction sets;
         endif;
      endfor;
   endif;
endprocedure CALCULATE_INDUCTION;
```

2.2.5 Beispiel

Als Beispiel betrachten wir Abbildung 2.1. Darin beschreibt die linke Spalte den Kontrollflussgraphen der Schleife, während die rechte Spalte den (impliziten) Graphen der Erweiterungskandidaten darstellt.

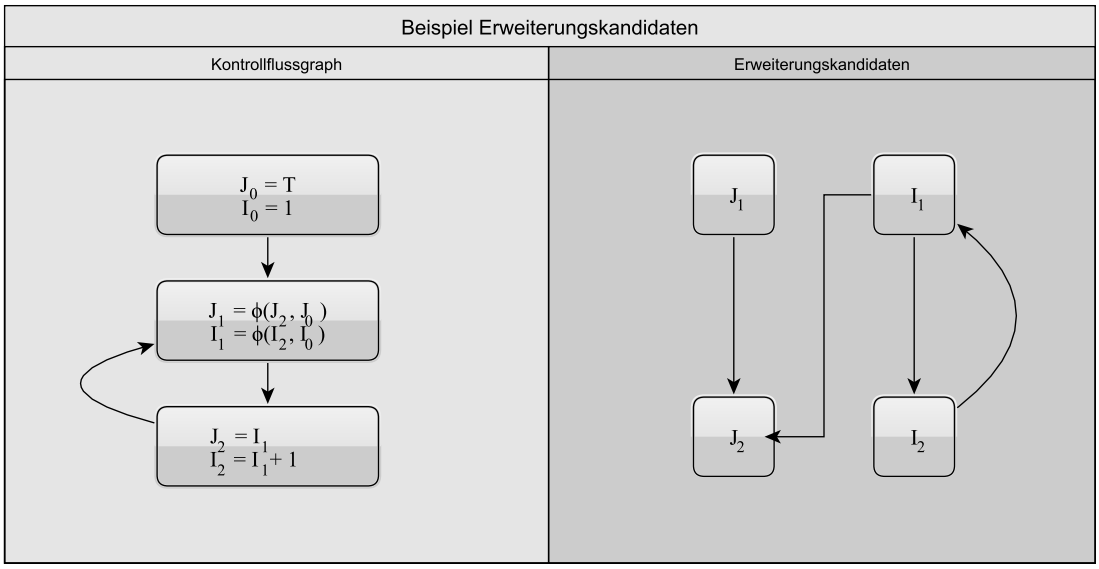

Abbildung 2.1: Beispiel Erweiterungskandidaten

J_1 und J_2 liegen dabei jedoch nicht in einem stark verknüpften Bereich. Falls man in J_1 begonnen und den Bogen rückwärts durchlaufen hat, kann man niemals zu J_1 zurückkehren. Das Tupel (I_1, I_2) stellt hier also das Erweiterungstupel dar.

2.3 Neuordnung

An diesem Punkt hat der Compiler nun die Informationen darüber bestimmt, welche Ausdrücke außerhalb der Schleife berechnet werden können, welche Ausdrücke Schleifeninvarianten sind, und wie die Erweiterungen bestimmt werden. Damit sind alle Voraussetzungen gegeben, eine Neuordnung vorzunehmen.

Ziel der Neuordnung ist es, die einzelnen Ausdrücke derart zu restrukturieren, dass möglichst viel außerhalb der Schleife berechnet werden kann. In einem späteren Schritt erfolgt dann eine partielle Ent-

fernung von Redundanzen.

Diese Phase der Neuordnung wird „reshape“ genannt.

2.3.1 Assoziativität, Distributivität - Wie hilft mir das?

Der Compiler nutzt zur Umformung die Assoziativität und Distributivität sowohl für Ganzzahlarithmetik als auch für logische Arithmetik aus, um durch Verschiebung eine Verbesserung zu erreichen.

Angenommen, es gibt n ineinander geschachtelte Schleifen $\mathtt{L}_1$ bis $\mathtt{L}_n$. Während der Neuordnungsphase reorganisiert der Compiler jeden Ausdruck $\mathtt{E}$, sodass er der Form

$$\mathtt{E} = \mathtt{E}' + (\mathtt{LC}_1 + (\mathtt{LC}_2 + (\mathtt{LC}_3 + \ldots + \mathtt{LC}_n)))$$

entspricht. Dabei ist $\mathtt{E}'$ ein Ausdruck, der keine Schleifeninvariante der innersten Schleife ist. $\mathtt{LC}_1$ ist unterdessen die Schleifeninvariante der innersten Schleife, $\mathtt{LC}_2$ die der nächsten übergreifenden Schleife, und so weiter, bis $\mathtt{LC}_n$ die Schleifeninvariante der äußersten Schleife ist.

Die gleiche Transformation wird entsprechend für Multiplikation und logische Operatoren durchgeführt. Diese Transformation erlaubt dem Compiler, $\mathtt{E}'$ in der innersten Schleife zu berechnen, $\mathtt{LC}_1$ in der nächst übergreifenden. Dies wird wiederholt, so dass $\mathtt{LC}_n$ entsprechend außerhalb der äußersten Schleife berechnet werden kann.

Je weiter rechts ein Operand also steht, um so weiter in einer übergreifendenden Schleife kann dieser berechnet werden.

Der Ausdruck für $\mathtt{E}'$ kann danach so umgeformt werden, dass die Induktionsvariablen herausgezogen werden können. Die wichtigste Form der Aufwandsreduzierung betrifft die Multiplikation. Falls ein Faktor eine Erweiterungsvariable ist und der andere Faktor sich innerhalb der Schleife nicht ändert, so kann dieses Produkt durch die wiederholte Ausführung von Additionen bei jedem Schleifendurchlauf einfach ersetzt werden. Um diese Fälle zu identifizieren, teilt der Compiler $\mathtt{E}'$ in Summanden der folgenden Form auf:

$$\mathtt{E}' = \mathtt{E}'' + \mathtt{FD}_1 \cdot \mathtt{I}_1 + \mathtt{FD}_2 \cdot \mathtt{I}_2 + \ldots + \mathtt{FD}_m \cdot \mathtt{I}_m$$

Dabei beschreibt $\mathtt{FD}_j$ eine Schleifeninvariante ($\mathtt{FD}$ steht für „First Difference“) und $\mathtt{I}_j$ eine Induktionsvariable der innersten Schleife. Die Erweiterungsvariablen für die äußeren Schleifen sind für diesen betrachteten Teil konstant, bleiben also irrelevant für diesen Ausdruck.

2.3.2 Gefahr: Pessimierung?

Es existiert allerdings eine latente Gefahr bei dieser Transformation. Die Neuanordnung all dieser Ausdrücke könnte dazu führen, dass der generierte Code sich deutlich vermehrt. Damit könnte eine Verlangsamung bei der Ausführung verbunden sein. Meistens wird dies jedoch nicht geschehen. Der Compiler versucht, diese Situationen im Vorhinein durch die folgenden Hilfsmittel zu verhindern.

Zunächst bettet der Compiler das beschriebene Vorgehen in einen Durchlauf des Dominatorbaumes ein, um die redundanten Ausdrücke zu eliminieren. Auf diese Weise wird jeder Ausdruck wahrscheinlich einmal ausgewertet. In dieser Phase jedoch entfernt der Compiler nicht die ursprünglichen Ausdrücke, sondern erledigt dies erst zu einem späteren Zeitpunkt, nach der globalen Optimierung, im Rahmen der „dead code elimination“.

Die Begründung ist darin zu finden, dass derartige Verschiebungen Ausdrücke in Schleifen verschieben können, insbesondere in deren Fallunterscheidungen. Aus Sicherheitsgründen kann der Compiler diese Ausdrücke aber nicht aus diesen Fallunterscheidungen heraus bewegen. Also belässt der Compiler die ursprünglichen Ausdrücke an Ort und stelle, was dazu führt, dass sie dort verfügbar sind, wo der Programmierer sie ursprünglich positioniert hat. Falls der Compiler einen Ausdruck nicht transformiert hat, wird er herausfinden, dass der verschobene Ausdruck redundant ist, und ihn von der Schleife entfernen.

2.3.3 Verwendete Eigenschaften

Für die Neuordnung werden insgesamt vier verschiedene Eigenschaften von Operationen verwendet. Vorrangig werden im Folgenden Addition, Subtraktion und Multiplikation betrachtet. Daneben sei aufgeführt, dass es es aber noch viele weitere Operationen mit der selben Charakteristik gibt. Der Compiler verwendet den selben Mechanismus für alle derartigen Operationen.

Die erste verwendete Eigenschaft ist die Kommutativität. Hierbei gilt, dass

$$x + y = y + x$$

ist. Für diese Operatoren kann der Compiler in jeder beliebigen Reihenfolge anordnen, um so redundante Ausdrücke zu erkennen.

Bei Operationen wie Subtraktionen werden Kombinationen von Addition und Negation entsprechend des Kommutativgesetzes neu angeordnet. Realisiert wird dies durch die Speicherung eines Negationsflags, der nach der Verarbeitung anzeigt, dass eine Subtraktion anstatt einer Addition erzeugt werden muss.

Assoziative Operationen erlauben mehr Verarbeitung. Angenommen, ein assoziativer Operator ist die Wurzel eines Ausdrucksbaumes, der den Term $(x + y)+(w + z)$ darstellt. Dann kann an der Wurzel dieses Baumes der Ausdruck mit den Operanden x, y, w, z umgeformt werden zu $(x + (y + (w + z)))$. Falls die Operation darüber hinaus auch noch Kommutativ ist, können die Operanden so umsortiert werden, dass die Ausdrücke für die innerste Schleife zuerst berechnet werden, dann die darüber, und so weiter.

Die Multiplikation als Beispiel für eine Distributive Operation bilden die vierte Kategorie. Dabei kann die Regel

$$x \cdot (y + z) = x \cdot y + x \cdot z$$

dazu benutzt werden, Kombinationen von Multiplikationen, Subtraktionen und Additionen als Summe von Produkten umzuformen.

2.3.4 Baumdurchlauf

Um die Neuordnung vorzunehmen, analysiert der Compiler den Ausdrucksbaum, um ihn als Summe von Produkten miteinander zu kombinieren. Der Compiler unterbricht den Baumdurchlauf, sobald er auf eine Konstante, `LOAD`-Anweisung, temporäre Variable oder Induktionsvariable trifft.

Sobald er den Baum eingelesen hat, beginnt er mit der Neuanordnung, wie dies im vorhergehenden Abschnitt beschrieben wurde. Dabei ist zu beachten, dass das Distributivgesetz in der korrekten Reihenfolge angewendet wird. Es wird so lange weitergemacht, bis keine weitere Neuordnung möglich ist. Dieser beschriebene Algorithmus wird „gierig“ genannt (Greedy-Algorithmus).

Als Ergebnis dieses Durchlaufes liegt der neu geformte Ausdrucksbaum im temporären Speicher. Zum Abschluss muss dieser noch in Ausweisungen übersetzt werden, die in den Kontrollflussgraphen eingesetzt werden. Auch hier werden wieder zunächst die alten Ausdrücke stehen gelassen und ein Dominatorbaum-Durchlauf dazu genutzt, Redundanzen zu entfernen.

2.4 Aufwandsreduzierung

An diesem Punkt sind die aufwendigen Vorarbeiten für die Aufwandsreduzierung abgeschlossen. In den folgenden Abschnitten widmen wir uns daher der Umsetzung der Aufwandsreduzierung, und welche Randbedingungen diesbezüglich zu beachten sind.

2.4.1 Aufgabenstellung

Zur besseren Verdeutlichung betrachten wir an dieser Stelle den Term, welchen Morgan in seinem Buch [Morgan(1998)] in Kapitel 8.7 „Strength Reduction“ einführt:

$$E = FD_1 \cdot I_1 + FD_2 \cdot I_2 + FD_j \cdot I_j + FD_m \cdot I_m + (LC_1 + (LC_2 + (LC_3 + ... + LC_n)))$$

Dabei sind I_j die Induktionsvariablen und alle anderen Ausdrücke sind Schleifeninvarianten der Schleife L.

2.4.2 Frühe Berechnung

Die Idee der Aufwandsreduzierung ist es, diesen Term vor dem Eintreten in die Schleife zu berechnen und jedes mal zu aktualisieren, sobald sich ein Operand ändert. Da die einzigen Operanden, die sich ändern können, die Induktionsvariablen sind, wird der Term E jedes mal angepasst, wenn sich eine der I_j ändert.

Da der Kontrollflussgraph zu diesem Zeitpunkt in SSA-Form vorliegt, kann der Compiler die temporäre Variable E nicht aktualisieren. Stattdessen generiert er eine Sammlung von Variablen $E_0, ..., E_q$, die jedes mal wächst, wenn eine Erweiterungsvariable sich ändert. Daraus folgt aber auch, dass der Compiler alle Verwendungen von E mit der neuen Variable ersetzen. Dies realisiert er, indem der Compiler sich eine Liste von redundanten Ausdrücken vorhält, und anhand des Dominatorbaumes durch die Schleife läuft. Auf diese effiziente Weise können alle Vorkommen ausgetauscht werden.

2.4.3 Beispiele

Die Variable E ändert sich in verschiedenen Szenarien, nämlich genau dann, wenn sich eine der Variablen aus dem Erweiterungstupel $\mathtt{IS_i}$ändert:

- Falls $\mathtt{T_1}$ und $\mathtt{T_2}$ Teil von $\mathtt{IS_i}$ mit $\mathtt{T_1} = \mathtt{T_2} \pm \mathtt{RC}$ sind, aktualisiere den Wert von E, indem die Berechnung $E_1 = E_2 \pm \mathtt{RC} \cdot \mathtt{FD_i}$ nach der Auswertung von $\mathtt{T_1}$. Falls der Compiler dabei feststellt, dass $\mathtt{RC} \cdot \mathtt{FD_i}$bereits verfügbar ist, braucht die Auswertung nicht in den Kontrollflussgraphen eingebaut werden. Ansonsten muss die Multiplikation an dieser Stelle eingefügt werden, um später von einer partiellen Redundanzen-Bereinigung entfernt zu werden.
- Falls $\mathtt{T_1}$ und $\mathtt{T_2}$ Teil von $\mathtt{IS_i}$ mit $\mathtt{T_1} = \pm\mathtt{T_2}$ oder $\mathtt{T_1} = \mathtt{T_2}$sind, füge eine Berechnung von $E_1 = \pm E_2$direkt nach der Zuweisung zu $\mathtt{T_1}$ ein.
- Falls $\mathtt{T_0}$ eine Variable im Erweiterungstupel mit $\mathtt{T_0} = \varphi(\mathtt{T_1}, ..., \mathtt{T_m})$ ist, dann muss für E ein φ-Knoten oben in diesem Block eingefügt werden.
- Falls $\mathtt{T_i}$ nicht im Erweiterungstupel ist, dann muss für E am Ende des Blockes eingefügt und eine temporäre Variable für die Zuordnung des φ-Knotens angelegt werden.

Kapitel 3

Zusammenfassung

Das letzte Kapitel dieser Seminararbeit fasst nochmal die behandelten Punkte zusammen, bevor ich mein persönliches Fazit des Themenkomplexes „Compilerbau“ gebe. Während meiner Arbeit bin ich auf viele weitere, interessante Quellen und Ansätze gestoßen, auf die ich im Ausblick gerne hinweisen möchte. Dabei versuche ich, diese auch thematisch zuzuordnen.

3.1 Was wurde gezeigt?

Zu Beginn meiner Arbeit habe ich zunächst die Grundbegriffe wie Kontrollflussgraph, Dominanzrelation oder Dominatorbaum eingeführt, um darauf im späteren Verlauf zurückgreifen zu können. Auf deren Basis konnte ich mich meinem eigentlichen Thema widmen, dem Bereich Erweiterung, Neuordnung und Reduzierung.

Basis für die Erweiterung, beziehungsweise die Bestimmung entsprechender Kandidaten, musste zunächst der Begriff „Schleifeninvariante“ geklärt werden.

Im Rahmen der Erweiterung wurden zunächst die „reguläre Variablilität“ erörtert, welche die Idee - nämlich beispielsweise die Ersetzung einer einfachen Multiplikation durch wiederholte Addition - verdeutlicht. In diesem Zusammenhang wurde der Begriff der Induktions- oder Erweiterungsvariable eingeführt. Dazu wurde im nächsten Schritt erst ein Kandidat definiert, die Kürzung einer Liste von Kandidaten, bevor schlussendlich die Bildung eines Erweiterungstupels beschrieben wurde.

Im nachfolgenden Schritt wurde die Neuordnung behandelt, sowie potentielle Probleme dieser „reshape“-Phase besprochen. Die verallgemeinernde Darstellung des Zustandes als Term von Induktionsvariablen und Schleifeninvarianten halfen im letzten Abschnitt dabei, das Thema „frühe Berechnung“ zu vertiefen, um so diese Phase der Optimierung abzuschließen.

3.2 Fazit

Als ich begann, mich mit dem Thema Compilerbau zu beschäftigen, hatte ich keine genaue Vorstellung, was ein optimierender Compiler genau tut. Hilfreich war jedoch, dass mich das Thema , wie ein Compiler genau arbeitet, schon vorher interessiert hat. Allerdings hatte ich zuvor immer nur einen einfachen Compiler im Kopf, wenn ich mich mit dieser Thematik beschäftigt habe (vergleiche Abbildung 1.1).

Während meiner Erfahrung als Java Softwareentwickler habe ich zwar häufig Neuerungen und Änderungen in den einzelnen Java-Versionen verfolgt. Allerdings konnte ich keinen direkten Bezug zur Optimierung meines eingegebenen Quelltextes durch einen Compiler herstellen.

Hier hat mir mein Seminarthema geholfen, zumindest den Teil der Maschinenunabhängigen Optimierung besser verstehen zu können. Natürlich ist mir bewusst, dass ich bisher nur die Spitze des Eisberges habe betrachten können. Trotzdem kann ich nun einige Optmierungsmöglichkeiten, die ein Compiler hat, besser nachvollziehen, und habe eine kleinen Einblick in die Probleme und Grenzen, denen ein Compiler unterliegt, erhalten können.

Im privaten Rahmen werde ich mich - neben den Themen, die die anderen Teilnehmer meines Seminars ausgearbeitet haben - in naher Zukunft auch mit der maschinenabhängigen Optimierung beschäftigen. Auch, wenn ich dieses Wissen nur indirekt in meinem Beruf als Softwareentwickler werde anwenden können, halte ich es dennoch für wichtig, den bekannten „Blick über den Tellerrand“ zu bekommen. Denn

direkt werde ich dieses Wissen verwenden können, um Neuerungen in den Unterschiedlichen Programmiersprachen besser verstehen zu können, und mir so einen frühzeitigen Überblick über neue Technologien verschaffen zu können. Dies bringt nicht nur mir persönlich einen Vorteil, sondern kann auch meine Position bei meinem Arbeitgeber stärken.

Zusammenfassend kann ich behaupten, dass mich die Gesamtthematik sehr interessiert, und die Entscheidung, dieses Seminar zu besuchen, auch wenn ich kaum Vorwissen im Compilerbau hatte, richtig war.

3.3 Ausblick

Ein guter Einstiegspunkt, um das Vorgehen eines einfachen Compilers zu verstehen, empfiehlt es sich, zunächst die Begriffe Sprache (reguläre und kontextfreie), Syntax, Parsing, Grammatik und Semantik in ihrer Tiefe zu verstehen. Standardwerk in diesem Zusammenhang ist „Grundlagen und Techniken des Compilerbaus“ von N. Wirth, [Wirth(1997)].

Zusammen mit dem genannten Buch und diesem Seminar (beziehungsweise meiner Ausarbeitung und die der anderen Teilnehmer) zeichnet sich ein recht deutliches Bild von allen Schritten bis zur Maschinenunabhängigen Optimierung. Ausgeklammert wird hier jedoch noch der dritte und letzte Teil der Maschinen- bzw. Architekturabhängigen Optimierung. Hier hilft das Werk [Aho(2008)], auch als „Drachenbuch“ bekannt, weiter.

Wenn man sich insgesamt mit dem Entwurf und der Programmierung von Compilern intensiver (z.B. für die Abschlussarbeit oder sogar beruflich) beschäftigen möchte, sollte man sich an vielen deutschen Hochschulen mit der Informatik-Disziplin Compiler- oder Übersetzerbau auseinandersetzen. Für die Motivation dieses Bereiches eignet sich meines Erachtens [Wilhelm et al.(2012)Wilhelm, Seidl, and Hack] recht gut.

Anhang

Auf den folgenden Seiten finden sich die Beschreibungen zu den Verweisen aus der Seminararbeit, sowie ein Index für Abbildungen, Algorithmen und Stichworte, um die Verwendung dieser Arbeit als Nachschlagewerk zu vereinfachen. Dabei findet sich im Literaturverzeichnis auch weiterführende Literatur wieder, auf die in der vorliegenden Ausarbeitung nur am Rande eingegangen werden konnte.

Abbildungsverzeichnis

Algorithmenverzeichnis

Index

Literaturverzeichnis

[Morgan(1998)] R. Morgan. *Building and Optimizing Compiler*. Butterworth-Heinemann, 1998. ISBN 9781555581794. URL http://books.google.de/books?id=CXlhQgAACAAJ. 5, 11, 12, 13, 14, 15, 19

[Kon(2014)] Kontrollflussgraph, 05 2014. URL http://de.wikipedia.org/wiki/Kontrollflussgraph. 9

[Liggesmeyer(2009)] P. Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2009. ISBN 9783827420565. URL http://books.google.de/books?id=-eoerZI2fYIC. 9

[Dom(2014)] Dominanzrelation, 2014. URL http://de.wikipedia.org/wiki/Dominanzrelation_(Kontrollflussgraph). 10

[Sch(2014)] Schleifeninvariante, 2014. URL http://de.wikipedia.org/wiki/Schleifeninvariante. 11

[Wirth(1997)] N. Wirth. *Grundlagen und Techniken des Compilerbaus*. Walter de Gruyter, 1997. ISBN 9783486243741. URL http://books.google.de/books?id=6tt_ngEACAAJ. 22

[Aho(2008)] A.V. Aho. *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Studium Informatik. Pearson Education Deutschland, 2008. ISBN 9783827370976. URL http://books.google.de/books?id=pTKAwL64NkoC. 22

[Wilhelm et al.(2012)Wilhelm, Seidl, and Hack] R. Wilhelm, H. Seidl, and S. Hack. *Übersetzerbau: Band 2: Syntaktische und semantische Analyse*. SpringerLink : Bücher. Springer, 2012. ISBN 9783642011351. URL http://books.google.de/books?id=0LSSVHYZHC4C. 22