Kapitel 19

Analyseverfahren

2 Ziele:

- Optimierung, insbesondere dynamic dispatch → statischer Prozeduraufruf und Platzreduktion
- Programmverstehen (Reengineering, Jahr 2000)

Wir behandeln hier:

1. Rapid Type Analysis
2. Points-to Analyse
3. Snelting/Tip Analyse

19.1 Rapid Type Analysis

einfache, schnelle, aber wirkungsvolle Methode

Ziel: Auflösen dynamischer Bindung, Entfernung toter Members/Methoden

Idee: wenn nie ein C-Objekt erzeugt wird, kann auch nie eine C-Funktion aufgerufen werden

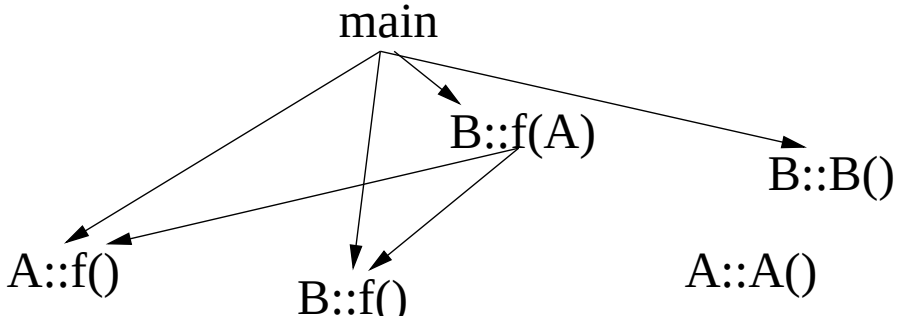
Schritt 1: Aufbau des Call-Graphen

- Knoten: alle (virtuellen) Funktionen $C :: m(x)$ nebst Signatur incl. $main$, Konstruktorfunktionen $C :: C()$
- Kanten: von $C :: f()$ nach $D :: g()$, wenn $C :: f()$ Aufruf $d \to g()$ enthält
- wg. dynamischer Bindung müssen *alle* potentiellen Ziele des Aufrufs $d \to g()$ berücksichtigt werden, also auch alle $E :: g()$ für $E : D$

Beispiel:

```
class A {
  int f() { return 42; };
};
class B : A {
  int f() { return 17; };
  int f(A x) { return x->f(); }
}
void main {
  B* p = new B;
  int r1 = p->f(p);
  int r2 = p->f();
  A* q = p;
  int r3 = q->f();
}
```

Aufrufgraph:



Schritt 2: Elimination von Methoden, die garantiert nicht benutzt werden

1. Markiere alle Kanten zu virtuellen Funktionen als “verboten“, alle Kanten von „main“ zu Konstruktorfunktionen als „erlaubt“
2. Traversiere den Graphen, ausgehend von $main$ entlang erlaubter Kanten; markiere erreichbare Knoten
3. wenn man auf Aufruf einer Konstruktorfunktion $C :: C()$ stößt, markiere alle Kanten von erreichbaren Knoten zu Funktionen in C’s Vtable (!) wieder als erlaubt
4. wiederhole, bis keine erlaubten Kanten mehr hinzukommen
5. Funktionen, zu denen kein erlaubter Pfad führt, werden garantiert nie aufgerufen ⇒ raus aus dem Code; evtl. Ersetzen von dynamischen Methodenaufrufen durch statische

No. 3 ist eine typische *konservative Approximation*: „wenn ein C-Objekt erzeugt werden kann, so können auch seine Methoden aufgerufen werden“ bzw „C-Methode wird nur dann nicht aufgerufen, wenn nie C-Objekt erzeugt wird“ ⇒ es wird nie ein möglicher Aufruf vergessen, aber nicht allzu viele Aufrufe werden fälschlich als möglich angenommen

Im Beispiel: Kante $\mathrm{main} \longrightarrow A :: f()$ wird nie erlaubt, da nie ein A-Objekt erzeugt wird

$\Rightarrow$ Aufruf $q \rightarrow f()$ kann statisch zu $q.B :: f()$ aufgelöst werden

Verbesserung: Graph enthält auch Data Members $\Rightarrow$ Entfernen toter Members, Objekte werden kleiner

empirische Studien (IBM):

- Analysegeschwindigkeit: 4000 LOC/sec auf PC
- in typischen C++-Programmen werden ca. 20% der Members von RTA als tot nachgewiesen
- Reduktion der Größe von Java Class Files: zwischen 40% und 80%!
- Grund: oft werden Bibliotheken benutzt, deren Funktionalität nur zum kleinen Teil wirklich verwendet wird

Aber: RTA ist ein ziemlich ungenaues Verfahren.

19.2 RTA als Constraint-Problem

Wie Palsberg/Schwartzbach, lässt sich RTA als Mengengleichungssystem schreiben

2 Mengen: R Menge der „lebendigen“ Members, S Menge der „lebendigen“ Klassen.

Initialisierung: $R = \{\text{main}\}$, $S = \{\text{Main}\}$

Inferenzregeln zur Vergrößerung von R und S:

1.

$$\frac{M \in R \qquad \text{new } C \in \text{body}(M)}{C \in S}$$

2.

$$\frac{\begin{array}{c} M \in R \qquad e.m(x) \in \text{body}(M) \\ C \leq \text{type}(e) \qquad C \in S \qquad \text{lookup}(C, m) = M' \end{array}}{M' \in R}$$

Es gibt eine ganzes Spektrum von Analysen, die sich durch ähnliche Constraintsysteme beschreiben lassen (Tip/Palsberg, OOPSLA 2000)

19.3 Points-to Analyse

- Ziel: bestimme für jeden Pointer p, auf welche Objekte er zeigen könnte (*Points-to Set*)

$$PT(p) = \{o_1, o_2, \ldots, o_k\}$$

- damit $PT(p)$ immer endlich bleibt, werden Objekte, die durch dieselbe new-Anweisung erzeugt werden, nicht unterschieden

  $\Rightarrow$ 1 Objektrepräsentant / new, auch wenn das new zB in Schleife ausgeführt wird („Trick von B. Ryder“ $\rightsquigarrow$ konservative Approximation!)

- exakte Points-to Analyse ist sogar mit Repräsentanten-Trick unentscheidbar (Reduktion aufs Postsche Korrespondenzproblem [Ramalingam 1995])

$\Rightarrow$ Näherungsverfahren. Grundprinzip: *konservative Approximation.*
$PT(p)$ darf zu groß sein, aber *niemals zu klein*

- weiteres Ziel: *Präzision.* $PT(p)$ soll zwar nicht zu klein sein, aber *möglichst klein*

- Verfahren:
  - Steensgard: relativ ungenau, aber schnell: $O(n \cdot \alpha(n)) \approx O(n)$
  - Anderssen: relativ genau, aber teuer: $O(n^3)$
  - Lhotak et al (2003)/ Whaley et al (2004): verwendet BDDs und Datalog $\Rightarrow$ Andersen-Präzision wird auch für große Programme nutzbar
- normalerweise *flußunabhängige* Analyse: 1 $PT(p)$ pro Programm, nicht etwa eins pro Anweisung
- *interprozedurale* Analyse $\rightsquigarrow$ Simulation von Parameterübergabe / Return-Value durch zusätzliche Zuweisungen
- *kontextsensitive* Analyse: verschiedene Aufrufkontexte von Prozeduren werden unterschieden

Anderssen-Verfahren

zentrale Datenstruktur: *Points-to Graph G*

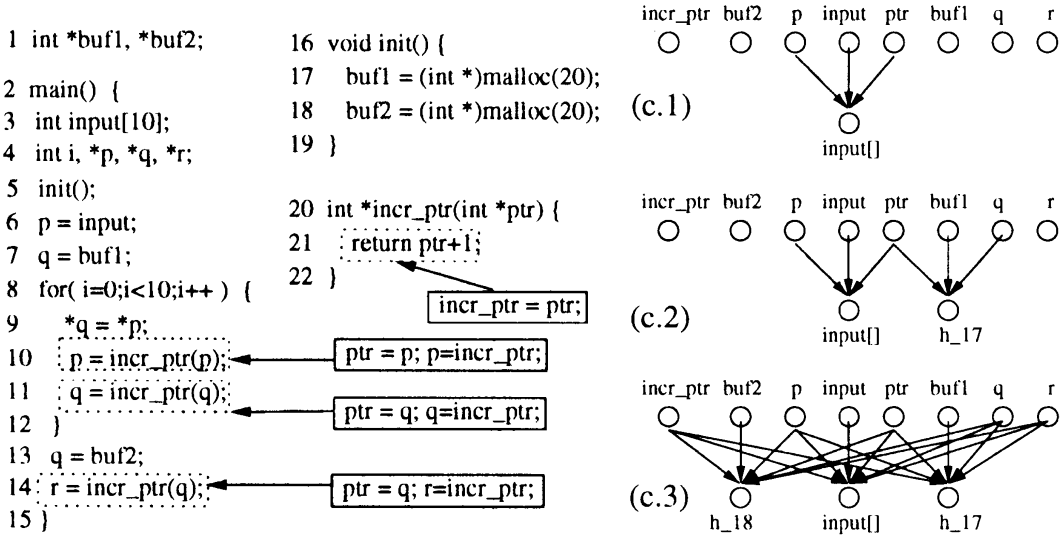
Knoten: Pointer und Objektrepräsentanten;
1 Repräsentant pro **new**-Anweisung

Kante $p \rightarrow o$, falls p auf o zeigen könnte. Pointer können auch auf Pointer zeigen. $o \in PT(p) \Leftrightarrow p \rightarrow p' \rightarrow p'' \rightarrow \ldots \rightarrow o \in G$

Aufbau des Graphen:

1. Objekterzeugung: `p = new C(...);`
   Kreiere Repräsentanten o; ziehe Kante $p \rightarrow o$
2. Zuweisung: `p = q;` hat Effekt $PT(p) := PT(p) \cup PT(q)$
   $\Rightarrow$ ziehe Kante $p \rightarrow q$ bzw $p \rightarrow r$ für alle $r \in PT(q)$

Beispiel (C):



Steensgard-Verfahren

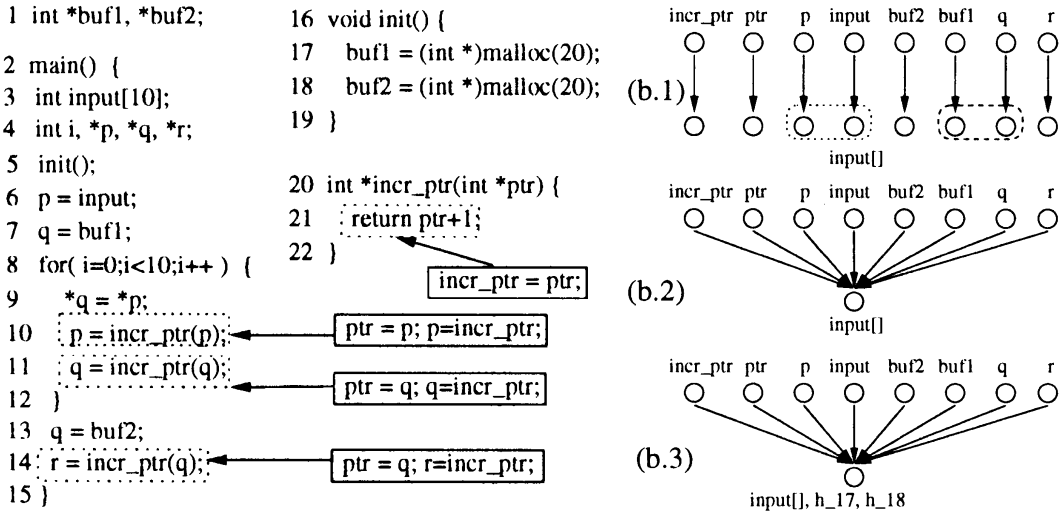
Idee: mache Graph ungenauer, aber kleiner;
für `p = q;` tue so, als ob auch `q = p;` vorhanden

$\Rightarrow$ *verschmelze* Knoten für $PT(p), PT(q)$

$\Rightarrow$ Äquivalenzklassenbildung; Union-Find Algorithmus

$\Rightarrow$ $O(n \cdot \alpha(n))$ mit α = inverse Ackermann-Funktion

Beispiel:



schnell! aber empirisch ca. 20% ungenauer

Algorithmen

Andersen-Pseudocode (für Sprachen ohne Vererbung/ dynamische Bindung):

```
for S_n ≡ x = malloc(...); do
  x → o_n ∈ G;
repeat
  for x = y; do
    for o ∈ PT(y) do
      x → o ∈ G;
until G unver"andert
```

Komplexität: $O(n^3)$ (n: Programmgröße)

Steensgard:

```
for S_n ≡ x = malloc(...); do
  x → o_n ∈ G;
for x = y; do
  union(PT(x),PT(y));
```

Komplexität: $O(n \cdot \alpha(n))$

19.4 Points-to für OO

Problem: Behandlung der dynamischen Bindung

muß statisch und konservativ approximiert werden

⇒ Static Lookup gekoppelt mit Fixpunktiteration

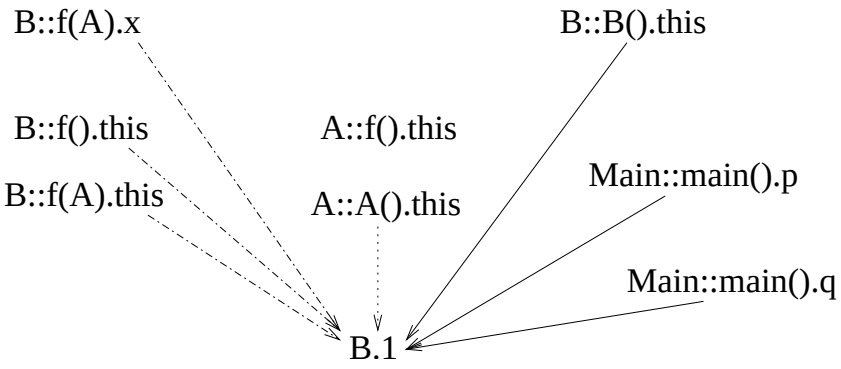
Algorithmus:

1. Konstruiere initialen Graphen aus Zuweisungen
2. für jeden Aufruf $o.m(x)$ tue:
   (a) Bestimme $PT(o) = \{o_1, o_2, \ldots, o_n\}$ sowie $PT(x) = \{x_1, \ldots, x_k\}$ aus aktuellem Graphen
   (b) statische Näherung der dynamischen Bindung: Verwende statischen Lookup für Auflösung der Aufrufe $o_i.m(x)$; diese seien $C_1 :: m(a_1), \ldots C_n :: m(a_n)$
   (c) Für jedes Paar a_i, x_j füge Kante $a_i \rightarrow x_j$ ein ($i = 1..n,\ j = 1..k$)
   (d) Für Rückkehrwert r_i füge ebenfalls entsprechende Kanten ein (s.o.)
3. Falls Graph sich geändert hat: GOTO 2

Beispiel: (vgl RTA)

```
class A { int f() { return 42; } };
class B extends  A {
  int f() { return 17; }
  int f(A x) { return x.f(); }
}
class Main {
  void main {
    B p = new B;
    int r1 = p.f(p);
    int r2 = p.f();
    A q = p;
    int r3 = q.f();}
}
```

Points-to-Graph: (durchgezogen: initiale Kanten)



→ this-Pointer von A::f() zeigt nie auf echtes Objekt!

allgemein: Points-to liefert wesentlich genauere Information als RTA

OO-Andersen als Constraint-System

Formulierung für OO-Sprachen:

1. `x=new C();` $\Rightarrow o_n \in PT(x)$
2. `x=y;` $\Rightarrow PT(y) \subseteq PT(x)$

   (Steensgard: $PT(x) = PT(y)$)
3. `y=x.m(z);`

$$\frac{o \in PT(x) \qquad \text{type}(o) = C \qquad \text{lookup}(C, m) = T\ m(U\ p)\{...\text{return}\ e...\}}{PT(z) \subseteq PT(p), PT(e) \subseteq PT(y)}$$

völlig analog zu Palsberg-Schwartzbach!

Tatsächlich ist

$$[\![p]\!] = \{type(o) \mid o \in PT(p)\}$$

Oberbegriff: 0-CFA (Shivers 91)

19.5 Snelting/Tip Analyse (KABA)

... Extrafolien ...