Ausgewählte Kapitel aus dem Übersetzerbau

Dr. Sabine Glesner
Fakultät für Informatik
Universität Karlsruhe

Sommersemester 2005

http://www.info.uni-karlsruhe.de/

Speicher SSA

Vorlesungsplan

- Einführung (1)
- Datenflußanalyse und monotone Frameworks (1-3)
- SSA-Darstellungen und sequentielle Optimierungen (4-7)
  - SSA Konstruktion (1. Foliensatz)
  - Optimierungen auf SSA-Form (2. Foliensatz)
    Operatorvereinfachung, Eliminierung gemeinsamer Teilausdrücke (CSE), Eliminierung partieller Redundanzen (PRE)
  - Speicher SSA
  - Globale kontextsensitive Wertanalyse auf SSA-Form
- Cache Optimierung (8-10)
  - Caches und ihre Problematik
  - Techniken zur Cacheoptimierung (und zur Parallelisierung): Schleifenoptimierungen für Reihungen, Optimierungen für dynamische Datenstrukturen, Vorladen und Befehlsanordnung
- Weitere Optimierungen (10)
  - Nachoptimierung, Registerzuteilung, Befehlsanordnung
- Nebenläufige Sprachen (11)
  - Begriffe und Konzepte
  - Parallele Hardware-Architekturen
  - Implementierung von Parallelität
- Werkzeuge (12): Cocktail, BEG, ELI, Suif, Trimaran, (PAG, Firm)
- Evtl. Übersetzerverifikation (12)

Inhalt – SSA

- Optimierungen auf SSA-Form
  - Operatorvereinfachung
  - Eliminierung gemeinsamer Teilausdrücke
  - Eliminierung partieller Redundanzen
- Speicher SSA
  - Haldenoperationen
  - Alias-Problematik
  - Points-To-Analyse
  - Namensschema
  - Iterative Analyse
- Globale kontextsensitive Wertanalyse
  - Interprozedurale Analyse
  - Nichtrealisierbare Pfade
  - χ-Terme
  - Symbolisches Rechnen mit χ-Termen
  - Optimistischer SSA Aufbau

Kommentarfolie: Laufzeitumgebungen (Wdhlg.)

- Datenrepräsentation: Basisdatentypen, Aufzählungstypen, Reihungen, Verbunde, Zeiger, Strings, Mengen, ...
- Speicherbenutzung:
  - statische Speicherorganisation: Anzahl und Größe aller Objekte zur Laufzeit bekannt
  - dynamische Speicherorganisation mit Keller (Stack): geschachtelte Umgebungen
  - dynamische Speicherorganisation mit Halde (Heap): bei Zeigern mit Aliasen, für anonyme Objekte, bei OO Sprachen
- Registerbenutzung: Information über Laufzeitumgebung in Registern, oft verwendete Variablen, temporäre Werte, ...
- Lokaler Kellerrahmen: Speicher lokal für einen Prozeduraufruf
- Laufzeitkeller: ein Kellerrahmen für jede aktive Prozedur
- Parameterübergabe-Mechanismen (Wert, Zeiger auf Speicher, ...)
- Prozeduraufrufe
- Literatur: Waite/Goos Kap. 3.3, Muchnick Kapitel 5

Haldenoperationen

- Bisher: SSA für lokale, alias-freie Variable
- Jetzt: Modellierung von
  - Reihungen (*arrays*)
  - Verbunden (*records*, *structs*, Objekten)
- Erweiterung von SSA
  - um Repräsentationen für Reihungen, Verbunde/Objekte,
  - um Operationen für den Zugriff,
  - um Modi für Repräsentationen
- Unterscheide zwei Fälle
  - Kellerobjekte mit möglichem Alias (Reihungen, Referenzparameter, . . .)
  - Haldenobjekte (Reihungen, Verbunde)
- Lösung des Haldenproblems löst auch das Kellerproblem

Haldenoperationen – Abbildung auf Hauptspeicher

- Strukturen, Verbunde, Objekte:

```
a: record A {
   T1 t1;
   T2 t2;
   T3 t3;
 }

a.t1 = a
a.t2 = a + sizeof (T1)
a.t3 = a + sizeof (T1) + sizeof (T2)
```

- Reihungen:

```
a: array of <T>, d := sizeof (T)

a [0] = a
a [i] = a + i * d
```

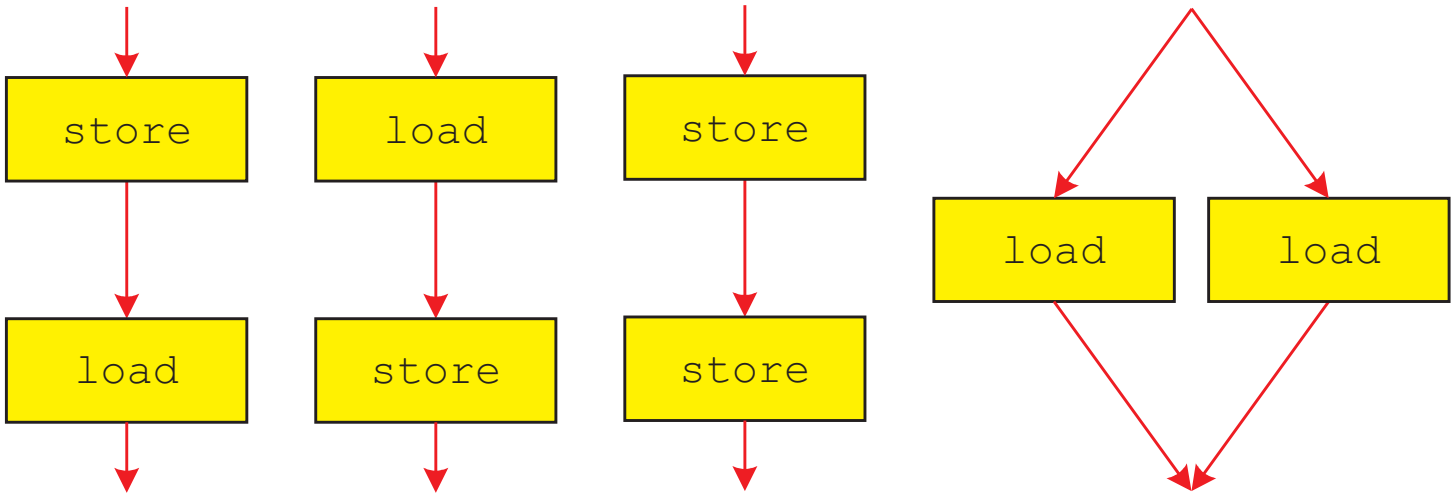
Alias-Problematik

- Problem: Namen bezeichnen Zugriffspfade, keine (absoluten) Adressen
- Definition „Alias“:
  - unterschiedliche (Quelltext-)Namen, aber
  - gleiche (physikalische) Ressource
    - potentiell: Kann-Alias (*may-alias*)
    - sicher: Muß-Alias (*must-alias*)
- Ursache:
  - Reihungen (`a[i]`, `a[j]`)
  - Referenzparameter
  - Zeiger, Referenzen, *address-of*-Operator `&`
  - *Call-by-Name*
- Folge:
  - Zugriffe auf Reihungen/Verbunde sind geordnet; Ordnung *nicht* mit bisherigem SSA-Aufbau darstellbar
  - Pessimistischer Ansatz: Totale Ordnung von Zugriffen auf Verbunde

Abhängigkeiten

- Klassifikation von Abhängigkeiten zwischen Zugriffen (Kann- oder Muß-Aliase):
  - Datenabhängigkeit, *Read-after-Write, RaW*
  - Antiabhängigkeit, *Write-after-Read, WaR*
  - Ausgabeabhängigkeit, *Write-after-Write, WaW*

  Formell existiert noch die
  - Eingabeabhängigkeit *Read-after-Read, RaR*



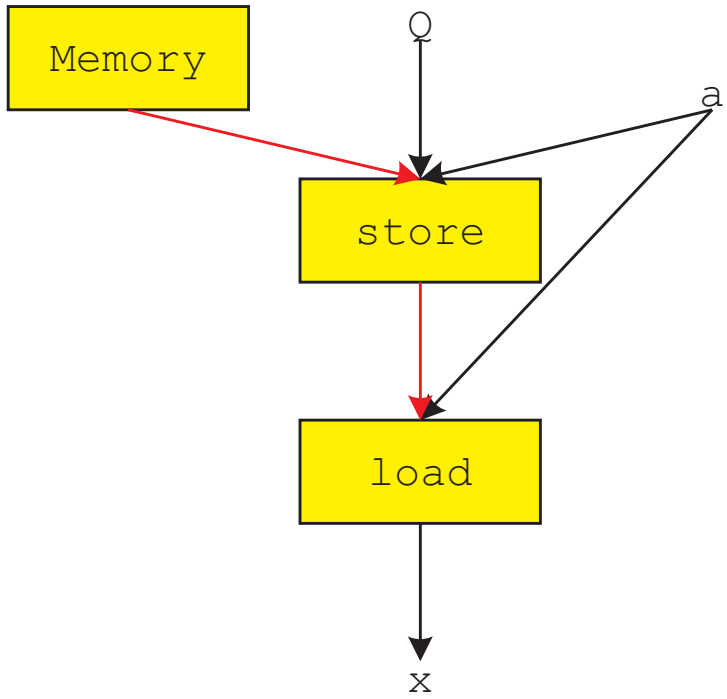
Haldenoperationen

- Idee:
  - Modelliere Hauptspeicher (Halde) als abstrakte Variable *memory*
    Speicherzugriffe lesen/schreiben *memory*
  - Bei Zusammenführungen des Steuerflußes:
    Setze Φ-Funktionen für *memory* ein (wie bisher!)
- Modi für Haldenoperationen:
  - Modus *p* (*Pointer*) für Referenzen, Deskriptoren
  - Modus *M* (*Memory*) für den Zustand des Speichers
- Definition von Haldenoperationen:
  - `Alloc` $(M) \rightarrow (M', p)$ liefert einen Deskriptor p auf ein neu angelegtes Objekt (Reihung, . . .)
  - `Load` $(M, p) \rightarrow (M', v)$ liest den Wert v zu Deskriptor p
  - `Store` $(M, p, v) \rightarrow M'$ schreibt Wert v an Stelle p.

Beispiel: SSA-Aufbau für Speicherzugriffe

```
foo (A a, A b)
{
  int x;
  a.x = Q;

  x = a.x;
}
```



Analyse von Haldenoperationen

- In „normaler“ SSA:
  - Wertanalyse auf SSA-Ecken
  - Operanden von Ecken eindeutig: Unmittelbare Vorgänger
  - *Wert* von Ecken *nicht* eindeutig (Φ-Funktion!)
  - neu: Abstraktion von Werten durch abstrakte Bereiche $\mathcal{V}$
- Für Haldenoperationen:
  - Wert einer `Load`-Operation und
  - Effekt einer `Store`-Operation

  auch durch *mittelbare* Vorgänger beeinflußt
- Haldenanalyse
  - finde geeignete Abstraktion für Zustand des Speichers
  - Modellierung der Effekte von Haldenoperationen auf den abstrakten Zustand

Haldenanalyse

- Analysen zur Alias-Problematik:
  - Alias-Analyse:
    Welche Namen (im Quellcode) können (zur Laufzeit) Aliase sein?
    Datenstruktur: Alias-Paare zwischen Quelltext-Namen
  - Verweisanalyse (*points-to-analysis*):
    Welche Referenz zeigt auf welches Objekt?
    Datenstruktur: Abstrakte Zuordnung von Referenzen zu Objekten
- Bewertung:
  - Alias-Paare sind per definitionem symmetrisch und transitiv,
    oft zu konservative Abschätzung
  - Verweisanalyse erfordert mehr Modellierungs-Aufwand (Beispiel: Referenz-Parameter),
    kann genauere Information liefern,
    per definitionem gerichtet, deshalb in der Praxis besser geeignet

Namensschema

- Ziel: Abstraktion über Objekte im Speicher
- Sei A die Menge aller *zur Laufzeit* auftretenden Referenzen a
- Namensschema *NS* ist eine statische Abbildung von A auf eine endliche Anzahl von Namen NS_A
- Abbildung $NS : A \to \mathfrak{P}(NS_A)$
  grundlegende Eigenschaft: $NS(a) \neq NS(b) \Rightarrow a \neq b$
  Beachte:
  - *NS* wird zur *Übersetzungszeit* bestimmt
  - a und b sind Werte zur *Laufzeit*
- Abstrakte Werte für Programmanalyse:
  - $V_p \subseteq \mathfrak{P}(NS_A)$: Abstraktion für Referenzen
  - V_t: Abstraktion für andere Werte (`int`, `float`, etc.)

Namensschema - Beispiele

- $NS_A = \{n\}; NS(p) := n$
  Alle Objekte werden auf denselben Namen abgebildet
- $NS_A = \{l_i, n\}$, $i \in \{1, \ldots, m\}$: Dabei steht
  - l_i für Objekte, die innerhalb der Prozedur erzeugt werden
  - n für alle anderen Objekte (Prozedur-Argumente, Ergebnisse von Load-Operationen)

  Definition von *NS*:
  - $NS(p) := l_i$, wenn p Resultat der i-ten Allokation ist
  - $NS(p) := n$, sonst
- $NS_A = \{l_T\}$, mit T: Typen („schnelle Typ-Analyse“, *rapid type analysis*)
  - $NS(p) := l_{\mathtt{T}}$, wenn `T` der deklarierte Typ von `p` ist.
  - Verschiedene Typen $\Rightarrow$ verschiedene Referenzen
  - In OO-Sprachen Unterklassen-Relation beachten!

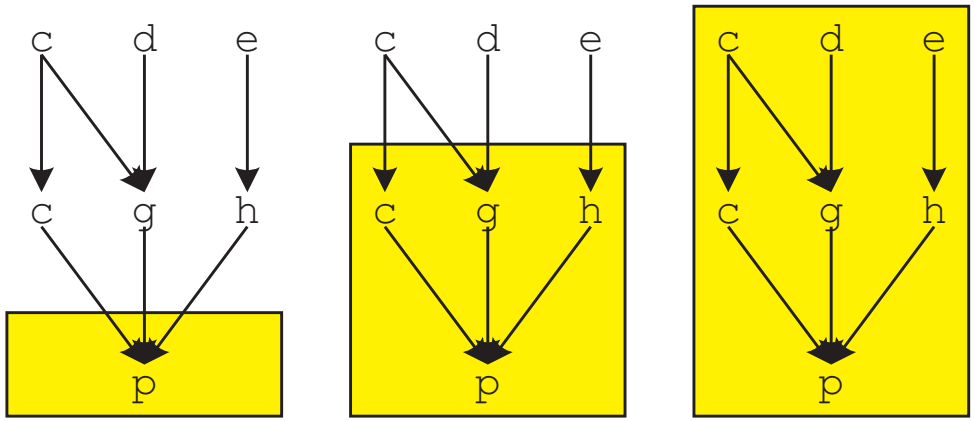
Verfeinerung des Namensschemas

- Wahl des Namensschemas:
  - Schema zu grob: Analyse und Optimierung zu konservativ, für kleine Programme werden Ressourcen nicht ausgeschöpft
  - Schema zu fein: Analyse größerer Programme scheitert am hohen Ressourcenbedarf der Analyse
- Definition *Verfeinerung* $\gg$ eines Namensschemas:
  $NS' \gg NS$ gdw. $NS\,(a) \neq NS\,(b) \Rightarrow NS'\,(a) \neq NS'\,(b)$ und $NS' \neq NS$
- Definition *Folge von Namensschemata*:
  Folge $\{NS_i\}$ für $i \in \mathbb{N}$, wobei $NS_j \gg NS_i$ für $j > i$
- Umsetzung:
  - Gegeben: Folge NS_i
  - Starte Analyse mit NS_1, Erweitere Analyse von NS_i auf NS_{i+1}, bis Ressourcen erschöpft

Verfeinerung des Namensschemas - Durchführung

- Initiales Namensschema, z.B. mit $NS_A = \{n\}; NS(p) := n$
- Erweiterung um
  - Typen
  - Statische Allokationen

  Kombination in OO-Sprachen sinnvoll
- Sei Δ der Aufrufkontext der zu analysierenden Prozedur p:
  Setze $NS_{i+1} := NS_i \times \Delta$
- Beispiel: Prozeduren c, d, e, f, g, h, p



Analyse auf Speicher-SSA

- Ansatz des *functional store* (Steensgaard 95):
  - Sei V_p eine Abstraktion für Adressen: $V_p = \mathfrak{P}(NS_A)$,
  - Sei V_t eine Abstraktion für andere Datentypen (`int`, `float` etc.).
  - Speicher(kanten) definieren Zuordnung $M : V_p \rightarrow V_p \cup V_t$
  - Zur rekursiven Definition von M:
    Abstrahiere Effekte der Haldenoperationen auf M für `Alloc`, `Load` und `Store`

Analyse auf Speicher-SSA

- Zuordnung $M \to M'$ für $Alloc(T)$:
  - Nur Erweiterung des Namensraums
  - Neu alloziertes Objekt ist nicht initialisiert ($\bot$):
    $M'(p) = \bot$,
    $\forall p' \neq p$: $M'(p') = M(p')$
    ($M(p)$ war nicht definiert!)
- Zuordnung $M \to M'$ für $Load(M, p)$:
  - Auslesen der Zuordnung:
    $v = \bigcup_{\rho \in p} M(\rho)$
    $M' = M$

Analyse auf Speicher-SSA

- Zuordnung $M \rightarrow M'$ für $Store(p, v, M)$:
  - Starke/Schwache Aktualisierung von M

    Wenn p eindeutig ist:

    $M'(p) = v$,

    $M'(p') = M(p') \; \forall p' \neq p$

    Alter Wert an p wird überschrieben

    Wenn p nicht eindeutig ist (z.B. $p = \phi(p_1, p_2)$):

    $M'(p) = v \cup M(p)$,

    $M'(p') = M(p') \; \forall p' \neq p$

Iterative Analyse

Analyse des Speicherzustandes:

- Führe Haldenanalyse durch mit initialem Namensschema NS_0
- Verfeinere Zuordnungen $M \rightarrow M'$ für Speicherkanten
- Wenn Verfeinerung nicht mehr möglich ist,
  - erweitere aktuelles Namensschema NS_i auf NS_{i+1}
  - starte Analyse erneut
- Fortsetzen, bis
  - Fixpunkt erreicht (keine weitere Verfeinerung möglich), oder
  - bis Ressourcen erschöpft

Zusammenfassung

- Speicher-SSA: Ausweitung von SSA auf Speicher-Operationen
- Analyse auf Speicher-SSA: Ausweitung von Wertanalysen auf Speicher-Operationen
- Gegenseitige Abhängigkeit

  Werte von Referenzen $\longleftrightarrow$ Speicher-Analyse
- Für Speicher-SSA zusätzlich nötig:
  - Namensschemata
  - schwache Aktualisierung
  - Iterative Analyse
- Problem: Haldenanalyse kostet extrem viel Speicher, schwierig für große Programme