PaCo – Kap1

- Programmanalyse: Berechnung der Laufzeiteigenschaften aus dem Code
  ↳ unentscheidbares Problem, daher Approximation
- Compileroptimierung: äquivalente Programmtransformation mit Verbesserung der Laufzeiteigenschaften (bzgl. Laufzeit, Speicherverbrauch oder Codelänge)
  90:10-Regel: 90% der Zeit wird in 10% des Codes verbracht.
- hier: verschiedene Analysen durchführen und danach optimieren

PaCo – Kap2 (SLC)

- SLC/Straight Line Code: keine Schleifen/Verzweigungen (Sprünge/Prozeduren)
  ↳ nur Zuweisungen (entschachtelt, da hier Zwischencode betrachtet)
- Basisblock ≙ hier: iteratives Programm

LV (Live Variables Analyse): Rückwärtstransformation, je Zeile LV-Mengen bilden

$LV_{n+1} := \emptyset$ $\quad LV_n(n: x \leftarrow e) := (LV_{n+1} \setminus \{x\}) \cup \{V_e\}$ (falls $x \in LV_{n+1}$)

"linke Seite rauswerfen und rechte rein, falls linke Seite enthalten"

Dead Code Elimination: oben starten, Zuweisungen $i: x \leftarrow e$ streichen falls $x \notin LV_i$.
↳ Codeverkürzung / Idempotent

AE (Available Expression Analyse): Vorwärtstransformation, je Zeile AE-Mengen bilden

$AE_1 := \emptyset$ $\quad AE_{n+1} = AE_n \cup \{n\}$, falls $n: x \leftarrow e$ und $x \leftarrow e$ noch nicht in AE_n
$\setminus \{m\}$, falls $m: x' \leftarrow e'$ und $V_e \ni x$

"reinnehmen wenn keine Anweisung mit dieser rechten Seite bereits enthalten ist, alle rauswerfen, wo daß x alte kaputtmacht, aber wo x auf der rechten Seite vorkam"

Hilfsmengen je Zeile Belegen: alle Zeilen mit gleicher Rechter Seite sind in AE_i enthalten

Common Subexpression Elimination: jede Zeile i mit $wh(i) \neq \beta$: Anweisung auf temporäre Variable kopieren und in allen Zeilen $j \in wh(i)$ benutzen
↳ Verringerung der Rechenoperationen (evtl. längerer Code) / Idempotent
↳ starke Äquivalenz, da von der Interpretation unabhängig

RD (Reaching Definition Analyse): Vorwärtstransformation, je Zeile RD-Mengen bilden

$RD_1 := \emptyset$ $\quad RD_{n+1}(n+1: x \leftarrow e)$, falls x mit RD_n berechenbar (x, A) aufnehmen, sonst $(x, ?)$

Konstantenfaltung: Eingabeunabhängige Werte bestimmen und durchreichen, ggf. linke Seiten ausrechnen
↳ produziert Dead Code → weitere Optimierung möglich

gerichteter (Σ, V, A)-Graph $D = \langle K; lab; suc \rangle$ mit Knoten, Beschriftung und Nachfolge
↳ kann zyklisch sein

azyklisch: jeder Knoten repräsentiert einen Term, SLC als (Σ, V, A)-DAG darstellbar
↳ $D(\pi)$

$\underline{val}: V_\pi \times \mathbb{N} \to k$ hier: $\underline{val}(x, i) = k$ (k ist der Knoten des Werts von x nach i Rechenschritten repräsentiert)

beim Aufbau verschiedene Knoten für verschiedene Ausdrücke → CS-Elimination

Berechnung alles konstanten Ausdrücke (kein Knoten repräsentiert konst. Berechnung) → konst Faltung

nur Knoten betrachten die zu Ausgabevariablen führen → dead Code Elimination

Variablen mit selben Wert teilen sich gleichen Knoten → Vermeidung von Kopieranweisungen

Iterativer Aufbau: - zunächst nur Knoten für gleiche Eingabevariable und alle Programmkonstanten

- dann Simulation des Programmablaufs: für jede Zeile i: $x \leftarrow e$ bilde

val (x, i) mit Zeiger auf Knoten der Wert von x repräsentiert.

dieser ist vorhanden wenn $e \in \{A, V\}$, falls Berechnung mit konstantem Ergebnis Zeiger dorthin (ggf. Knoten einfügen, auch bei Unentscheidbarkeit, wie floats).

sonst Knoten einfügen mit entsprechenden Nachfolgern und darauf verweisen.

Eigenschaften: jeder Knoten repräsentiert verschiedenen Term (CS-Elimination) und zwar keinen Konstantenterm (Konstantenfaltung), es müssen nur Ausgaberelevante Knoten (die zu Ausgabevariablen führen) betrachtet werden (Dead Code Elimination)

Optimierung von SLC mit $D(\pi)$: Bestimmung des ausgaberelevanten Operationsknoten, diesen Namen geben ($h_0 - h_n$) und Anweisungen erzeugen: $h_0 \leftarrow x+y$, etc.

Schließlich bestimmen welche dieser Knoten die Ausgabevariablen repräsentiert und entsprechend umbenennen (z.B. $h_2 \hat{=} u$, dann alle h_2 durch u ersetzen), hier allerdings können gleiche Werte auftreten, deshalb entsprechende Kopieranweisungen einfügen (z.B. $h_2 \hat{=} u, v$, dann alle h_2 durch u ersetzen und am Ende $v \leftarrow u$ einfügen)

Algebraische Transformationen lokal ohne globale Analyse durchführen

Dead Code / Common Subexpression / Copy propagation sind stark äquivalent (strukturell verändern) und kommen ohne Interpretation eines Ausdrucks aus.
nicht jedoch die Konstantenfaltung (klar)

hier: Betrachtung von Kommutativität / Distributivität / Assoziativität und neutrale Elemente.

Bei Peephole-Optimierung Fenster über Code schieben und entsprechend vereinfachen

Bei Reduction in Strength teure Operationen durch billige ersetzen $x^2 \hat{=} x*x$ / $x*2 \hat{=} x+x$

Optimalität: $c: SLC \to \mathbb{N}$ Bewertungsfunktion $\pi \sim \pi'$ und $c(\pi) \leq c(\pi') \Rightarrow \pi$ c-optimal

es exist. c-optimales Programm für jedes SLC-Programm (wegen Mächtigkeit)

kann berechnet werden, wenn c-berechenbar, äquivalenz berechenbar und Aufzählverfahren exist.

i.A. nicht entscheidbar, für Polynome bzw. Horner-Code, der aber auch nicht immer optimal (bzw. für $x^{32} + x^{16} + x^8 + x^4 \ldots$, wo größere Potenz durch Aufteilen in niedrigere Potenz mit sich selbst berechnet werden kann)

PaCo-Kap 3 (Ordnungstheorie)

Halbordnung: Für Menge D und Halbordnungsrelation $\leq$ gilt
$\mathcal{D}=\langle D;\leq\rangle$

- reflexivität : $a \leq a$
- transitivität : $a \leq b,\ b \leq c \Rightarrow a \leq c$
- antisymmetrie : $a \leq b,\ b \leq a \Rightarrow a = b$

Schranken: betrachte Teilmenge $T \subseteq D$, für ein Element $a \in D$ welches

$\leq$ alle Elemente in T : untere Schranke
$\geq$ alle Elemente : obere Schranke
} mehrere Elemente $a \in D$ können es.

wenn a in der Teilmenge liegt : kleinstes / größtes Element ($\bot$ / $\top$) ist eindeutig bestimmt (Antisymm.)

$\leq$ alle Elemente und größtes a mit dieser Eigenschaft: größte untere Schranke
$\bigsqcap T$: Infimum

$\geq$ alle Element und kleinstes a mit dieser Eigenschaft: kleinste untere Schranke
$\bigsqcup T$: Supremum

Ketten: Funktion σ, die jedem $i \in \mathbb{N}$ ein Element aus D zuordnet, heißt

aufsteigende ω-Kette $\Leftrightarrow \sigma(i) \leq \sigma(i+1)$ $\quad \forall i \in \mathbb{N}$
absteigende ω-Kette $\Leftrightarrow \sigma(i) \geq \sigma(i+1)$

Eine Halbordnung heißt <u>vollständig</u> $\Leftrightarrow$ es es. kleinstes Element und jede ω-Kette hat kleinste obere Schranke

$\mathcal{D}$ ist ein vollständiger <u>Verband</u> $\Leftrightarrow$ für alle $T \subseteq D$ es. Supremum und Infimum

Operationen: das Produkt zweier Halbordnungen ergibt wieder eine Halbordnung (Elemente komponentenweise vergleichen)
gilt wegen Funktionenraum (Halbordnung mit Menge verknüpft und Funktionen die Elemente aus M auf D abbilden, $f \leq g \Leftrightarrow f(m) \leq f(g)$) auch für vollständige Halbordnungen / Verbände

Betrachte für zwei Halbordnungen Funktion $f : D_1 \to D_2$, dann ist f

<u>monoton</u> $\Leftrightarrow$ $a \leq_1 b \Rightarrow f(a) \leq_2 f(b)$ [Abbildung erhält Ordnung]

falls weiterhin gilt: Halbordnung vollständig und für aufsteigende Ketten gilt:

$f\left(\bigsqcup (a_i \mid i \in \mathbb{N})\right) = \bigsqcup (f(a_i) \mid i \in \mathbb{N})$ $\Leftrightarrow$ <u>Stetig</u> [Abbildung erhält Supremum]

Für f monoton und Halbordnung endlich $\Rightarrow$ f ist auch stetig

Fixpunktsatz 1: vollst. Halbordnung und stetige Funktion $\Rightarrow$ kleinster Fixpunkt es., berechenbar

Fixpunktsatz 2: vollst. Verband und monotone Funktion $\Rightarrow$ kleinster / größter Fixpunkt es.

Prüfungsfragen Beispiele:

- in Verband cpo. wegen Monotonie ein FP (2. Satz) der wegen acc (→ also stetig) und 1. Satz iterativ berechenbar ist.
- Kalbrechnung zur Definition der Fortsetzungsfunktionen benötigt ⇒ Modellierung Programmsemantik
- acc ⇒ jede ω-kett. endl. viele Berechnungsschritte stationär
- MOP ≙ MFP bei Distributivität

PaCo-Kapt (IC)

IC ≙ SLC mit bedingten und unbedingten Sprüngen (benutze Label + Prädikate)

mind. eine Ausgabevariable / beliebig viele Eingabe + lokale Variablen / alle verschied

π ist standardmarkiert $i = 1 \ldots q$, $q+1$ Stopmarke

Zustandsraum $Z := \{\sigma \mid \sigma : V_\pi \to A\}$ mit $\sigma_i : Z \to Z$, $i = 1..q+1$

Fixpunkt-/Fortsetzungssemantik, für π Programmvariablen V_π als Gleichungssystem mit $\bar{a} = (a_1, \ldots, a_s) \in A^s$: E_π

Eingabevariablen $\bar{x} = (x_1, \ldots, x_n) \in A^n$
Programmvariablen $\bar{v} = (v_1, \ldots, v_s) \in A^s$
Ausgabevariablen $\bar{y} = (y_1, \ldots, y_m) \in A^m$

$\nabla = (\bar{x}, \underbrace{a, \ldots a}_{s})$

π durch Funktionsgleichungen darstellbar: $F(\bar{x}) = F_1(\bar{\nabla})$ ($A^n \to A^m$) (Variablen $\notin \bar{x}$ beliebig)

↳ diese sind Endrekursiv, also Funktionsaufrufe nur in Termpositionen, kein Rücksprungkeller notwendig.

$F_i(\nabla) = J_i : A^s \to A^n$, $i = 1 \ldots q$
$F_{q+1}(\nabla) = \nabla \quad A^s \to A^s$

Der Lösungsraum ist das Kreuzprodukt der Gleichungen, also

$$FR_\pi := PF(A^n, A^m) \times PF(A^s, A^s)^{q+1}$$

Dieser Funktionsraum ist vollst. Halbordnung / Verband

T_π die Funktion $FR_\pi \to FR_\pi$ (stetig)

Der Fixpunkt von T_π entspricht Lösung von E_π

↳ der kleinste Fixpunkt ex. wegen Verbandseigenschaft und Stetigkeit

Die erste Komponente ($\in PF(A^n, A^m)$) wird als Fixpunktsemantik bezeichnet

Bei IC wegen Selbsten äquivalenz zweier Programme nicht entscheidbar ⇒ keine optimalen Programme. Programmpfade regulär / Berechnungspfade unentscheidbar

IC als Basisblöcke: $i=1$ ist Blockanfang, Sprungziel ist Blockanfang, Vorgänger eines Sprung

Flußdiagramm: Abfolge der Anweisungen mit Prädikaten

Flußgraph: Basisblöcke und Stop-Knoten als Knoten, Kanten wie Flußdiagramm weglassen der Prädikate → betrachte Programmpfade. Hier: Kanten mit Label

Bei Konstantenfaltung: führe RD-Analyse durch und bestimme für jede Zeile konst. Variable

↳ für Berechnungspfade i.A. unentscheidbar: sonst Halteproblem lösbar

↳ Eingabe x: $0 \to \pi$ ausführen, Ausgabe $y=1$
$1 \to$ Ausgabe $y=0$

falls Konstanteneigenschaft entscheidbar und $y=0 \Rightarrow \pi$ divergiert ⇒ Widerspruch

Als Approximation (schwächere Sicht) betrachte die Programmpfade in zweistufiger Berechnung: globale Analyse: Bestimmung konstanter Var. am Blockanfang
lokale Analyse: Information innerhalb Basisblock best.

SLC: $A := [\delta \mid \delta : V_{ff} \to A \cup \{\top\}]$ $\Longleftrightarrow$ LC: $D := [\delta \mid \delta : V_{ff}^r \to \mathbb{Z} \cup \{\bot, \top\}]$

hier: nur relevante Variablen (rechte Seite oder Ausdruck in Bedingung): V_{ff}^r

Verband $\hat{\mathbb{Z}} := \mathbb{Z} \cup \{\bot, \top\}$ und $\leq$: $\ldots -2\ -1\ \underset{\bot}{\overset{\top}{0}}\ 1\ 2 \ldots$

$D := \hat{\mathbb{Z}}^{|V_{ff}^r|}$ $\quad (\perp, 4, 5, 6) \sqcup (1, 4, 6, \top) = (\top, 4, \top, \top)$

$RD_i \in D$ sind die konstanten Information am Blockanfang von i
$t_i : D \to D$ Transformation durch Block i

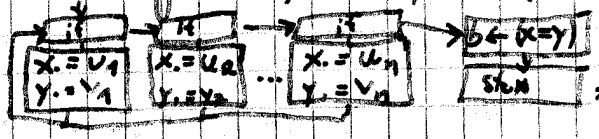
die Pfadmenge $Pfad[1,2]$ beschreibt die Menge alle Pfade die zu einem Vorgänger von i führen. $\Longrightarrow$ i.A. unendlich viele

$RD_i\langle p \rangle$ ist Konstanteninformation auf einem bestimmten Pfad p, die durch Join der einzelnen t_j mit $j \in p$ entsteht.

$RD_1 = \top^{|V_{ff}^r|}$ (Startinformation mit Top initialisiert, wegen Eingabevariablen)
↳ natürlich können mit $\perp$ init.

MOP-Information ist das der Join über alle möglichen Pfade: $RD_i^{MOP} := \bigsqcup \{RD_i\langle p \rangle \mid p \in Pfad[1,i]\}$

↳ i.A. unentscheidbar wegen MFP

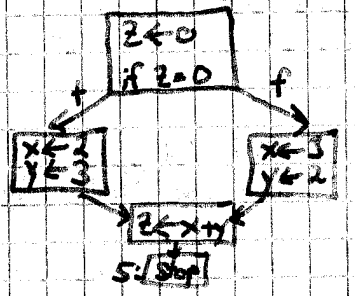


(wenn an (*) bekannt, dann MFP lösbar
→ Widerspruch)

MFP-Informationen: betrachte RD-Information als Join über die Transformation der Vorgänger RDs. Allgemeine Transformation $T : D^{|V_{ff}^r|} \to D^{|V_{ff}^r|}$

wenn T monoton (gilt im Beispiel wegen Abklitten) und acc (gegeben wegen $\hat{\mathbb{Z}}$)
dann T auch stetig → Fixpunkt ex. und in endlich vielen Schritten berechenbar
↳ erhalte MFP RD_i^{MFP} → damit lokale Analyse je Basisblock, ggf. Sprungoptimierung

Informationen werden immer Schwächer



Berechnungspfad: $RD_5 = (2, 3, 5)$

MOP: $RD_5^{MOP} = (\top, \top, 5)$

MFP: $RD_5^{MFP} = (\top, \top, \top)$

Alternative zu Basisblöcken: Single Instruction (SI) - Graph

betrachte statt Blöcken einzelne Anweisungen
↳ größere Gleichungssysteme, aber durch Hintereinander ausführen von Funktionen lösbar
↳ nur eine Analyse statt global + lokal

Datenflussanalyse (monotone Framework)

Führe allgemeine Analysen durch, darum:

Flussgraph und abstrakte Interpretation $\Longrightarrow$ DFA-System

Flussgraph: $G = (K_{GO}, K_{GK}, S)$ Menge von Knoten (Blöcke des Programms) und Kanten zwischen denen (Programmpfade). Definiere Vorgänger und Pfadmenge von einem Knoten zu seinen Vorgängern (unendlich viele)
↳ ggf. abwickeln

Forderung: ausgezeichneter Startknoten ohne Vorgänger, von dem jeder andere Knoten erreichbar ist.

abstrakte Interpretation: $J = (\mathcal{D}, \varphi, d_S)$ vollständiger Verband mit monotonen Abbildern φ (Information Transforme) für jeden Knoten aus K_q

DFA-System $\triangleq$ monotone Framework und bestimmt MOP/MFP-Lösung
$\Delta = (J, G)$

MOP-Lösung: Für jeden Knoten betrachte Analyseinformation AI_k^{MOP}, mit $AI_S^{MOP} = d_S$

AI für einen Pfad p ist die hintereinander Ausführung der auf dem Pfad liegenden Knoten beginnend bei der Starkinformation d_S.

AI_k^{MOP} ist der Join über alle $AI\langle p\rangle$ der Pfade p die vom Startknoten zum Vorgänger von k führen, also $p \in Pfad(S, k)$

MFP-Lösung: Gleichungssystem E_J aufstellen, wobei AI_k für jeden Knoten der Join der Vorgängerknoten ist und $AI_S = d_S$.
Kleinsten Fixpunkt über $T_S : \mathcal{D}^n \to \mathcal{D}^n$ finden

↳ ex. wegen Monotonie → berechenbar wg. Stetigkeit durch Iteration mit $\perp$
↳ Iteration bricht wegen acc-Eigenschaft ab (sofern gegeben).

gilt Distributivität des Information Transforms, dann $AI^{MFP} = AI^{MOP}$
(Completeness/Coincidence Theorem) → erst Join, dann Transformer $\leq$ erst Transformer, dann Join

es gilt immer $AI^{MOP} \leq AI^{MFP}$ (Correctness-Theorem)

AE-Analyse für IC

Betrachte die Menge der verfügbaren Ausdrücke CE_P (alle Ausdrücke die in Unterausdrücken vorkommen)

Bitvektor mit Eintrag 0 für verfügbare Ausdrücke, 1 sonst → ACC Eigenschaft

$\perp = (0, \ldots 0)$ $\top = (1, \ldots, 1)$

Information Transforme durch gen (verfügbar machen), dann kill (zerstörte auf 1 setzen)
↳ ist distributiv, also MFP = MOP

Optimierung: wegen Schleifenstrukturen nicht wie bei SLC nach Wiederholungen gelangen (schwierig festzustellen), sondern grundsätzlich bei nicht verfügbaren Ausdrücken auf temp. Var. zwischenspeichern und bei verfügbaren Ausdrücken → dadurch überflüssige Kopieranweisungen).

Fixpunktiteration kann durch „chaotisches" Ausführen optimiert werden, indem komponentenweise gerechnet wird. Eine feste Indexfolge die die Komponenten auszählt wäre ein Beispiel:

hier besser: betrachte Flussgraph und berechne durch Worklist-Algorithmus immer nur die Komponenten die beitragen.

dazu: - initialisiere Worklist mit allen vorkommenden Kanten und A/s der Knoten mit $\perp_D$ bzw. d_S
- iteriere solange wie Worklist Elemente enthält, entferne darin jeweils vorderes Element (k, k') und prüfe ob diese Kante zur MFP-Iteration beiträgt, also die A verändert. Wenn gilt $\varphi_k(A/k) > A/k'$ setze A/k' auf Join von $\varphi_k(A/k) \sqcup A/k'$ und füge Kanten die von k' ausgehen neu in die Worklist (falls nicht vorhanden)

NV-Analyse (Needed-Variables):

Rückwärtstransformation nötig: Invertiere der Kanten im Flussgraphen Start- und Endknoten vertauscht

Analyseinformation: Potenzmenge der Programmvariablen

Startinformation: $D_S = \emptyset$

Informationstransformer: für Ausgabeknoten, Menge der Ausgabevariablen, $\varphi_{out}(M) := OV$

für die anderen Knoten erst kill, dann gen wie folgt: für Zuweisung $x \leftarrow e$ und $x \in$ Menge: kill x, dann gen e

für Bedingung if be then: gen bc

auch hier gilt φ_i ist distributiv: $NV^{MFP} = NV^{MOP}$

mit NVs aus globaler Analyse die lokale Analyse durchführen, ~~tote~~ und toten Code (Zuweisungen $x \leftarrow e$ mit $x \notin NV$) entfernen.

PaCo – Kap 5 (Schleifenoptimierung)

Bestimmung von Schleifen für: schnellere FP Berechnung / Code Motion / Reduktion der Stärke / Elimination von Induktionsvariablen

Für Schleifen $S \in Kno$ in Flußgraphen gilt, daß S stark zusammenhängend ist (von jedem Knoten ist jeder andere erreichbar) und nur ein Eingang (bzw. Startknoten) existiert.

Dominatoren: k' dom k, wenn jeder Weg vom Start zu k über k' läuft
k' dom k direkt, wenn jeder andere Knoten k'' dom k auch k'' dom k'

allgemein: $\rightarrow$ dominiert jeden Knoten und jeder Knoten dominiert sich selbst.
bis auf den Startknoten hat jeder Knoten einen direkten Dominator (Dominatorbaum)

Schleifen mit Rückwärtskanten bestimmen, dazu suche Kante (b,a) für die a dom b gilt. Nehme alle Knoten bis auf b fließen, aber über a zu gehen. Diese Menge ergibt Schleife mit Anfang a
↳ nat. Schleife, keine weiteren Schleifen / Zyklen.

Ein Flußgraph muß reduzierbar sein (nach entfernen aller Rückwärtskanten keine Zyklen)

ucd-chains: für eine Variable v und eine Marke i die Menge aller Marken an denen v definiert wird, so daß die Definition bei i gültig ist. v muß an Stelle i benutzt werden!

duc-chains: für eine Variable v / Label i die Menge aller Label wo v genutzt wird und hier definiert wurde $duc(v,i) := \{ l \mid i \in ud(v,l) \}$

Berechnung durch Vorwärtsanalyse mit SV-Graph

RD$_I$-Mengen: enthalten für Label l die Menge der „gültigen" Variablen, also Paare (v,i) wenn v bei i definiert und bis l gültig, wenn v an l benutzt wird → $ud(v,l) := \{ l' \mid (v,l') \in RD_l \}$

Abstrakte Interpretation: $D := pot(V_\pi \times L_\pi)$ mit V_π Programmvar und L_π Labels mit 0 := Eingang

Startinformation: $d_0 := \{(v,0) \mid v \in V\}$
Halbordnungsrelation: Teilmenge $\subseteq$

Informationstransformer: $i: v \leftarrow e$

kill $\{(v,*)\}$ (bisherige v's ungültig)

gen (v,i) (neues v wird hier definiert)

φ distributiv ?

Anweisungen sind Schleifeninvariant, wenn bei jedem Schleifendurchlauf derselbe Wert berechnet wird, betrachte bei $l: x \leftarrow e$ die Variablen V_e und bei $l: if\ e\ ...$ die V_{be}

für diese muss gelten: $ud(V, l) \cap S = \emptyset$ (die Definitionen erfolgen außerhalb der Schleife)

oder $ud(V, l) := \{l' \mid l' \subseteq S$ mit l' Schleifeninvariant$\}$

Kann induktiv berechnet werden.

Code-Motion: verschieben der invarianten Codes vor Schleifeneintritt
wenn folgende hinreichende Bedingungen gelten:
$l: x \leftarrow e$

1) an allen Ausgangsknoten ist x bereits gültig: l dom k'
   $\hookrightarrow k'' \neq k'$ dom k'', $k'' \notin S$

2) nur eine Definition von x in S

3) Benutzung von x nur hinter der Definition: wenn x an l' benutzt $\Rightarrow ud(x, l') = \{l\}$

Induktionsvariablen: eliminieren und Reduce in Strength

Basisinduktionsvariablen (BIV) werden konstant erhöht: $x \leftarrow e$ mit $e = x + c$

abhängige Induktionsvar (AIV) erhalten konst. Teilfaches von BIV oder Addition: $y \leftarrow e$ mit $e = x + c / x * c$

Konstante Veränderungen von BIV übertragen sich mit Faktor auf AIV.

Strength Reduction für eine Anweisung $y \leftarrow x * c$, mit $y \in AIV$, $x \in BIV$

$x \leftarrow x + 3$
$y \leftarrow x \cdot 4$
↓
$t \leftarrow x * 4$
$x \leftarrow x + 3$
$t \leftarrow t + 12$
$y \leftarrow t$

↳ an Schleifeneintritt mit $t \leftarrow x * c$ init. $c \triangleq const$
hinter jeder Def von x die Anweisung $(x \leftarrow x + d)$
$t \leftarrow t + (c * d)$
und statt y: $y \leftarrow t$

Elimination falls weiterhin gilt, dass y nur eine Definition hat und x nur für y und Bedingung benutzt, dann (nicht innerhalb S neu definiert)
x weglassen und direkt t einsetzen (ggf. Red. entsprechend Definition von t multiplizieren)

$x \leftarrow 1$
$t \leftarrow x \cdot 2$
---
if $x < 10$
$y \leftarrow t$
$t \leftarrow t + 2$

↳
$t \leftarrow 2$
---
if $t < 20$
$y \leftarrow t$
$t \leftarrow t + 2$

ggf. $> \rightarrow <$ / $<\rightarrow>$ (Relation umdrehen bei neg. Faktor!)

ICP ≙ IC-Programm mit Prozeduren (Deklarationen nicht geschachtelt)
Jede Prozedur mit eigenen Variablen / aufrufen über Bereiche

Beschreibung durch Gleichungssystem E_i für Prozedur i, Semantik wie IC
Bei Prozeduraufruf: rekursive Auflösung des entsprechenden Gleichungssystems, dann endrekursive Auflösung der nächsten Anweisung (Rücksprungfall)

MOP-Analyse

naives Verfahren: reguläre Pfadmenge benutzen (Aufruf- und Rücksprungkanten nicht gekoppelt)
↳ DFA-Technik direkt benutzbar aber sehr schwache Sicht, da unmögliche Pfade benutzt werden können.

deshalb: kontextfreie Pfadmenge mit Nichtterminalsymbolen $[i,j]$ für jeden Knoten i mit Endknoten j.
Produktionen: für jeden Knoten mögliche Nachfolger, ggf linksfaktorisiert

Flußgraph durch Vereinigung der einzelnen Flußgraphen der Prozeduren mit regulären Pfaden → vollständige Pfad P_{G_Π} mit Produktionen wie oben,
je Anweisung $P[i,j]$ mit Produktionen i gefolgt von Nachfolgei. bei Prozeduraufruf Startproduktion der Prozedur + Produktion nächster Knoten.

Die Sprache $L(P_{G_\Pi})$ erzeugt rekursive Pfadmenge mit allen gültigen Pfaden $0_1 \to 0_{q+1}$

Abstrakte Interpretation: mit Keller / Stack $D \to D$ normal, $D \times D \to D$ Rücksprung
Startinformation d_S für Startknoten des Hauptprogramms

Informationstransformer für Keller (Spitze rechts) wird:
$\varphi(w,d) \to w\,\varphi(d)$ normal
$\varphi(w,d) \to w d\,\varphi(d)$ Aufruf
$\varphi(w d_1 d_2) \to w\,\varphi(d_1,d_2)$ Rückgang

Lösung entsprechend Join über gültige Pfade

MFP-Analyse: Führe parallel zu den Gleichungen für die A_is noch Gleichungen zur Berechnung Φ_k für volle Transformation eine Prozedur k ein ($\Phi_k : D \to D$). Diese Gleichungen werden bei Prozeduraufrufen statt der normalen benutzt.

Informationstransformer: $\varphi_{i,j}$ / $\varphi^{call}_{i,j}$ / φ^{dum}_{ij}

Für pid_k: $\varphi_{ij}(d) = \varphi^{return}_{ij}(d, \Phi_k(\varphi^{call}_{ij}(d)))$