11. Kapitel

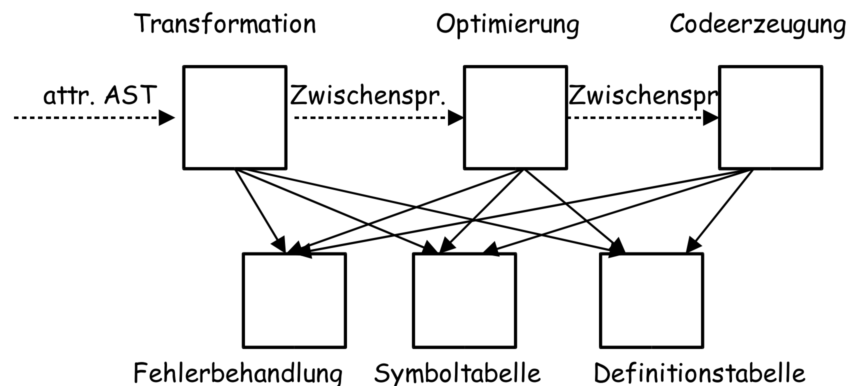
Optimierung
Teil 2

Einige Verfahren

Kapitel 11: Optimierungen

0. Einbettung
1. Elimination gemeinsamer Teilausdrücke
2. Wertnumerierung
3. Weitere globale Optimierungen
4. Operatorvereinfachung
4. Nachoptimierung

11. Optimierung und optimierende Codeerzeugung



11. Optimierung

- Aufgaben der globalen Optimierung:
  Zwischensprachenrepräsentation so umformen, daß anschließend die Codeselektion besseren Code erzeugt
- optimierende Codeerzeugung:
  Codeerzeugung unter Berücksichtigung fortgeschrittener Techniken zur Registerzuteilung und Befehlsanordnung
- Nachoptimierung (*peephole optimization*):
  lokale Verbesserung des erzeugten Codes nach Abschluß der Codeerzeugung

- Beachte: Optimierung produziert nur selten optimalen Code; sie verbessert nur den Code
- Begriff „optimal" hängt von Voraussetzungen ab, z.B. Codeerzeugung aus Baum/DAG, Alias-Analyse ja/nein, usw.

11. Das Vollbeschäftigungstheorem

Satz (Rice, 1953): Zu jedem algorithmisch arbeitenden Übersetzer *U* gibt es einen Übersetzer *U'*, der für bestimmte Programme kürzeren Code erzeugt.

Korollar [Vollbeschäftigungstheorem für Übersetzerbauer]:
Zu jedem optimierenden Übersetzer gibt es einen besseren.

Beweis des Satzes:

Annahme: es gibt einen Übersetzer *U*, der jedes Programm π mit algorithmischen Methoden in das absolut kürzeste Programm Opt(π) mit gleichem Ein-/Ausgabeverhalten übersetzt.

Sei π ein nicht haltendes Programm ohne E/A. Dann wird π von *U* übersetzt in Opt(π): *m*: **goto** *m*

Um zu prüfen, ob π nicht hält, muß man nur Opt(π) berechnen und das Ergebnis inspizieren. Damit ist das Halteproblem gelöst. Da das Halteproblem unentscheidbar ist, kann *U* nicht existieren. q.e.d

Kapitel 11: Optimierungen

0. Einbettung
1. Elimination gemeinsamer Teilausdrücke
2. Wertnumerierung
3. Weitere globale Optimierungen
4. Operatorvereinfachung
4. Nachoptimierung

11.1 Elimination gemeinsamer Teilausdrücke (*CSE*)

- Beobachtung:
  Adreßrechnung führt häufig zur Wiederholung von Berechnungen
- Ziel: lokal in einem Grundblock jeden Ausdruck nur einmal berechnen
  - Ergebnis bei Bedarf wieder verwenden
- Anwendung: vor allem wiederholte Berechnung von Adreßausdrücken vermeiden
- Verfahren Wertnumerierung: Dem Wert jedes Ausdrucks eine eindeutige Nummer zuordnen, mit dem er wieder erkannt werden kann
  - Bedingung: nur Ausdrücke mit gleichem Wert erhalten gleiche Wertnummer

11.1 Erkennen gleicher Ausdrücke

- Zwei Ausdrücke heißen kongruent, wenn sei den gleichen Operator und die gleichen Operanden haben
- Zwei Ausdrücke sind syntaktisch gleich, wenn sie kongruent sind und die Operanden die gleichen Wertnummern besitzen
  - Erweiterung auf semantische Gleichheit, z.B. $a+a = 2*a$, möglich
- Implementierung mit Haschtabelle:
  - Schlüssel - Operatoren τ, *LD, ST*. Kollisionsliste besteht aus Paaren von Operanden, d.h. Wertnummern.
- Globalisierung (über Grundblockgrenzen hinaus) macht Probleme
  - aber für erweiterte Grundblöcke möglich!

11.1 Abhängigkeitsmengen eines Tupels t

Ein Tupel t kann in Beziehung zu anderen Tupeln t' stehen:
Unterscheide:

- Verwendet: $U(t) = \{t' \mid t'$ ist ein Operand in $t\}$
- Definiert: $D(t) = \{t' \mid t'$ wird definiert durch $t\}$
- Invalidiert: $X(t) = \{t' \mid t'$ ungültig gemacht durch $t\}$

Bei Speicherzugriffen auf Adresse $>a<$ zusätzlich:

- Ausgabe: $aus(t) = \{t' \mid t$ definiert gleichen Speicherplatz a wie zuletzt $t'\}$
- Anti: $anti(t) = \{t' \mid t'$ liest von Speicherplatz a, den t danach definiert$\}$

Es gilt:

- $t' \in aus(t) \Rightarrow t' \in X(t)$, $t' \in anti(t) \Rightarrow t' \in X(t)$,

11.1 Beispiel

```
                 s1: ST   >c<    2       c=a+1+b;    t10: LD   <a>
                 s2: LD   <c>                        t11: ADD  t10   1
c=0;             s3: GT    0                         t12: LD   <b>
if x > 0 {       s4: JMP FALSE u1                    t13: ADD  t11   t12
                                                     t14: ST   >c<   t13
                 t1: ST   >a<    2                   t15: JMP  u1
                 t2: LD    <a>
                 t3: LD    <x>
                 t4: MUL   t2    t3      }           u1: LD    <c>
                 t5: ADD   t4    1       x=c;        u2: ST     x    u1
                 t6: ST    >b<   t5
a=2;             t7: LD    <x>
b=a*x+1;         t8: MUL    2    t7
a=2*x;           t9: ST    >a<   t8
```

11.1 Abhängigkeitsmengen eines Tupels t

Verwendet: $U(t) = \{t' \mid t'$ ist ein Operand in $t\}$

Definiert: $D(t) = \{t' \mid t'$ wird definiert durch $t\}$

Invalidiert: $X(t) = \{t' \mid t'$ ungültig gemacht durch $t\}$

	U	D	X
t_1: ST $>a<$ 2	$\{\}$	$\{<a>\}$	$\{t_2. t_3, t_4 + 1, t_{10} + 1, t_{11} + t_{12}\}$
t_2: LD $<a>$	$\{\}$	$\{<a>\}$	$\{\}$
t_3: LD $<x>$	$\{\}$	$\{<x>\}$	$\{\}$
t_4: MUL t_2 t_3	$\{<a>, <x>\}$	$\{t_2, t_3\}$	$\{\}$
t_5: ADD t_4 1	$\{t_4\}$	$\{t_4 + 1\}$	$\{\}$
t_6: ST $>b<$ t_5	$\{t_5\}$	$\{<b>\}$	$\{t_{11} + t_{12}\}$
t_7: LD $<x>$	$\{\}$	$\{<x>\}$	$\{\}$
t_8: MUL 2 t_7	$\{<x>\}$	$\{2, t_7\}$	$\{\}$
t_9: ST $>a<$ t_8	$\{t_8\}$	$\{<a>\}$	$\{t_2. t_3, t_4 + 1, t_{10} + 1, t_{11} + t_{12}\}$
t_{10}: LD $<a>$	$\{\}$	$\{<a>\}$	$\{\}$
t_{11}: ADD t_{10} 1	$\{<a>\}$	$\{t_{10} + 1\}$	$\{\}$
t_{12}: LD $<b>$	$\{\}$	$\{<b>\}$	$\{\}$
t_{13}: ADD t_{11} t_{12}	$\{t_{11}, <b>\}$	$\{t_{11} + t_{12}\}$	$\{\}$
t_{14}: ST $>c<$ t_{13}	$\{t_{11} + t_{12}\}$	$\{<c>\}$	$\{\}$

Kapitel 11: Optimierungen

0. Einbettung
1. Elimination gemeinsamer Teilausdrücke
2. Wertnumerierung
3. Weitere globale Optimierungen
4. Operatorvereinfachung
4. Nachoptimierung

11.2 Idee der Wertnumerierung

- Werte werden durch Operationen definiert und durch andere Operationen verwendet
- Werte, die öfter verwendet werden, sollten nur einmal berechnet werden
- Wie kann man algorithmisch die Gleichheit von Werten, die an unterschiedlichen Programmstellen definiert werden, herausfinden?
- Idee:
  - Basisfall: Konstanten
  - Induktion: Wenn Eingaben einer Operation und Operation selbst gleich sind, sind es auch die dadurch definierten Werte
- Probleme:
  - Alias: wohin werden Werte gespeichert?
  - Zusammenfluß im Ablauf: welcher Wert gilt?

11.2 Wertnumerierung

Ziel: Definiere Wertnummern für Operationen, so daß gleiche Wertnummern gleiche Werte implizieren

Typ: INT für ganzzahlige Konstanten; $BOOL$ für boolesche Konstanten etc.

Sonst: $\{v_1, \ldots, v_n\}$

Datenstruktur: Haschtabelle zur Zuordnung von Wertnummer zu Tupeln $\tau\ t\ t'$, wobei t, t' ebenfalls durch Wertnummer, d.h. durch Einträge in die Hashtabelle oder Konstanten, repräsentiert werden.

Achtung: eine Tabelle pro Basisblock!

11.2 Wertnumerierung mit lokalen Variablen ohne Alias Problem

1. initial: Wertnummer $wn(t) = ungültig$ für alle Tupel t; wn(Konstante) =Konstante.
2. Für alle Tupel t in aufsteigender Reihenfolge:

   (a) Sei $t = ST >local< t'$

   $wn(t) := wn(ST >local< wn(t'))$

   Wenn $wn(t) = ungültig$:

   $wn(LD <local>) := wn(t')$, $wn(t) :=$ neue Wertnummer,

   Generiere: $wn(t)$: $ST >local< wn(t')$

   (b) Sei $t = LD <local>$, $wn(t) := wn(LD <local>)$.

   Wenn $wn(t) =$ ungültig:

   $wn(t) :=$ neue Wertnummer, Generiere: $wn(t)$: t

   (c) Sei $t = \tau\ t'\ t''$

   $wn(t) := wn(\tau\ wn(t')\ wn(t''))$

   Wenn $wn(t) =$ ungültig:

   $wn(t) :=$ neue Wertnummer, Generiere: $wn(t)$: $\tau\ wn(t')\ wn(t'')$.

   (d) Sei $t = call\ \ proc\ t'\ t'' \ldots$ -- *analog* (c) *mit* $t = call\ \ proc$

11.2 Wertnumerierung mit globalen Variablen ohne Alias Problem

- Nichtrekursive Prozeduren:
  - Neue Behandlung für Fall (d) Sei $t = call\ \ proc\ t'\ t'' \ldots$
  - Prozedur muß so analysiert werden, als ob sie offen eingebaut wäre
- Rekursive Prozeduren:
  - Neue (zusätzliche) Behandlung für Fall

    (a) Sei $t = ST >global< t'$

    $wn(t) := wn(ST >global<$ Wertnummer (t')

    Wenn $wn(t) =$ ungültig:

    $wn(LD <global >) := wn(t')$,

    $wn(t) :=$ neue Wertnummer,

    Generiere immer: $wn(t)$: t.
  - Behandlung für Fall (d) wie bisher

    Aber wenn Wert (potentiell) $global$ umdefiniert in $proc$, so setze Wertnummer für Tupel $ST >global<$, $LD <global>$ und für davon transitiv abhängige Tupel auf ungültig

11.2 Allgemeine Wertnumerierung

(e) Sei $t = ST\ t'\ t''$
-- t' ist Adresse, deren Wert unbekannt ist (keine konstante Adresse)

$wn(t) := wn(ST\ wn(t')\ wn(t''))$

Wenn $wn(t)$ = ungültig:
- $wn(LD\ wn(t')) := wn(t')$,
- $wn(t) :=$ neue Wertnummer,

Generiere: $wn(t)$: t.

Wenn Wert (t') eventuell gleich Wert (tt):
- $wn(ST\ wn(t)\ ...) :=$ ungültig
- $wn(LD\ wn(t)) :=$ ungültig,
- Wertnummer transitiv abhängiger Tupel := ungültig

11.2 Zusammenfassung Wertnumerierung

(1) initial: Wertnummer wn(Konstante)=Konstante; $wn(t) = ungültig$ für andere Tupel t.
(2) Für alle Tupel t in aufsteigender Reihenfolge:
- (a) Sei $t = ST >a< t'$:
  $wn(t) := wn(ST >a< wn(t'))$,
  $wn(LD <a>) := wn(t')$,
  wenn $wn(ST >a< wn(t'))$ noch nicht definiert, generiere: $wn(t)$: t,
- (b) Sei $t = LD <a>$:
  $wn(t) := wn(LD <a>)$,
  wenn $wn(LD <a>)$ noch nicht definiert, generiere: $wn(t)$: t,
- (c) Sei $t = \tau\ t'\ t''$: -- analog bei einstelligen Operationen
  $wn(t) := wn(\tau\ wn(t')\ wn(t''))$,
  wenn $wn(\tau\ wn(t')\ wn(t''))$ noch nicht definiert, generiere: $wn(t)$: t
- (d) Sei $t = Prozeduraufruf(...)$:
  $wn(t) := wn(Aufruf(wn(...)))$
  wenn $wn(Aufruf(wn(...)))$ noch nicht definiert, dann generiere: $wn(t)$: t und setze alle Wertnummern für Tupel ungültig, die durch Nebenwirkungen ihren Wert ändern können (im Zweifel: alle)

11.2 Hinweise zur Wertnumerierung

- Initial sind alle Wertnummern außer bei Konstanten ungültig: dies kann durch Kenntnisse über vorangehende Grundblöcke abgeschwächt werden.
- Jeder neue Eintrag in die Haschtabelle erzeugt eine neue Wertnummer. Nach Prozeduraufrufen t können Eintrage t' ungültig werden, $t' \in X(t)$.
- Im allgemeinen gilt nur $t \in D(t)$. Bei Speichere-Operationen gilt zusätzlich $t' \in D(t)$ für alle Ladeoperationen von der gleichen Adresse. Bei Prozeduraufrufen gilt zusätzlich $t' \in D(t)$ für alle potentiellen Ergebnisparameter.
- Eine Speichere-Operation $ST >a< t'$ macht alle $wn\ (LD <a'>)$ ungültig, für die nicht sicher ist, ob $a = a'$ oder $a \neq a'$ (Alias-Problem).
- Problem Ablaufsteuerung behandelt mit SSA

11.2 Beispiel für Wertnumerierung

Original	U	D	X	Resultat
$t_1: ST\ >a<\ 2$	$\{\}$	$\{<a>\}$	$\{t_2, t_3, t_4+1, t_{10}+1, t_{11}+t_{12}\}$	$v_1: ST\ >a<\ 2$
$t_2: LD\ <a>$		$\{<a>\}$	$\{\}$	
$t_3: LD\ <x>$		$\{<x>\}$	$\{\}$	$v_2: LD\ <x>$
$t_4: MUL\ t_2\ t_3$	$\{<a>, <x>\}$	$\{t_2, t_3\}$	$\{\}$	$v_3: MUL\ v_2\ 2$
$t_5: ADD\ t_4\ 1$	$\{t_4\}$	$\{t_4+1\}$	$\{\}$	$v_4: ADD\ v_3\ 1$
$t_6: ST\ >b<\ t_5$	$\{t_5\}$	$\{<b>\}$	$\{t_{11}+t_{12}\}$	$v_5: ST\ >b<\ v_4$
$t_7: LD\ <x>$	$\{\}$	$\{<x>\}$	$\{\}$	
$t_8: MUL\ 2\ t_7$	$\{<x>\}$	$\{2, t_7\}$	$\{\}$	
$t_9: ST\ >a<\ t_8$	$\{t_8\}$	$\{<a>\}$	$\{t_2, t_3, t_4+1, t_{10}+1, t_{11}+t_{12}\}$	$v_6: ST\ >a<\ v_3$
$t_{10}: LD\ <a>$	$\{\}$	$\{<a>\}$	$\{\}$	
$t_{11}: ADD\ t_{10}\ 1$	$\{<a>\}$	$\{t_{10}+1\}$	$\{\}$	
$t_{12}: LD\ <b>$	$\{\}$	$\{<b>\}$	$\{\}$	
$t_{13}: ADD\ t_{11}\ t_{12}$	$\{t_{11}, <b>\}$	$\{t_{11}+t_{12}\}$	$\{\}$	$v_7: ADD\ v_4\ v_4$
$t_{14}: ST\ >c<\ t_{13}$	$\{t_{11}+t_{12}\}$	$\{<c>\}$	$\{\}$	$v_8: ST\ >c<\ v_7$

Kapitel 11: Optimierungen

0. Einbettung
1. Elimination gemeinsamer Teilausdrücke
2. Wertnumerierung
3. Weitere globale Optimierungen
4. Operatorvereinfachung
4. Nachoptimierung

11.3 Weitere globale Optimierungen

- CSE - Elimination gemeinsamer Teilausdrücke: globale Wertnumerierung?
- Konstantenfaltung, partielle Auswertung
- toten Code entfernen (Code trägt nichts zum Ergebnis bei)
- nicht erreichbaren Code entfernen
- Schleifenoptimierung:
  - schleifeninvariante Berechnungen nach vorne/hinten aus Schleife herausschieben
  - Operatorvereinfachung (*strength reduction*): Multiplikationen durch Additionen ersetzen u. ä., vor allem Vereinfachung der Zugriffsfunktionen auf Reihungen
- PRE - Elimination partieller Redundanzen: Berechnungen an die Eingänge eines Bereichs des Flußdiagramms verschieben, in dem das Ergebnis gebraucht wird, statt es im Bereich mehrfach zu berechnen

11.3 Weitere globale Optimierungen II

Interprozedurale Optimierung:
- Aufrufkontext aufgerufener Prozeduren genauer berücksichtigen
- offener Einbau von Prozeduren

objektorientierte Optimierungen (nach Verweisanalyse)
- Reduktion der Anzahl indirekter Prozeduraufrufe
- Objekte begrenzter Lebensdauer von der Halde in den Keller verlegen

Lokalität ausnutzen
- Cacheoptimierung
- Plattenzugriffe verringern

Übersetzerbau II – Ausgewählte Kapitel

- Werkzeuge
  - Cocktail, BEG, ELI, SUIF, Trimaran, GCC
- Sequentielle Optimierungen
  - SSA Konstruktion
  - Optimierungen auf SSA-Form:
    Operatorvereinfachung, Eliminierung gemeinsamer Teilausdrücke (CSE), Eliminierung partieller Redundanzen (PRE)
  - Speicher SSA
  - Globale kontextsensitive Wertanalyse auf SSA-Form
- Cache Optimierung
  - Caches und ihre Problematik
  - Techniken zur Cacheoptimierung (und zur Parallelisierung): Schleifenoptimierungen für Reihungen, Optimierungen für dynamische Datenstrukturen, Vorladen und Befehlsanordnung
- Weitere Optimierungen
  Nachoptimierung, Registerzuteilung, Befehlsanordnung
- Nebenläufige Sprachen
  - Begriffe und Konzepte
  - Parallele Hardware-Architekturen
  - Implementierung von Parallelität