Helmut Seidl

Programmoptimierung

TU München

Wintersemester 2004/05

Kommentare im Skript : F. Hassmann

Inhaltsverzeichnis

Organisatorisches

Termine:

Vorlesung:	Montag, 12-14
	Dienstag, 12-14
Übung:	Freitag, 10-12
	K. Neeraj Verma: verma@in.tum.de
Materialien:	Folien, Aufzeichnung :-)
	Literatur :-))
	(Erhältlich über die Website des Lehrstuhls)
	Vorlesungs-Mitschrift (in Überarbeitung)

Schein:

- 50% der Aufgaben
- zweimal vorrechnen :-)

Geplanter Inhalt:

1. Vermeidung Überflüssiger Berechnungen

   → verfügbare Ausdrücke

   Vermeidung der Neuberechnung von Ausdrücken, die vorher eben schon berechnet worden sind.

   → Konstantenpropagation/Array-Bound-Checks

   Falls man erkennt, dass ein komplizierter Ausdruck immer den gleichen Wert hat, kann dieser durch eine Konstante ersetzt werden.
   Array-Bound-Checks: z.B. Bei JAVA : Grenzen von arrays werden bei jedem Zugriff wieder überprüft.
   In Schleifen wird somit immer die Gültigkeit des Index überprüft.

   → Code Motion

   „Verschieben“ von Code. z.B. : In einer Schleife wird die gleiche Berechnung immer wieder durchgeführt. Diese Berechnung wird dann aus der Schleife herausgeschoben.

2. Ersetzen teurer Berechnungen durch billigere.
   Prinzipielle Frage : Was ist eine „teuere“bzw. eine „billige“Berechnung?

→ Peep Hole Optimierung

Idee : Nach der Erzeugung von normalem Code werden kleine Stückchen des Codes auf ihre Effizienz hin untersucht.
z.B. Einer Zuweisung folgt eine Inkrementberechnung.
Dies kann eventuell durch einen einzigen Befehl ersetzt werden.

→ Inlining

Prozeduraufrufe sind „teuer“.
Falls nun ein Prozedurrumpf relativ klein ist,
kann man diesen an die Aufrufstellen kopieren und somit den „overhead“ eines Prozeduraufrufs vermeiden.

→ Reduction of Strength

Ursprünglich der Versuch Multiplikationen durch Additionen zu ersetzen.
Dies ist auf modernen Rechnerarchitekturen wohl nicht mehr sehr effizient.

3. Anpassung an Hardware

Ein Programm sollte möglichst gut den gegebenen, speziellen Hardware-Eigenschaften angepasst werden.
Ein Compiler sollte die speziellen features des Prozessors kennen und effektiv benutzen.

→ Instruktions-Selektion

Es sollen die effizientesten Assemblerinstruktionen ausgewählt werden,
da es eine Vielfalt von Instruktionen gibt, die das Gleiche bewerkstelligen.
Falls konkurrierende Kombinationen von Instruktionen vorhanden sind,
soll dann die „billigste“, bzw. die „geschickteste“ausgewählt werden

→ Registerverteilung

Da der Zugriff auf ein Register schnell ist, wird versucht, wichtige Daten (auf die eben oft zugegriffen wird) in Registern zu halten um die Programmausführung zu beschleunigen. Es gibt natürlich auch noch andere Speicherbereiche mit unterschiedlich schnellen Zugriffszeiten. Hier sind die sog. „cashes“zu erwähnen. Diese haben ähnlich schnelle Zugriffszeiten wie Register. Somit ist die effiziente Benutzung der verschiedenen cashes zur Datenhaltung geeignet. Nur können diese von der Seite des Programmierers schlecht kontrolliert werden.

→ Scheduling

Moderne Prozessoren bieten „Nebenläufigkeit“auf dem Chip an. Diese Parallelität ist schwierig nutzbar, falls das auszuführende Programm sequentiell ist. Falls vorhandene

„Onchip-parallelität“gegeben ist und genutzt werden soll, muss die entsprechende Parallelität vorgenommen werden.
Falls ein Programm „Nebenläufigkeiten“ enthält, müssen die ausführenden Recheneinheiten mit Daten versorgt werden.

→ Speicherverwaltung

Durch Umstrukturierung eines Programms, kann eine bessere Nutzung des Speichers erlangt werden.

Im Bereich der Optimierung kommen eine Reihe von allgemeinen wichtigen Softwaretechniken zum Einsatz.

0 Einführung

Beobachtung 1:

Intuitive Programme sind oft ineffizient.

Beispiel:

swap, *Unterprogramm eines Sortieralgorithmus, z.B. quicksort.*

```
void swap (int i, int j) {
     int t;
     if (a[i] > a[j]) {
         t = a[j];
         a[j] = a[i];
         a[i] = t;
        }
     }
```

Ineffizienzen:

- Adressen `a[i]`, `a[j]` werden je dreimal berechnet :-(
  Also 6 Adressberechnungen. Hier sollten 2 Adressberechnunegn genügen.
- Werte `a[i]`, `a[j]` werden zweimal geladen :-(
  Also 4 Ladeoperationen. Hier sollten 2 Ladeoperationen genügen.

Verbesserung:

- Gehe mit Pointer durch das Feld `a`;

- speichere die Werte von `a[i]`, `a[j]` zwischen!

```
void swap (int *p, int *q) {
     int t, ai, aj;
     ai = *p; aj = *q;
     if (ai > aj) {
         t = aj;
         *q = ai;
         *p = t;    // t kann auch noch
        }           // eingespart werden!
     }
```

Beobachtung 2:

Höhere Programmiersprachen (sogar C :-) abstrahieren von Hardware und Effizienz.
Deshalb sollten C-compiler in gewisser Weise auch „intelligent“sein.

Aufgabe des Compilers ist es, den natürlich erzeugten Code an die Hardware anzupassen.

Beispiele:

Zur vernünftigen Ausnuzung der Hardware

. . . Füllen von Delay-Slots;
z.B. : Bei einem Sprung wird die Instruktion hinter dem Sprung auch noch ausgeführt **bevor** *der Sprung durchgeführt wird : (NOP)(SPARC-Architektur)*
Dient zur effizienten „Füllung der pipeline “.
Man lädt also die nächste Instruktion schon mal prophylaktisch!

. . . Einsatz von Spezialinstruktionen;

. . . Umorganisation der Speicherzugriffe für besseres Cache-Verhalten;
Es wird ein Datenblock in den Cache eingelesen. Bei Zugriff auf Elemente eines arrays sollte beachtet werden, dass diese innerhalb dieses Blockes liegen. Man sollte also versuchen bei der Iteration über ein array so sequentiell wie möglich auf dieses zuzugreifen.

. . . Beseitigung (unnötiger) Tests auf Overfbw/Range.

Beobachtung 3:

Programm-Verbesserungen sind nicht immer korrekt :-(

Beispiel:

```
y = f() + f();   ⟹   y = 2 * f();
```

Idee: Spare zweite Auswertung von `f()` ???

Problem: Die zweite Auswertung könnte ein anderes Ergebnis liefern als die erste (z.B. wenn `f()` aus der Eingabe liest :-)

Die Verbesserung ist also nur richtig, falls beide Funktionsaufrufe garantiert das gleiche Ergebnis liefern. (In einer imperativen Programmiersprache kann der zweite Aufruf einer Funktion natürlich einen anderen Wert liefern, als der erste. In einer funktionalen Programmiersprache ohne Seiteneffekte könnte die Verbesserung möglicherweise richtig sein.)

Folgerungen:

⟹ Optimierungen haben Voraussetzungen.
Ergo: Nicht jede plausibel erscheinende Optimierung ist vernünftig.
Was ist nun die Voraussetzung, damit die Umorganisierung eines Programmes korrekt ist?
Es muss ein Verfahren zur Überprüfung der Korrektheit der Umorganisierung gefunden werden.
Korrektheitsproblem: Überprüfung der Semantik.

⟹ Die Voraussetzungen muss man:

- formalisieren,
- Überprüfen :-)

⟹ Man muss beweisen, dass die Optimierung korrekt ist, d.h. die Semantik erhält !!!
D.h. Es muss gezeigt werden, dass die Semantik des Programms nach der Optimierung erhalten bleibt.
Eben, dass das ursprüngliche und das transformierte Programm äquivalent sind. Programmiersprachen sind zumindest in der Theorie turingmächtig, somit ist jede nicht triviale Frage über ein Programm entscheidbar. Man muss also ein Verfahren finden, welches die Korrektheit eines transformierten Programms überprüfen kann. Das Programm sollte vor und

nach der Optimierung das Gleiche tun. Dies ist u.U. sehr schwierig. Insbesondere, wenn man neue Transformationen entwickelt, sollte deren Korrektheit sehr genau überprüft werden.

Beobachtung 4:

Optimierungs-Techniken hängen von der Programmiersprache ab:
Verschiedene Programmiersprachen haben unterschiedliche Ineffizienzen.
Bei sehr unterschiedlichen Sprachen wie z.B. JAVA und PROLOG ist dies sehr evident.

→ welche Ineffizienzen auftreten;

→ wie gut sich Programme analysieren lassen;
z.B. „C“ ist schwierig zu analysieren, insbesondere durch die Verwendung von Pointern in C.
JAVA-Programme sind leichter zu analysieren, Schwierigkeiten ergeben sich hier z.B. beim dynamischen Laden von Klassen, oder bei Bibliotheken.
Einfach zu analysieren sind Programmiersprachen mit einer klaren Semantik :
z.B. die funktionale Sprache HASKELL.
Bei Metaprogrammierungseigenschaften in einer Programmiersprache ist es i.A. schwierig festzustellen, was ein Programm macht. Insbesondere ist es hier auch schwierig nach einer Transformation die Korrektheit festzustellen.

→ wie schwierig / unmöglich es ist, Korrektheit
zu beweisen ...

Beispiel: Java

Was gibt es hier für Ineffizienzen?
Wie sieht es mit der Analysierbarkeit und den Korrektheitsbeweisen aus?

Unvermeidbare Ineffizienzen:

- Array-Bound Checks;
- dynamische Methoden-Auswahl;
- bombastische Objekt-Organisation ...

Analysierbarkeit:

\+ keine Pointer-Arithmetik;
\+ keine Pointer in den Stack;
− dynamisches Klassenladen;

(Nachladen von Programmteilen unbekannter Herkunft)

– Reflection, Exceptions, Threads, ...

Korrektheitsbeweise:

\+ mehr oder weniger definierte Semantik;
Bei Java wird in verschiedenen Projekten versucht, eine formalisierte Semantik zu erstellen.

– Features, Features, Features;
Bei einem Korrektheitsbeweis muss eine Fallunterscheidung nach **allen** *Features durchgeführt werden. Deshalb wird in der Vorlesung Java nicht als Beispielsprache benutzt.*
Es ändert sich auch z.B. die Semantik von Klassen je nach Version.

– Bibliotheken mit wechselndem Verhalten ...
Bibliotheken haben nicht immer eine konsistente Semantik, je nach Version kann diese sich ändern.

... in der Vorlesung:

Wir benützen eine einfache imperative Programmiersprache. Sie soll so einfach wie möglich, aber doch noch so realistisch sein, dass sie anwendbar ist.
Man kann sich diese Programmiersprache als eine Art von Zwischensprache vorstellen, in die das ursprüngliche Programm übersetzt sein soll.

eine einfache imperative Sprache mit:

• Variablen	//	Register
• $R = e;$	//	Zuweisungen
• $R_1 = M[R_2];$	//	Laden
• $M[R_1] = R_2;$	//	Speichern
• if (e) s_1 else s_2	//	bedingte Verzweigung
• goto $L;$	//	keine Schleifen :-)

Beachte:

- Vorerst verzichten wir auf Prozeduren :-)
- Externe Funktionen berücksichtigen wir, indem wir als Ausdruck e auch $f(R_1, \ldots, R_k)$ gestatten für eine unbekannte Funktion f.
  D.h. In den Registern werden die Argumente übergeben. Die Register werden vor dem Programmaufruf gerettet und danach restauriert. Dies sind auch die üblichen Aufrufkonventionen

bei Compilern, sowie auch bei der Hardware.

$\Longrightarrow$ intra-prozedural

Optimierungen werden vorerst nunr innerhalb von Prozeduren durchgeführt.

$\Longrightarrow$ eine Art Zwischensprache, in die man (fast) alles übersetzen kann.

Man kann hiermit im Prinzip Assemblercode analysieren, man hat Zuweisungen, Gotos und Verzweigungen.

Beispiel: `swap()`

Die Assemblerebene hat den Vorteil, dass nun die Ineffizienzen die vorher nur implizit im Programm vorhanden waren, völlig sichtbar werden. Die Berechnung der Adressen muss explizit durchgeführt werden.

```
0 :   A1 = A0 + 1 * i;          //   A0 == &a
1 :   R1 = M[A1];               //   R1 == a[i]
2 :   A2 = A0 + 1 * j;
3 :   R2 = M[A2];               //   R2 == a[j]
4 :   if (R1 > R2) {
5 :         A3 = A0 + 1 * j;
6 :         t  = M[A3];
7 :         A4 = A0 + 1 * j;
8 :         A5 = A0 + 1 * i;
9 :         R3 = M[A5];
10 :        M[A4] = R3;
11 :        A6 = A0 + 1 * i;
12 :        M[A6] = t;
            }
```

Dies ist nun der „naive Code“. Man sieht, dass dieser viele Ineffizienzen enthält.

Optimierung 1: $1 * R \Longrightarrow R$ *Der Skalierungsfaktor ist in unserem Fall 1 und kann somit weggelassen werden.*

Optimierung 2: Wiederbenutzung von Teilausdrücken

Bestimmte Register enthalten immer den gleichen Wert.
Somit kommt man auch mit weniger Registern aus.

$$A_1 == A_5 == A_6$$
$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$
$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

Nun kann das Programm automatisch transformiert werden.

Damit erhalten wir:

$$\begin{array}{lcl} A_1 & = & A_0 + i; \\ R_1 & = & M[A_1]; \\ A_2 & = & A_0 + j; \\ R_2 & = & M[A_2]; \\ \text{if } (R_1 > R_2) \{ & & \\ \quad t & = & R_2; \\ \quad M[A_2] & = & R_1; \\ \quad M[A_1] & = & t; \\ \quad \} & & \end{array}$$

Hierbei fällt auf, dass auch **t** *überflüssig ist. Hier kann eine Verkürzung von Zuweisungsketten durchgeführt werden.*

Optimierung 3: Verkürzung von Zuweisungsketten :-)

Ersparnis:

	vorher	nachher
$+$	6	2
$*$	6	0
load	4	2
store	2	2
$>$	1	1
$=$	6	2

All diese Optimierungen sollten automatisch realisiert werden. Dies wird durch mehrere Programmtransformationen geschehen.

1 Vermeidung überflüssiger Berechnungen

Man kann prinzipiell oft „Rechenzeit“gegenüber „Speicherplatz“ausspielen. Wird eine Berechnung zweimal ausgeführt, kann der Wert nach der ersten Berechnung abgespeichert werden. Die zweite Berechnung wird dann durch „Nachschlagen“ersetzt.

1.1 Mehrfach-Berechnungen

Idee:

Wird der gleiche Wert mehrfach berechnet, dann

→ speichere ihn nach der ersten Berechnung;

→ ersetze jede weitere Berechnung durch Nachschlagen!
Manchmal ist aber auch eine Neuberechnung „billiger“, insbesondere wenn der Speicherplatz knapp ist. Nur solange genügend Platz (cache, Register, Speicher) vorhanden ist, sollte eine Neuberechnung vermieden werden.

$\Longrightarrow$ Verfügbarkeit von Ausdrücken

$\Longrightarrow$ Memoisierung
z.B. Dynamisches Programmieren : z.B. Parsen von kontexfreien Grammatiken

Problem: Erkenne Mehrfach-Berechnungen!

Beispiel:

$$\begin{array}{lll} & z & = & 1; \\ & y & = & \mathsf{read}(); \\ A: & x_1 & = & \boxed{y+z}; \\ & & \ldots & \\ B: & x_2 & = & \boxed{y+z}; \end{array}$$

Achtung:
Hier ist nicht sicher, dass der Ausdruck „y + z“bei der zweiten Zuweisung den gleichen Wert hat, wie bei der ersten.

B ist eine Mehrfach-Berechnung des Werts von $\boxed{y+z}$, falls:

(1) A stets vor B ausgeführt wird; und

(2) y und z an B die gleichen Werte haben wie an A :-)

$\Longrightarrow$ Wir benötigen

→ eine operationelle Semantik :-)

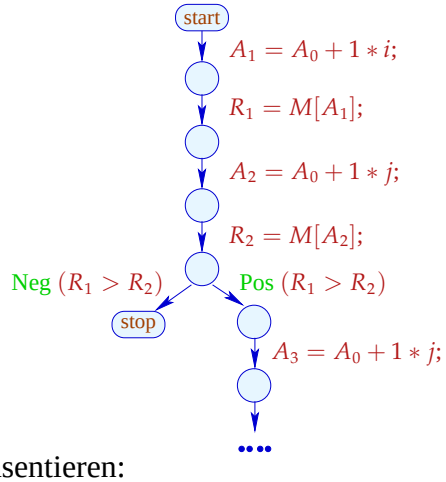
→ ein Verfahren, das einige Mehrfach-Berechnungen erkennt ...

Es müssen nicht alle Mehrfachberechnungen erkannt werden, obwohl dies das Optimum wäre. Somit erscheinen diese zwei Voraussetzungen hinreichend für das Vorliegen einer Mehrfachberechnung an der Stelle B gegenüber der Stelle A.

Exkurs 1: Eine operationelle Semantik

Wir wählen einen small-step operationellen Ansatz.
Wir formalisieren was ein Berechnungsschritt ist. Eine Berechnung ist dann eine Folge von Berechnungsschritten.
Programme repräsentieren wir als Kontrollfluss-Graphen.
Die Grundlage für die operationelle Semantik ist ein Kontrollfluss-Graph. Die Knoten entsprechen den Programmpunkten, die während einer Berechnung durchlaufen werden. Während eines Berechnungsschritts werden die Befehle ausgeführt die an einer Kante stehen. (Zuweisungen, loads, stores, Bedingungen und NOP)

Im Beispiel:



Dabei repräsentieren:

Knoten	Programm-Punkt
start	Programm-Anfang
stop	Programm-Ende
Kante	Berechnungs-Schritt

Kanten-Beschriftungen:

Test :	Pos (e) oder Neg (e)
Zuweisung :	$R = e;$
Load :	$R_1 = M[R_2];$
Store :	$M[R_1] = R_2;$
Nop :	;

Berechnungen folgen Pfaden.

Berechnungen transformieren den aktuellen Zustand

$$s = (\rho, \mu)$$

wobei:

$\rho : \mathit{Vars} \to \mathbf{int}$	Inhalt der Register
$\mu : \mathbb{N} \to \mathbf{int}$	Inhalt des Speichers

Der Einfachheit halber, beschränken wir uns auf Integer.

Jede Kante $k = (u, lab, v)$ definiert eine partielle Transformation
Die Werte der Register und die Werte des Speichers sind also die Werte unseres Zustandes. Partiell heisst, dass für manche inputs die Transformation nicht definiert ist. (z.B Bedingungen).
Diese Transformation ist nur von der Aktion abhängig die an der Kante steht, nicht vom Programmpunkt. Die Kanten entsprechen Transformationen des Zustands. Diese Transformationen hängen von der Aktion ab. Der Zustand ist Register- und Speicherinhalte.

$$[\![k]\!] = [\![lab]\!]$$

des Zustands:

Nop: Zustand bleibt erhalten

$$[\![;]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu)$$

Bedingung (Partielle Identität) : Zustand (Register und Speicher) ändern sich nicht.

$$[\![\mathrm{Pos}\,(e)]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu) \qquad \text{falls } [\![e]\!]\,\rho \neq 0$$
$$[\![\mathrm{Neg}\,(e)]\!]\,(\rho, \mu) \quad = \quad (\rho, \mu) \qquad \text{falls } [\![e]\!]\,\rho = 0$$

// $[\![e]\!]$: Auswertung des Ausdrucks e, z.B.

// $[\![x + y]\!]\,\{x \mapsto 7, y \mapsto -1\} = 6$
// $[\![!(x == 4)]\!]\,\{x \mapsto 5\} = 1$

Zuweisung: Variablenbelegung ändert sich

$$[\![R = e;]\!]\,(\rho, \mu) \quad = \quad (\rho \oplus \{R \mapsto [\![e]\!]\,\rho\}, \mu)$$

// wobei "$\oplus$" eine Abbildung an einer Stelle ändert

Load: Speicherinhalt bleibt erhalten, Variablenbelegung (Register) ändert sich.

$$[\![R_1 = M[R_2];]\!](\rho,\mu) \quad = \quad (\boxed{\rho \oplus \{R_1 \mapsto \mu(\rho(R_2))\}}, \mu)$$

Store: Register (Variablen) bleiben gleich, Speicher ändert sich.

$$[\![M[R_1] = R_2;]\!](\rho,\mu) \quad = \quad (\rho, \boxed{\mu \oplus \{\rho(R_1) \mapsto \rho(R_2)\}})$$

Beispiel:

Das $\oplus$ ist praktisch ein „Links-update“.
Es nimmt die Variablenbelegung links und überschreibt das Paar für R_1
Eine Zuweisung bewirkt also eine Transformaton des Systemzustandes.
Dieser Systemzustand besteht ja aus zwei Teilen:
Den Werten der Variablen, bzw. Registern und
den Wertes des Speichers.
Eine Zuweisung an eine Variable modifiziert die Variable (Register).
Der Speicher bleibt unverändert.

$[\![x = x + 1;]\!](\{x \mapsto 5\}, \mu) = (\rho, \mu)$ wobei:

$$\begin{aligned} \rho &= \{x \mapsto 5\} \oplus \{x \mapsto [\![x+1]\!]\{x \mapsto 5\}\} \\ &= \{x \mapsto 5\} \oplus \{x \mapsto 6\} \\ &= \{x \mapsto 6\} \end{aligned}$$

Was passiert nun an einem Pfad?
Eine Berechnung besteht aus einer Folge von Kanten.
Ein Pfad besteht aus einer Folge von Kanten.
Jede Kante ist eine Transformation des Zustandes.
Es ist zu beachten, dass es Kanten gibt, deren Transformation partiell ist (z.b. bei Bedingungen). Somit ergeben sich Transformationen die nicht definiert sind.

Ein Pfad $\pi = k_1 k_2 \ldots k_m$ ist eine Berechnung für den Zustand s falls:

$$s \in def\,([\![k_m]\!] \circ \ldots \circ [\![k_1]\!])$$

Das Ergebnis der Berechnung ist:

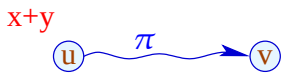
$$[\![\pi]\!]\, s = ([\![k_m]\!] \circ \ldots \circ [\![k_1]\!])\, s$$

Die Komposition der Effekte aller Kanten

Anwendung:

Verfügbarkeit von Ausdrücken um Mehrfachberechnungen einzusparen.

Nehmen wir an, wir hätten am Punkt u den Wert von $x + y$ berechnet:



Wir führen eine Berechnung entlang des Pfades π aus und erreichen v, wo wir erneut $x + y$ berechnen sollen ...

Wann können wir also davon ausgehen, dass x+y den gleichen Wert als zuvor liefert, und somit einfach durch Nachschlagen gefunden werden kann?
Man muss herausfinden, ob x und y entlang des Pfades verändert wurden.

Idee:

Wenn x und y in π nicht verändert werden, dann muss $x + y$ in v den gleichen Wert liefern wie in u :-)
Diese Eigenschaft können wir an jeder Kante in π überprüfen :-}
Das macht i.A. auch ein Compiler

Allgemeiner:

Nehmen wir an, in u hätten wir die Werte der Ausdrücke aus $A = \{e_1, \ldots, e_r\}$ zur Verfügung.

Jede Kante k transformiert diese Menge in eine Menge $[\![k]\!]^\sharp A$ von Ausdrücken,
die nach Ausführung von k verfügbar sind ...

Beim Entlanggehen einer Kante ändert sich also die Menge der verfügbaren Ausdrücke durch Hinzufügen und Herausnehmen.

... die wir zur Ermittlung des Effekts eines Pfads $\pi = k_1 \ldots k_r$ zusammen setzen können:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp$$

Der Effekt ist also die Komposition dieser Transformationen.
Der Effekt an einer Kante hängt also von der Aktion (= Label) an dieser Kante ab.

Der Effekt $[\![k]\!]^\sharp$ einer Kante $k = (u, lab, v)$
hängt nur vom Label lab ab, d.h. $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ wobei:

$$
\begin{array}{lcl}
[\![;]\!]^\sharp\, A & = & A \\
[\![Pos(e)]\!]^\sharp\, A & = & [\![Neg(e)]\!]^\sharp\, A \quad = \quad A \cup \{e\} \\
[\![R = e;]\!]^\sharp\, A & = & (A \cup \{e\}) \backslash Expr_R \qquad \text{wobei} \\
 & & Expr_R \text{ alle Ausdrücke sind, die } R \text{ enthalten}
\end{array}
$$

$$
\begin{array}{lcl}
[\![R_1 = M[R_2];]\!]^\sharp\, A & = & A \backslash Expr_{R_1} \\
[\![M[R_1] = R_2;]\!]^\sharp\, A & = & A
\end{array}
$$

Damit können wir jeden Pfad untersuchen :-)
In einem Programm kann es mehrere Pfade geben :-(
Bei jeder Eingabe kann ein anderer gewählt werden :-((

$\Longrightarrow$ Wir benötigen die Menge:

$$\mathcal{A}[v] \quad = \quad \bigcap \{ [\![\pi]\!]^\sharp \emptyset \mid \pi : start \rightarrow^* v \}$$

Im Klartext:

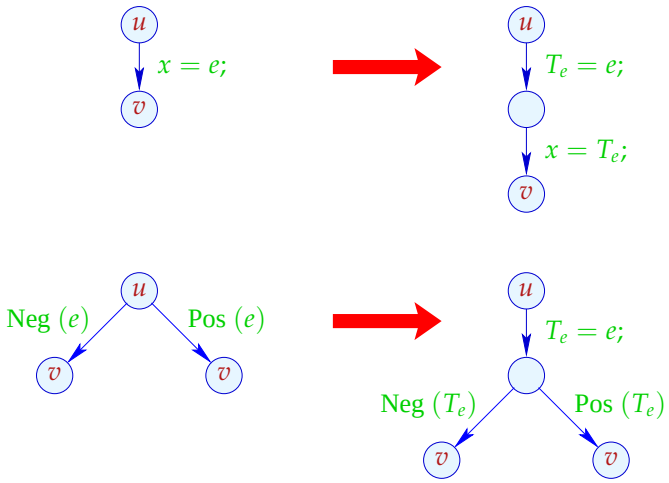
- $\rightarrow$ Wir betrachten sämtliche Pfade, die v erreichen.
  Man weiss nicht, welcher Pfad bei der Programmausführung genommen wird, deshalb müssen alle Pfade untersucht werden.
- $\rightarrow$ Für jeden Pfad π bestimmen wir die Menge der entlang π verfügbaren Ausdrücke.
- $\rightarrow$ Vor Programm-Ausführung ist nichts verfügbar :-)
- $\rightarrow$ Wir bilden den Durchschnitt $\Longrightarrow$ sichere Information

Wie nutzen wir diese Information aus ???

Der Kontrollflussgraph wird abgeändert, indem man neue Knoten und Kanten einführt.

Transformation 1:

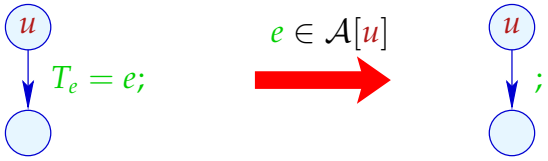
Wir stellen neue Register T_e als Speicherplatz für die e bereit:
Wir stellen Hilfsregister zur Verfügung für alle Ausdrücke e deren Werte aufgehoben werden sollen.
Es ergibt sich eine Graphersetzungsrgegel folgender Form:



Transformation 2:

Falls e am Punkt u verfügbar ist, wird e nicht neu berechnet:

e ist also am Punkt u in der Menge der verfügbaren Ausdrücke enthalten.



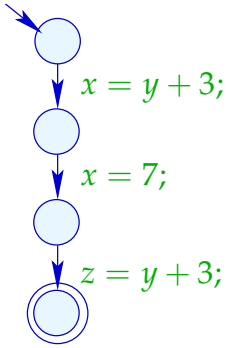
Wir ersetzen dann die Zuweisung durch Nop :-)

Die Transformationen sind i.A. einfach. Aufwendig ist es, die notwendigen Vorbedingungen zu schaffen.

Hier die Frage bei Konstanten :
Es muss je nach den Prozessoreigenschaften berücksichtigt werden,
ob das Laden von Konstanten „teuer“ist.
Ggf. kann das Laden von Konstanten auch durch Nachschlagen ersetzt werden.

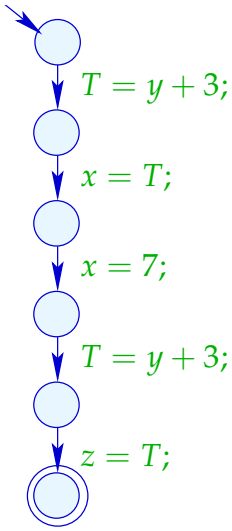
Beispiel:

$$\begin{aligned} x &= y+3; \\ x &= 7; \\ z &= y+3; \end{aligned}$$



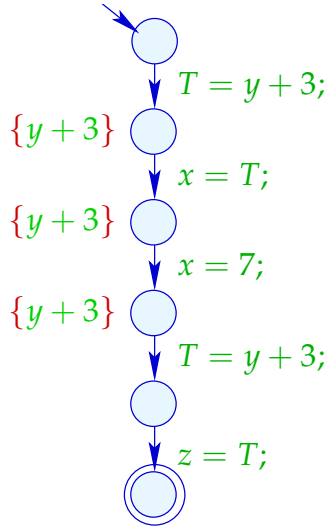
Beispiel (Schritt 1):

$$\begin{aligned} x &= y+3; \\ x &= 7; \\ z &= y+3; \end{aligned}$$



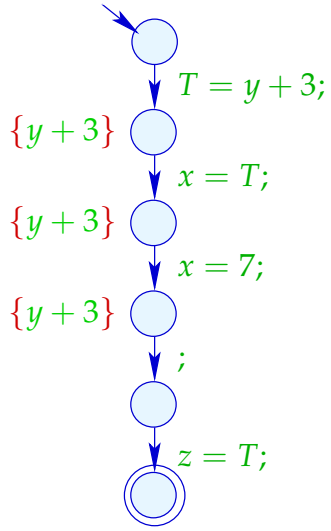
Beispiel (Schritt 2):

$x = y+3;$
$x = 7;$
$z = y+3;$



Beispiel (Schritt 3):

$x = y+3;$
$x = 7;$
$z = y+3;$



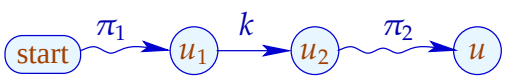
Korrektheit: (Idee)

Die Transformation ändert die Bedeutung des Programms nicht.
Auch die berechneten Informationen an den alten Knoten u und v bleiben erhalten.

Transformation 1 erhält offenbar die Bedeutung und $\mathcal{A}[u]$ für alle Knoten u :-)

Sei $\pi : start \to^* u$ der Pfad, den eine Berechnung nimmt.
Ist $e \in \mathcal{A}[u]$, dann auch $e \in [\![\pi]\!]^\sharp \emptyset$.

Dann muss es eine Zerlegung von π geben:



mit den folgenden Eigenschaften:

- Der Ausdruck e wird an der Kante k berechnet;
- Der Ausdruck e wird an keiner Kante in π_2 aus der Menge der verfügbaren Ausdrücke entfernt, d.h. keine Variable von e erhält einen neuen Wert :-)

$\Longrightarrow$

Wird u erreicht, enthält das Register T_e den Wert von e :-))

Achtung:

Die Transformation 1 ist nur sinnvoll an Zuweisungen $x = e;$, wobei:

→ $x \notin Vars(e)$;

→ $e \notin Vars$;

→ sich die Berechnung von e lohnt :-}

Hier stellt sich die Frage, was ist ein komplizierter Ausdruck.
Dies ist sehr stark von der Hardwarekonfiguration abhängig.
Oder: Sollte man Werte von Konstanten aufheben?
Falls man z.B. 2 Assemblerbefehle benötigt, könnte man Konstanten auch in Registern halten. (z.B. SPARC- oder RISC-Architekturen.
Es kann somit durchaus sinnvoll sein, Konstante in Registern aufzubauen und diese dann durch einen MOVE-Befehl zu laden. Auch z.B. 0 (Die Null). Diese lässt sich mit einem XOR.-Befehl auf einen beliebigen Wert herstellen.

Bleibt die Preisfrage ...

Wie berechnen wir $\mathcal{A}[u]$ für jeden Programmpunkt u ??

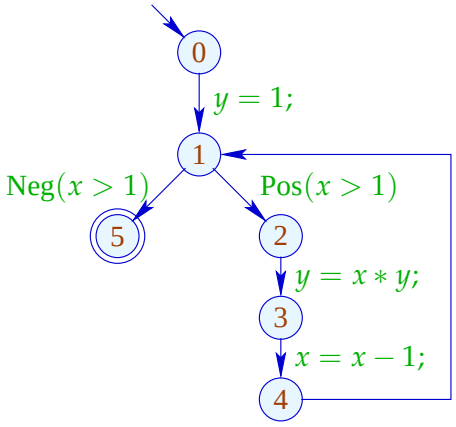
Idee:

Wir stellen ein Constraint-System *(Ungleichungssystem]* auf, das alle Bedingungen an die Werte $\mathcal{A}[u]$ sammelt:

$$\begin{array}{lll} \mathcal{A}[start] & \subseteq & \emptyset \\ \mathcal{A}[v] & \subseteq & [\![k]\!]^\sharp(\mathcal{A}[u]) \qquad\qquad k = (u, \_, v) \quad \text{Kante} \end{array}$$

Eine Kante besteht also aus 3 Bestandteilen: Startpunkt (u), Zielpunkt (v) und Label (_).
Man nimmt den Durchschnitt der Werte aller eingehenden Kanten von v.

Beispiel:

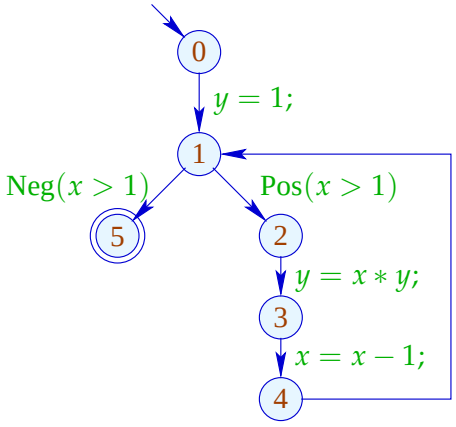


$$
\begin{array}{lcl}
\mathcal{A}[0] & \subseteq & \emptyset \\
\mathcal{A}[1] & \subseteq & (\mathcal{A}[0] \cup \{1\}) \backslash Expr_y \\
\mathcal{A}[1] & \subseteq & \mathcal{A}[4] \\
\mathcal{A}[2] & \subseteq & \mathcal{A}[1] \cup \{x > 1\} \\
\mathcal{A}[3] & \subseteq & (\mathcal{A}[2] \cup \{x * y\}) \backslash Expr_y \\
\mathcal{A}[4] & \subseteq & (\mathcal{A}[3] \cup \{x - 1\}) \backslash Expr_x \\
\mathcal{A}[5] & \subseteq & \mathcal{A}[1] \cup \{x > 1\}
\end{array}
$$

Gesucht:

- möglichst große Lösung (??)
  Am Besten eine! grösste Lösung
- Algorithmus, der diese berechnet :-)

Beispiel:



Lösung:

$$
\begin{array}{lcl}
\mathcal{A}[0] & = & \emptyset \\
\mathcal{A}[1] & = & \{1\} \\
\mathcal{A}[2] & = & \{1, x > 1\} \\
\mathcal{A}[3] & = & \{1, x > 1\} \\
\mathcal{A}[4] & = & \{1\} \\
\mathcal{A}[5] & = & \{1, x > 1\}
\end{array}
$$

Beobachtung:

Die Menge aller möglichen Werte haben eine gewisse Struktur.
Die Struktur heisst : Vollständiger Verband.

Somit benötigt man zur Lösung dieser Ungleichungen vollständige Verbände
und die weitere Voraussetzung, dass die Funktionen an den Kanten monoton sind.

- Die möglichen Werte für $\mathcal{A}[u]$ bilden einen vollständigen Verband:

$$\mathbb{D} = 2^{Expr} \quad \text{mit} \quad B_1 \sqsubseteq B_2 \quad \text{gdw.} \quad B_1 \supseteq B_2$$

- Die Funktionen $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ sind monoton, d.h.

$$[\![k]\!]^\sharp(B_1) \sqsubseteq [\![k]\!]^\sharp(B_2) \quad \text{gdw.} \quad B_1 \sqsubseteq B_2$$

Exkurs 2: Vollständige Verbände

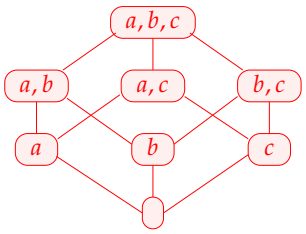
Eine Menge $\mathbb{D}$ mit einer Relation $\sqsubseteq \subseteq \mathbb{D} \times \mathbb{D}$ ist eine partielle Ordnung *Halbordung* falls für alle $a, b, c \in \mathbb{D}$ gilt:

($\sqsubseteq$ ist die Abstraktion der Zeichen $\subseteq$ und $\leq$)

$$\begin{array}{lll} a \sqsubseteq a & & \textit{Reflexivität} \\ a \sqsubseteq b \wedge b \sqsubseteq a & \Longrightarrow \; a = b & \textit{Anti - Symmetrie} \\ a \sqsubseteq b \wedge b \sqsubseteq c & \Longrightarrow \; a \sqsubseteq c & \textit{Transitivität} \end{array}$$

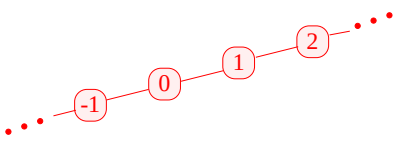
Beispiele:

1. $\mathbb{D} = 2^{\{a,b,c\}}$ mit der Relation "$\subseteq$" :

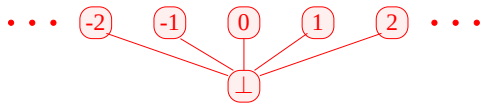


2. $\mathbb{Z}$ mit der Relation "$=$" :

3. $\mathbb{Z}$ mit der Relation "$\leq$" :



4. $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$ mit der Ordnung:



$\perp$ *ist das sog. Bottom-Element*

$d \in \mathbb{D}$ heißt obere Schranke für $X \subseteq \mathbb{D}$ falls

$$x \sqsubseteq d \qquad \text{für alle } x \in X$$

d heißt kleinste obere Schranke (lub) *lowest upper bound* falls

1. d eine obere Schranke ist und
2. $d \sqsubseteq y$ für jede obere Schranke y für X.

Achtung:

- $\{0, 2, 4, \ldots\} \subseteq \mathbb{Z}$ besitzt keine obere Schranke!
- $\{0, 2, 4\} \subseteq \mathbb{Z}$ besitzt die oberen Schranken $4, 5, 6, \ldots$

Ein vollständiger Verband (cl (complete lattice)) $\mathbb{D}$ ist eine partielle Ordnung, in der jede Teilmenge $X \subseteq \mathbb{D}$ eine kleinste obere Schranke $\bigsqcup X \in \mathbb{D}$ besitzt.

Beachte:

Jeder vollständige Verband besitzt

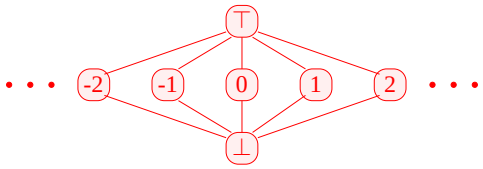
$\rightarrow$ ein kleinstes Element $\perp = \bigsqcup \emptyset \in \mathbb{D}$;(Bottom-Element)

$\rightarrow$ ein größtes Element $\top = \bigsqcup \mathbb{D} \in \mathbb{D}$.(Top-Element)

Beispiele:

1. $\mathbb{D} = 2^{\{a,b,c\}}$ ist ein cl :-)
2. $\mathbb{D} = \mathbb{Z}$ mit "$=$" ist keiner.
3. $\mathbb{D} = \mathbb{Z}$ mit "$\leq$" ebenfalls nicht.
4. $\mathbb{D} = \mathbb{Z}_\perp$ auch nicht :-(
5. Mit einem zusätzlichen Symbol $\top$ erhalten wir den flachen Verband $\mathbb{Z}^{\top}_{\perp} = \mathbb{Z} \cup \{\perp, \top\}$:

Die kleinste obere Schranke von Mengenverbänden ist die Vereinigung dieser Verbände.



Durch das Hinzufügen des Top-Elements erhält man einen vollständigen Verband.
Es ist zu beachten, dass man die selbe Trägermenge (hier $\mathbb{Z}$)
mit unterschiedlichen Ordnungen ausstatten kann.
Es gibt hier z.B $0 \leq \top$, aber keine!!! Beziehung zwischen den Zahlen.

Es gilt:

Satz:

In jedem vollständigen Verband $\mathbb{D}$ besitzt jede Teilmenge
$X \subseteq \mathbb{D}$ eine größte untere Schranke $\bigsqcap X$.

Beweis:

Konstruiere $U = \{u \in \mathbb{D} \mid \forall\, x \in X :\ u \sqsubseteq x\}$.
// die Menge der unteren Schranken von X :-)
Setze: $g := \bigsqcup U$
Behauptung: $g = \bigsqcap X$

(1) g ist eine untere Schranke von X :

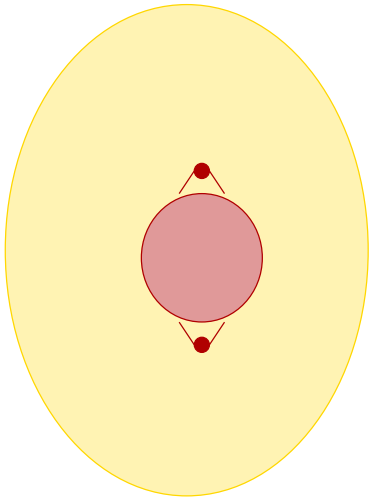
Für $x \in X$ gilt:

$u \sqsubseteq x$ für alle $u \in U$
$\Longrightarrow$ x ist obere Schranke von U
$\Longrightarrow$ $g \sqsubseteq x$:-)

(2) g ist größte untere Schranke von X :

Für jede untere Schranke u von X gilt:

$u \in U$
$\Longrightarrow$ $u \sqsubseteq g$:-))



Ein cl (vollständiger Verband) hat immer ein kleinstes Element $\bot$ *(Bottom) und ein grösstes Element* $\top$ *(Top).*
Per Definition hat jede Teilmenge (/q Eigelb /qq) eine kleinste obere Schrank, dies impliziert, dass sie auch eine grösste untere Schranke hat.

Wir suchen Lösungen für Constraint-Systeme der Form:
Diese Constraint-Systeme sind im eigentlichen Sinn Einschränkungen an die Werte von Variablen

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

wobei:

x_i	Unbekannte	hier:	$\mathcal{A}[u]$
$\mathbb{D}$	Werte	hier:	2^{Expr}
$\sqsubseteq \subseteq \mathbb{D} \times \mathbb{D}$	Ordnungsrelation	hier:	$\supseteq$
$f_i \colon \mathbb{D}^n \to \mathbb{D}$	Bedingung	hier:	...

Unbekannte : (available expressions);
Werte : Menge der möglichen Werte, vollständiger Verband;
Ordnungsrelation : Halbordnungsrelation auf der Menge der Werte;
(Wobei $\bot$ *alle geltenenden Ausdrücke umfasst und*
$\top$ *keinen Ausdruck (leere Menge)*
Bedingung : f_i *Nebenbedingnung für die Variable* i

Constraint für $\mathcal{A}[v]$:

$$\mathcal{A}[v] \quad \subseteq \quad \bigcap \{ [\![k]\!]^\sharp (\mathcal{A}[u]) \mid k = (u, \_, v) \text{ Kante} \}$$

Denn:

$$x \sqsupseteq d_1 \wedge \ldots \wedge x \sqsupseteq d_k \quad \text{gdw.} \quad x \sqsupseteq \bigsqcup\{d_1, \ldots, d_k\} \qquad \text{:-)}$$

Eine Abbildung $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt monoton, falls $f(a) \sqsubseteq f(b)$ für alle $a \sqsubseteq b$.

Beispiele:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ für eine Menge U und $f\,x = (x \cap a) \cup b$.
Offensichtlich ist jedes solche f monoton :-)
Aber z.B : $f\,x = a \setminus x$ *ist nicht monoton.*

(2) $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{Z}$ (mit der Ordnung "$\leq$"). Dann gilt:

- $\mathsf{inc}\,x = x + 1$ ist monoton.
- $\mathsf{dec}\,x = x - 1$ ist monoton.
- $\mathsf{inv}\,x = -x$ ist nicht monoton :-)

Satz:

Sind $f_1 : \mathbb{D}_1 \to \mathbb{D}_2$ und $f_2 : \mathbb{D}_2 \to \mathbb{D}_3$ monoton,
dann ist auch $f_2 \circ f_1 : \mathbb{D}_1 \to \mathbb{D}_3$ monoton :-) *Konkatenation*

Satz:

Ist $\mathbb{D}_2$ ein vollständiger Verband, dann bildet auch die Menge $[\mathbb{D}_1 \to \mathbb{D}_2]$ der monotonen Funktionen $f : \mathbb{D}_1 \to \mathbb{D}_2$ einen vollständigen Verband, wobei

$$f \sqsubseteq g \quad \text{gdw.} \quad f\,x \sqsubseteq g\,x \quad \text{für alle } x \in \mathbb{D}_1$$

Insbesondere ist für $F \subseteq [\mathbb{D}_1 \to \mathbb{D}_2]$,

$$\bigsqcup F = f \quad \text{mit} \quad f\,x = \bigsqcup\{g\,x \mid g \in F\}$$

Für Funktionen $f_i\,x = a_i \cap x \cup b_i$ können wir die Operationen "$\circ$", "$\sqcup$" und "$\sqcap$" explizit angeben:

$$\begin{array}{lcll}
(f_2 \circ f_1)\,x & = & \boxed{a_1 \cap a_2} \cap x \cup \boxed{a_2 \cap b_1 \cup b_2} & \textit{Komposition} \\
(f_1 \sqcup f_2)\,x & = & \boxed{(a_1 \cup a_2)} \cap x \cup \boxed{b_1 \cup b_2} & \textit{Vereinigung} \\
(f_1 \sqcap f_2)\,x & = & \boxed{(a_1 \cup b_1) \cap (a_2 \cup b_2)} \cap x \cup \boxed{b_1 \cap b_2} & \textit{Durchschnitt}
\end{array}$$

Gesucht:

möglichst kleine Lösung für:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.
Vektorschreibweise : $\underline{X} \sqsupseteq F\underline{X}$

Idee:

- Betrachte $F : \mathbb{D}^n \to \mathbb{D}^n$ mit
  $$F(x_1, \ldots, x_n) = (y_1, \ldots, y_n) \quad \text{wobei} \quad y_i = f_i(x_1, \ldots, x_n).$$
- Sind alle f_i monoton, dann auch F :-)
- Wir approximieren sukzessive eine Lösung. Wir konstruieren:
  $$\underline{\bot}, \quad F\,\underline{\bot}, \quad F^2\,\underline{\bot}, \quad F^3\,\underline{\bot}, \quad \ldots$$

Strategie : Iteration von F mit Startpunkt $\bot$ (Bottom)

Hoffnung:

Wir erreichen irgendwann eine Lösung ... ???

Beispiel:

$\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq = \subseteq$

Der Wertebereich ist ein vollständiger Verband, die rechten Seiten sind monoton.
Die Ordnung ist die Untermengenrelation,

$$\begin{aligned} x_1 &\supseteq \{a\} \cup x_3 \\ x_2 &\supseteq x_3 \cap \{a, b\} \\ x_3 &\supseteq x_1 \cup \{c\} \end{aligned}$$

Die Iteration:

Man startet bei $\bot$ (hier leere Menge) und setzt die gefundenen Lösungen immer wieder in das Ungleichungssystem ein, bis diese sich nicht mehr ändern

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	dito
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	
x_3	$\emptyset$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	

Offenbar gilt:

- Gilt $F^k\ \underline{\bot} = F^{k+1}\ \underline{\bot}$, ist eine Lösung gefunden :-)
- $\underline{\bot}, F\ \underline{\bot}, F^2\ \underline{\bot}, \ldots$ bilden eine aufsteigende Kette :

$$\underline{\bot} \quad \sqsubseteq \quad F\ \underline{\bot} \quad \sqsubseteq \quad F^2\ \underline{\bot} \quad \sqsubseteq \quad \ldots$$

- Sind alle aufsteigenden Ketten endlich, gibt es k immer.

Es gibt also immer eine Lösung

Die aufsteigenden Ketten in dem vollständigen Verband sind endlich. Deshalb terminiert die Iteration!
Die zweite Aussage folgt mit vollständiger Induktion:

Anfang: $F^0\ \underline{\bot} = \underline{\bot} \sqsubseteq F^1\ \underline{\bot}$:-)

Schluss: Gelte bereits $F^{i-1}\ \underline{\bot} \sqsubseteq F^i\ \underline{\bot}$. Dann

$$F^i\ \underline{\bot} = F\ (F^{i-1}\ \underline{\bot}) \sqsubseteq F\ (F^i\ \underline{\bot}) = F^{i+1}\ \underline{\bot}$$

da F monoton ist :-)

Fazit:

Wenn $\mathbb{D}$ endlich ist, finden wir mit Sicherheit eine Lösung :-)

Fragen:

1. Gibt es eine kleinste Lösung ?
2. Wenn ja: findet Iteration die kleinste Lösung ??
3. Was, wenn $\mathbb{D}$ nicht endlich ist ???

Satz — Knaster – Tarski

d_0 (der kleinste Fixpunkt) ist die grösste untere Schranke der Menge der Präfixpunkte. Diese Präfixpunkte sind die Menge der Lösungen eines Ungleichungssystems. Somit ist d_0 die kleinste Lösung dieses Ungleichungssystems.

In einem vollständigen Verband $\mathbb{D}$ hat jede monotone Funktion $f : \mathbb{D} \to \mathbb{D}$ einen kleinsten Fixpunkt d_0.

Sei $P = \{d \in \mathbb{D} \mid f\ d \sqsubseteq d\}$ die Menge der Präfixpunkte.
Dann ist $d_0 = \bigsqcap P$.

Bronislaw Knaster (1893-1980), topology

Beweis:

(1) $d_0 \in P$:

$f\,d_0 \sqsubseteq f\,d \sqsubseteq d$ für alle $d \in P$

$\Longrightarrow$ $f\,d_0$ ist untere Schranke von P

$\Longrightarrow$ $f\,d_0 \sqsubseteq d_0$ weil $d_0 = \bigsqcap P$

$\Longrightarrow$ $d_0 \in P$:-)

(2) $f\,d_0 = d_0$:

$f\,d_0 \sqsubseteq d_0$ wegen (1)

$\Longrightarrow$ $f(f\,d_0) \sqsubseteq f\,d_0$ wegen Monotonie von f

$\Longrightarrow$ $f\,d_0 \in P$

$\Longrightarrow$ $d_0 \sqsubseteq f\,d_0$ und die Behauptung folgt :-)

(3) d_0 ist kleinster Fixpunkt:

$f\,d_1 = d_1 \sqsubseteq d_1$ weiterer Fixpunkt

$\Longrightarrow$ $d_1 \in P$

$\Longrightarrow$ $d_0 \sqsubseteq d_1$:-))

Bemerkung:

Der kleinste Fixpunkt d_0 ist in P und untere Schranke :-)

$\Longrightarrow$ d_0 ist der kleinste Wert x mit $x \sqsupseteq f\,x$

Anwendung:

Sei $x_i \sqsupseteq f_i(x_1,\ldots,x_n), \quad i = 1,\ldots,n$ $(*)$
ein Ungleichungssystem, wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

$\Longrightarrow$ kleinste Lösung von $(*)$ $==$ kleinster Fixpunkt von F :-)

Beispiel 1: $\mathbb{D} = 2^U, \quad f\,x = x \cap a \cup b$

f	$f^k\,\bot$	$f^k\,\top$
0	$\emptyset$	$\top$
1	b	$a \cup b$
2	b	$a \cup b$

Beispiel 2: $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Für die Funktion $f\,x = x + 1$ ist:

$$f^i\,\bot = f^i\,0 = i \quad \sqsubset \quad i+1 = f^{i+1}\,\bot$$

$\Longrightarrow$ Die normale Iteration erreicht nie einen Fixpunkt :-(
$\Longrightarrow$ Man benötigt manchmal transfinite Iteration :-)

Satz:

Sei $f : \mathbb{D} \to \mathbb{D}$ monoton und $X \subseteq \mathbb{D}$ die kleinste Menge mit:

(a) $\bot \in X$;
(b) $f\,d \in X$ falls $d \in X$;
(c) $\bigsqcup X_0 \in X$ für alle $X_0 \subseteq X$.

// diese Menge existiert offenbar :-)

Dann ist $d_0 = \bigsqcup X$ der kleinste Fixpunkt von f.

Beweis:

(1) $f\, d_0 \sqsubseteq d_0$ d.h. d_0 ist Präfixpunkt:

$d_0 \in X$ wegen (c)

$\Longrightarrow$ $f\, d_0 \in X$ wegen (b)

$\Longrightarrow$ $f\, d_0 \sqsubseteq d_0$:-)

(2) d_0 ist kleinster Präfixpunkt:

Sei d_1 weiterer Präfixpunkt, d.h. $f\, d_1 \sqsubseteq d_1$.

Dann erfüllt die Menge: $X_1 = \{x \in \mathbb{D} \mid x \sqsubseteq d_1\}$

die Eigenschaften (a), (b) und (c) :-)

$\Longrightarrow$ $X \subseteq X_1$

$\Longrightarrow$ d_1 ist obere Schranke von X

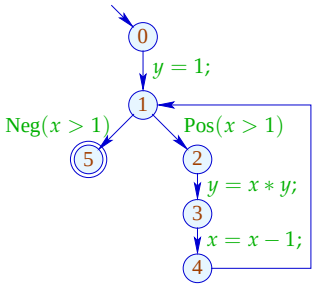
$\Longrightarrow$ $d_0 = \bigsqcup X \sqsubseteq d_1$:-))

Fazit:

Wir können Constraint-Systeme durch Fixpunkt-Iteration lösen,
d.h. durch wiederholtes Einsetzen :-)

Achtung: Naive Fixpunkt-Iteration ist ziemlich ineffizient :-(

Beispiel:



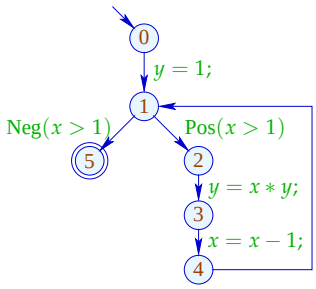
	1	2	3	4	5
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	
1	$\{1, x > 1, x-1\}$	$\{1\}$	$\{1\}$	$\{1\}$	
2	*Expr*	$\{1, x > 1, x-1\}$	$\{1, x > 1\}$	$\{1, x > 1\}$	
3	$\{1, x > 1, x-1\}$	$\{1, x > 1, x-1\}$	$\{1, x > 1, x-1\}$	$\{1, x > 1\}$	dito
4	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$	
5	*Expr*	$\{1, x > 1, x-1\}$	$\{1, x > 1\}$	$\{1, x > 1\}$	

Wenn der Verband die Höhe n hat und das Programm die Grösse p dann hat man $p * n$ Iterationen.

Idee: Round Robin Iteration

Benutze bei der Iteration nicht die Werte der letzten Iteration, sondern die jeweils aktuellen :-)

Beispiel:



	1	2
0	$\emptyset$	
1	$\{1\}$	
2	$\{1, x > 1\}$	
3	$\{1, x > 1\}$	dito
4	$\{1\}$	
5	$\{1, x > 1\}$	

Zusammenfassend ist zu bemerken, dass die normale Fixpunktiteration nicht effizient ist, da zu viele Iterationen benötigt werden.
Bei der Round Robin Iteration nimmt man nun die gerade vorher errechneten Werte und erhöht so die Effizienz..

Der Code für Round Robin Iteration sieht in Java so aus:

```
for (i = 1; i ≤ n; i++) x_i = ⊥;
do {
    finished = true;
    for (i = 1; i ≤ n; i++) {
        new = f_i(x_1, ..., x_n);
        if (!(x_i ⊒ new)) {
            finished = false;
            x_i = x_i ⊔ new;
        }
    }
} while (!finished);
```

Zum Algorithmus:
Die Zuweisung $x_i = x_i \sqcup new;$ *wird gemacht um* x_i *eine sichere obere Schranke des alten Wertes* x_i *zuzuweisen. Falls die rechten Seiten nicht monoton sind, wird dadurch der Algorithmus robuster.*

Es wird somit x_i ein Wert zugewiesen, der eine sichere obere Schranke des alten Wertes ist **und** *desjenigen Wertes den die rechte Seite geliefert hat.*

Worst case - Abschätzung des Algorithmus :
Die Höhe des Verbandes sei h;
Die Anzahl der Variablen sei n;
Dann ist die Anzahl der Iterationen $\leq n * h$.

Verfeinerte Abschätzung:
Spezialfall: Wenn die rechten Seiten von der Form $fx = a \cap x \cup b$ *sind, dann ist die Anzahl der Iterationen maximal n.*
Falls man geeignete Anordungen der Variablen trifft, kann man auch mit weniger Iterationsschritten auskommen. Die Anzahl der Durchläufe hängt im wesentlichen von den Schachtelungstiefen der Schleifen ab.
Allgemein ist zu bemerken, dass es effizientere Algorithmen als den Round Robin-Algorithmus gibt.

Zur Korrektheit:

Wir wollen nun beweisen, dass die Round Robin-Iteration das gleiche Ergebnis liefert wie die normale Fixpunktiteration.
Die Round Robin-Iteration ist quasi eine beschleunigte Fixpunktiteration in der die neu gewonnenen Werte gleich wieder verwendet werden.

Vergleich der RR-Iteration mit der normalen Fixpunkiteration:

Sei $y_i^{(d)}$ die i-te Komponente von $F^d \underline{\perp}$. Sei $x_i^{(d)}$ der Wert von x_i nach der i-ten RR-Iteration.

Man zeigt:
(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)
(2) $x_i^{(d)} \sqsubseteq z_i$ für jede Lösung $(z_1, \ldots, z_n)$:-)
(3) Terminiert RR-Iteration nach d Runden, ist $(x_1^{(d)}, \ldots, x_n^{(d)})$ eine Lösung :-))
Die RR-Iteration terminiert mit der kleinsten Lösung.

Achtung:

Die Effizienz von RR-Iteration hängt von der Anordnung der Variablen ab !!!

Günstig:

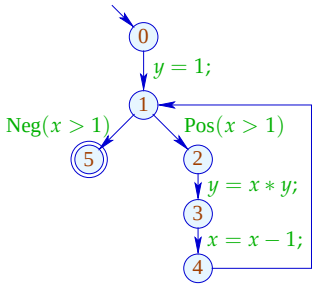
- $\rightarrow$ u vor v, falls $u \rightarrow^* v$;
- $\rightarrow$ *(Topologisches Sortieren; Dies funktioniert nur in zyklenfreien Graphen)*
- $\rightarrow$ Eintrittsbedingung vor Schleifen-Rumpf :-)
- $\rightarrow$ *Zyklen kommen in Schleifen vor;*
  (Deshalb : Zerlegung der Graphen in azyklische Teile und Schleifenköpfe)

Ungünstig:

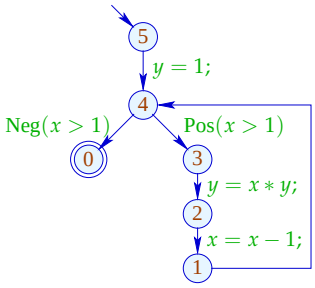
z.B. post-order DFS auf dem CFG, startend von start :-)

Alle nachfolgenden Knoten (Söhne) des Startknotens werden **vor** *dem Startknoten (Wurzel) numeriert. s. Beispiel.*

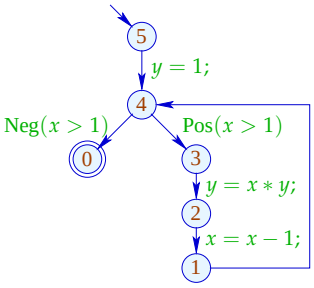
Günstig:



Ungünstig:



Ungünstige Round Robin Iteration:



	1	2	3	4
0	*Expr*	$\{1, x > 1\}$	$\{1, x > 1\}$	
1	$\{1\}$	$\{1\}$	$\{1\}$	
2	$\{1, x - 1, x > 1\}$	$\{1, x - 1, x > 1\}$	$\{1, x > 1\}$	dito
3	*Expr*	$\{1, x > 1\}$	$\{1, x > 1\}$	
4	$\{1\}$	$\{1\}$	$\{1\}$	
5	$\emptyset$	$\emptyset$	$\emptyset$	

$\Longrightarrow$ deutlich weniger effizient :-)

Es sind zwar weniger Iterationen notwendig, als bei einer trivialen Fixpunktiteration, aber durch die ungünstige Anordnung ist diese RR-Iteration weniger effizient.

Durch eine ungünstige Nummerierung der Knoten werden eben nicht die aktuellsten Werte genommen.

... Ende des Exkurses: Vollständige Verbände

Letzte Frage:

Wieso hilft uns eine (oder die kleinste) Lösung des Constraint-Systems weiter ???

Betrachte für einen vollständigen Verband $\mathbb{D}$ Systeme:

$$\begin{array}{lll} \mathcal{I}[start] & \sqsupseteq & d_0 \\ \mathcal{I}[v] & \sqsupseteq & [\![k]\!]^\sharp\,(\mathcal{I}[u]) \qquad k = (u, \_, v) \quad \text{Kante} \end{array}$$

wobei $d_0 \in \mathbb{D}$ und alle $[\![k]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ monoton sind ...

$[\![k]\!]^\sharp$ könnte mal auch als den Effekt der Kante k bezeichnen.

$\Longrightarrow$ monotoner Analyse-Rahmen

Diese Art von Informationsberechnung funktioniert für viele Analyseverfahren.

Jeffrey D. Ullman, Stanford

Gesucht: MOP (Merge Over all Paths)

$$\mathcal{I}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : start \to^* v\}$$

Theorem — Kam, Ullman 1975

Die kleinste Lösung des Constraint-Systems ist zumindest eine sichere Approximation an den MOP. Eine Lösung des Constraint-Systems ist mit Sicherheit eine obere Schranke des MOP.

Sei $\mathcal{I}$ die kleinste Lösung des Constraint-Systems.

Dann gilt: $\mathcal{I}[v] \sqsupseteq \mathcal{I}^*[v]$ für jedes v

Insbesondere: $\mathcal{I}[v] \sqsupseteq [\![\pi]\!]^\sharp\, d_0$ für jedes $\pi : start \to^* v$

Beweis: Induktion nach der Länge von π.

Anfang: $\pi = \epsilon$ (leerer Pfad)

Dann gilt:

$$[\![\pi]\!]^\sharp\, d_0 = [\![\epsilon]\!]^\sharp\, d_0 = d_0 \sqsubseteq \mathcal{I}[start]$$

Schluss: $\pi = \pi' k$ für $k = (u, \_, v)$ Kante.

Dann gilt:

$$\begin{array}{rlll} & [\![\pi']\!]^\sharp\, d_0 & \sqsubseteq\ \mathcal{I}[u] & \text{wegen I.H. für } \pi \\ \Longrightarrow & [\![\pi]\!]^\sharp\, d_0 & =\ [\![k]\!]^\sharp\, ([\![\pi']\!]^\sharp\, d_0) & \\ & & \sqsubseteq\ [\![k]\!]^\sharp\, (\mathcal{I}[u]) & \text{da } [\![k]\!]^\sharp \text{ monoton} \\ & & \sqsubseteq\ \mathcal{I}[v] & \text{da } \mathcal{I} \text{ Lösung :-))} \end{array}$$

Enttäuschung:

Liefern Lösungen des Constraint-Systems nur obere Schranken ???

Antwort:

Im allgemeinen: ja :-(

Es sei denn, alle Funktionen $[\![k]\!]^\sharp$ sind distributiv ... :-)

Der eigentliche MOP ist in vielen Fällen nicht berechenbar. Falls aber alle Kanteneffekte nicht nur monoton, sondern auch distributiv sind, lässt sich der MOP mit dem Constraint-System und der RR-Iteration berechnen.

Falls nicht alle Kanteneffekte distributiv sind, lässt sich nur eine sichere obere Schranke für den MOP mit dem Constraint-System und der RR-Iteration berechnen.

Die Funktion $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt

- distributiv, falls $f\,(\bigsqcup X) = \bigsqcup\{f\,x \mid x \in X\}$ für alle $\emptyset \neq X \subseteq \mathbb{D}$;
- strikt, falls $f\,\bot = \bot$.
- total distributiv, falls f distributiv und strikt ist.

Beispiele:

- $f\,x = x \cap a \cup b$ für $a, b \subseteq U$.

  Striktheit: $f\,\emptyset = a \cap \emptyset \cup b = b \quad = \quad \emptyset$ sofern $b = \emptyset$:-(

  Distributivität:

$$\begin{array}{rcl} f\,(x_1 \cup x_2) & = & a \cap (x_1 \cup x_2) \cup b \\ & = & a \cap x_1 \cup a \cap x_2 \cup b \\ & = & f\,x_1 \cup f\,x_2 \qquad \text{:-)} \end{array}$$

- $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad \mathsf{inc}\,x = x + 1$

  Striktheit: $f\,\bot = \mathsf{inc}\,0 = 1 \quad \neq \quad \bot$:-(

  Distributivität: $f\,(\bigsqcup X) \quad = \quad \bigsqcup\{x + 1 \mid x \in X\}$ für $\emptyset \neq X$:-)

- $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2, \quad \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x_1, x_2) = x_1 + x_2$:

  Striktheit: $f\,\bot = 0 + 0 \quad = \quad 0$:-)

  Distributivität:

$$\begin{array}{rcl} f\,((1,4) \sqcup (4,1)) & = & f\,(4,4) \quad = \quad 8 \\ & \neq & 5 \quad = \quad f\,(1,4) \sqcup f\,(4,1) \qquad \text{:-(} \end{array}$$

Bemerkung:

Distributive Funktionen sind automatisch monoton.

Ist $f : \mathbb{D}_1 \to \mathbb{D}_2$ distributiv, dann auch monoton :-)

Offenbar gilt: $a \sqsubseteq b$ gdw. $a \sqcup b = b$.
Daraus folgt:

$$\begin{array}{rcl} & f\,b & = & f\,(a \sqcup b) \\ & & = & f\,a \sqcup f\,b \\ \Longrightarrow & f\,a & \sqsubseteq & f\,b \qquad \text{:-)} \end{array}$$

Gary A. Kildall (1942-1994).
Hat später am Betriebssystem CP/M und
an GUIs für PCs gearbeitet.

Satz von Kildall:
Sind alle Kanteneffekte distributiv, dann ist der MOP gleich der kleinsten Lösung des Constraint-Systems, unter der Voraussetzung, dass alle Programmpunkte vom Startpunkt aus erreichbar sind.

Annahme: alle v sind von *start* erreichbar.
Dann gilt:

Theorem

Kildall 1972

Sind alle Kanten-Effekte $[\![k]\!]^\sharp$ distributiv, dann ist: $\mathcal{I}^*[v] = \mathcal{I}[v]$ für alle v .

Beweis:

Offenbar genügt es zu zeigen, dass $\mathcal{I}^*$ eine Lösung ist :-)
Wir zeigen, dass $\mathcal{I}^*$ alle Ungleichungen erfüllt :-))

(1) Für *start* zeigen wir:

$$\begin{aligned} \mathcal{I}^*[start] &= \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : start \to^* start\} \\ &\sqsupseteq [\![\epsilon]\!]^\sharp\, d_0 \\ &\sqsupseteq d_0 \qquad \text{:-)} \end{aligned}$$

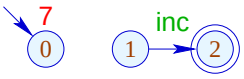
(2) Für jedes $k = (u, \_, v)$ zeigen wir:

$$\begin{aligned}
\mathcal{I}^*[v] &= \bigsqcup\{[\![\pi]\!]^\sharp\, d_0 \mid \pi : \mathit{start} \to^* v\} \\
&\sqsupseteq \bigsqcup\{[\![\pi' k]\!]^\sharp\, d_0 \mid \pi' : \mathit{start} \to^* u\} \\
&= \bigsqcup\{[\![k]\!]^\sharp\,([\![\pi']\!]^\sharp\, d_0) \mid \pi' : \mathit{start} \to^* u\} \\
&= [\![k]\!]^\sharp\,(\bigsqcup\{[\![\pi']\!]^\sharp\, d_0 \mid \pi' : \mathit{start} \to^* u\}) \\
&= [\![k]\!]^\sharp\,(\mathcal{I}^*[u])
\end{aligned}$$

da $\{\pi' \mid \pi' : \mathit{start} \to^* u\}$ nicht-leer ist :-)

Achtung:

- Auf die Erreichbarkeit aller Programm-Punkte können wir nicht verzichten.
  Betrachte:



wobei $\mathbb{D} = \mathbb{N} \cup \{\infty\}$

Dann ist:

$$\begin{array}{lllll}
\mathcal{I}[2] & = & \mathsf{inc}\,0 & = & 1 \\
\mathcal{I}^*[2] & = & \bigsqcup\emptyset & = & 0
\end{array}$$

- Unerreichbare Programmpunkte können wir aber stets wegwerfen :-)

Zusammenfassung und Anwendung:

→ Die Kanteneffekte der Analyse zur Verfügbarkeit von Ausdrücken sind distributiv:

$$\begin{aligned}
(a \cup (x_1 \cap x_2)) \setminus b &= ((a \cup x_1) \cap (a \cup x_2)) \setminus b \\
&= ((a \cup x_1) \setminus b) \cap ((a \cup x_2) \setminus b)
\end{aligned}$$

→ Sind alle Kanteneffekte distributiv, lässt sich der MOP mithilfe des Constraint-Systems und RR-Iteration ausrechnen :-)

→ Sind nicht alle Kanteneffekte distributiv, lässt sich eine sichere obere Schranke für den MOP mithilfe des Constraint-Systems und RR-Iteration berechnen :-)

1.2 Beseitigung überflüssiger Zuweisungen

Rückblick: Bisherige optimierende Transformation: Verfügbarkeit von Ausdrücken:
Falls der Wert eines Ausdrucks bereits ermittelt ist und dieser neu berechnet werden soll, kann dieser evlt. durch Nachschlagen ersetzt werden.
Nun eine weitere Optimierung : Beseitigung von überflüssigen Zuweisungen,

Beispiel:

$$\begin{aligned} 1: &\quad x = y + 2; \\ 2: &\quad y = 5; \\ 3: &\quad x = y + 3; \end{aligned}$$

Der Wert von x an den Programmpunkten 1, 2 wird überschrieben, bevor er benutzt werden kann.
Die Variable x nennen wir deshalb an diesen Programmpunkten tot :-)

Beachte:

- → Zuweisungen an tote Variable können wir uns schenken ;-)
- → Solche Ineffizienzen können u.a. durch andere Transformationen hervorgerufen werden.

Ob eine Variable tot ist hängt von der „Zukunft“dieser Variablen ab. Im Gegensatz bei der Verfügbarkeit von Ausdrücken, hier ist deren „Vergangenheit“relevant.
Es kann sein, dass andere optimierende Transformationen eine Variable „tötet“. Dann können diese toten Variablen (d.h. Zuweisungen) mit dieser Transformation beseitigt werden.

Formale Definition:

Die Variable x heißt lebendig an u entlang des Pfads π , falls sich π zerlegen lässt in $\pi = \pi_1\, \pi_2\, k\, \pi_3$ so dass gilt:
d.h. Der Wert der Variablen spielt möglicherweise während der Programmausführung eine Rolle.

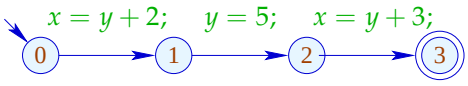
- π_1 erreicht u ;
- k ist eine Benutzung von x ;
- π_2 enthält keine Überschreibung von x.

Die Menge der an einer Kante $k = (\_, lab, \_)$ benutzten bzw. überschriebenen Variablen ist dabei gegeben durch:

lab	benutzt	überschrieben
$;$	$\emptyset$	$\emptyset$
$\text{Pos}\,(e)$	$Vars\,(e)$	$\emptyset$
$\text{Neg}\,(e)$	$Vars\,(e)$	$\emptyset$
$R = e;$	$Vars\,(e)$	$\{R\}$
$R_1 = M[R_2];$	$\{R_2\}$	$\{R_1\}$
$M[R_1] = R_2;$	$\{R_1, R_2\}$	$\emptyset$

Eine Variable x, die nicht lebendig an u entlang π ist, heißt tot an u entlang π.

Beispiel:



Wir bemerken:
Man beginnt am Programmpunkt 3 und geht im Pfad zurück.

	lebendig	tot
0	$\{y\}$	$\{x\}$
1	$\emptyset$	$\{x, y\}$
2	$\{y\}$	$\{x\}$
3	$\emptyset$	$\{x, y\}$

Die Variable x ist lebendig an u falls x lebendig ist an u entlang irgend eines Pfads. Andernfalls ist x tot an u.

Frage:

Wie berechnet man für jedes u die Menge der dort lebendigen/toten Variablen ???

Idee:

Wie bei den verfügbaren Ausdrücken, als man eine Funktion suchte, welche die Menge der verfügbaren Ausdrücke **am Anfang** *der Kante in die Menge der verfügbaren Ausdrücke* **am Ende** *der Kante transformiert, sucht man hier eine Funktion welche die Menge der lebendigen Variablen* **am Ende** *einer Kante in die Menge der lebendigen Variablen* **am Anfang** *der Kante transformiert.*

Definiere für jede Kante $k = (u, \_, v)$ eine Funktion $[\![k]\!]^\sharp$, die die Menge der an v lebendigen Variablen in die Menge der an u lebendigen Variablen transformiert ...
Sei $\mathbb{L} = 2^{Vars}$.
Für $k = (\_, lab, \_)$ definieren wir $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ durch:

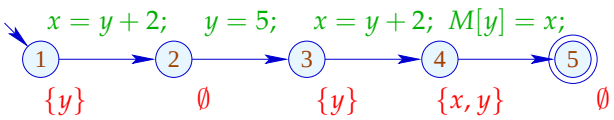
$$\begin{array}{lcl}
[\![;]\!]^\sharp L & = & L \\
[\![\mathrm{Pos}(e)]\!]^\sharp L & = & [\![\mathrm{Neg}(e)]\!]^\sharp L \;=\; L \cup Vars(e) \\
[\![x = e;]\!]^\sharp L & = & (L \backslash \{x\}) \cup Vars(e) \\
[\![R_1 = M[R_2];]\!]^\sharp L & = & (L \backslash \{R_1\}) \cup \{R_2\} \\
[\![M[R_1] = R_2;]\!]^\sharp L & = & L \cup \{R_1, R_2\}
\end{array}$$

$[\![k]\!]^\sharp$ können wir wieder zu Effekten $[\![\pi]\!]^\sharp$ ganzer Pfade $\pi = k_1 \ldots k_r$ fortsetzen durch:

$$[\![\pi]\!]^\sharp = [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_r]\!]^\sharp$$

Dies ist die Komposition der Kanteninformationen die von Vorne nach Hinten durchgereicht werden; Also von k_r bis k_1.

Wir vergewissern uns, dass diese Definitionen vernünftig sind :-)



Die Menge der an u lebendigen Variablen ist dann:

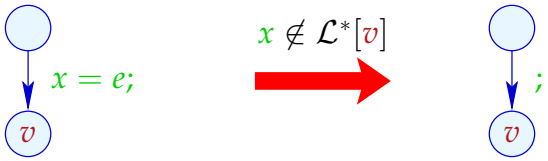
$$\mathcal{L}^*[u] \;=\; \bigcup \{ [\![\pi]\!]^\sharp \emptyset \mid \pi : u \to^* stop \}$$

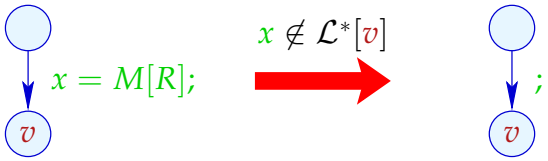
... in Worten:

- Die Pfade starten in u :-)

- x ist lebendig, wenn es nur entlang irgend eines Pfads lebendig ist :-)
  $\Longrightarrow$ Als Halbordnung für $\mathbb{L}$ benötigen wir $\sqsubseteq = \subseteq$.
- Am Programmende ist keine Variable mehr lebendig :-)

Transformation 3:





Zur Korrektheit zeigt man:

→ Korrektheit der Kanten-Effekte: Falls L die Menge der legendigen Variablen am Ende eines Pfads π sind, dann ist $[\![\pi]\!]^\sharp\, L$ die Menge der am Anfang lebendigen Variablen :-)

→ Korrektheit der Transformation auf einem Pfad: Wird auf den Wert einer Variable zugegriffen, ist diese stets lebendig. Der Wert toter Variablen ist darum egal :-)

→ Korrektheit der Transformation: Bei Ausführung des transformierten Programms haben bei jedem Besuch eines Programmpunkts die lebendigen Variablen den gleichen Wert :-))

Berechnung der Mengen $\mathcal{L}^*[u]$:

Durch die Kantentransformation kann man für jede Kante eine entsprechende Ungleichung aufstellen.
Unterschied zu den früheren Ungleichungen (Verfügbare Ausdrücke) *: Auf der linken Seite steht nun der Anfangspunkt der Kante.*
Der Wert am Anfang der Kante wird also in Abhängigkeit des Wertes am Ende der Kante beschrieben.

(1) Aufstellen des Constraint-Systems:

$$\begin{array}{lcll} \mathcal{L}[stop] & \supseteq & \emptyset & \\ \mathcal{L}[u] & \supseteq & [\![k]\!]^\sharp\,(\mathcal{L}[v]) & k = (u, \_, v) \quad \text{Kante} \end{array}$$

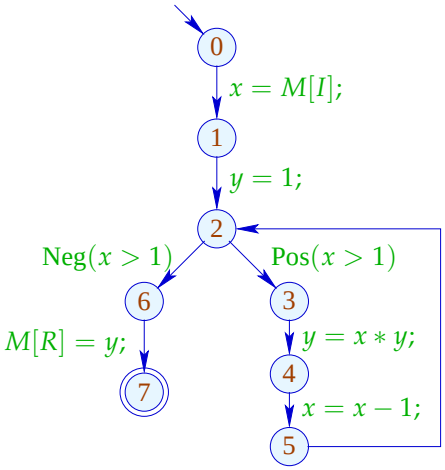
Man hat also nun ein Constraint-System, einen vollständigen Verband (Teilmengenverband), man kennt die Ordnung. Somit kann man dieses Ungleichungssystem mittels Fixpunktiteration (RR-Iteration) lösen. Da der Verband endlich ist, terminiert der Algorithmus.

(2) Lösen des Constraint-Systems mittels RR-Iteration.
Da $\mathbb{L}$ endlich ist, terminiert die Iteration :-)

(3) Die kleinste Lösung $\mathcal{L}$ des Constraint-Systems ist gleich $\mathcal{L}^*$ da alle $[\![k]\!]^\sharp$ distributiv sind :-))
Die kleinste Lösung des Constraint-systems ist also der MOP.

Achtung: Die Information wird rückwärts propagiert !!!

Beispiel: *Fakultät*

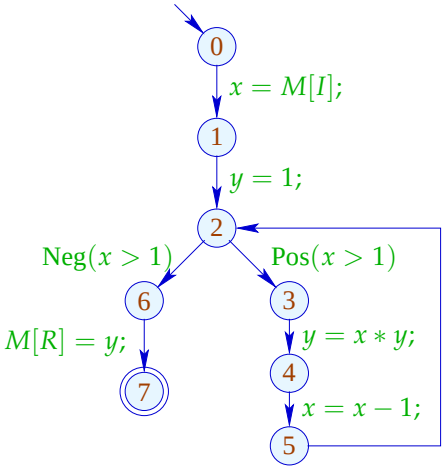


Ungleichungen:

$$
\begin{aligned}
\mathcal{L}[0] &\supseteq (\mathcal{L}[1]\backslash\{x\}) \cup \{I\} \\
\mathcal{L}[1] &\supseteq \mathcal{L}[2]\backslash\{y\} \\
\mathcal{L}[2] &\supseteq (\mathcal{L}[6] \cup \{x\}) \cup (\mathcal{L}[3] \cup \{x\}) \\
\mathcal{L}[3] &\supseteq (\mathcal{L}[4]\backslash\{y\}) \cup \{x, y\} \\
\mathcal{L}[4] &\supseteq (\mathcal{L}[5]\backslash\{x\}) \cup \{x\} \\
\mathcal{L}[5] &\supseteq \mathcal{L}[2] \\
\mathcal{L}[6] &\supseteq \mathcal{L}[7] \cup \{y, R\} \\
\mathcal{L}[7] &\supseteq \emptyset
\end{aligned}
$$

Das Ungleichungssystem entspricht eigentlich dem Kontrollflussgraphen.
In der Praxis wird das Ungleichungssystem nicht aufgebaut.
Man merkt sich für jede Kante die zugehörige Transformation.
Der Analysator arbeitet dann direkt auf dem Kontrollflussgraphen.

Lösungen:
(Anordung der Kanten beim Durchlauf ist wichtig)

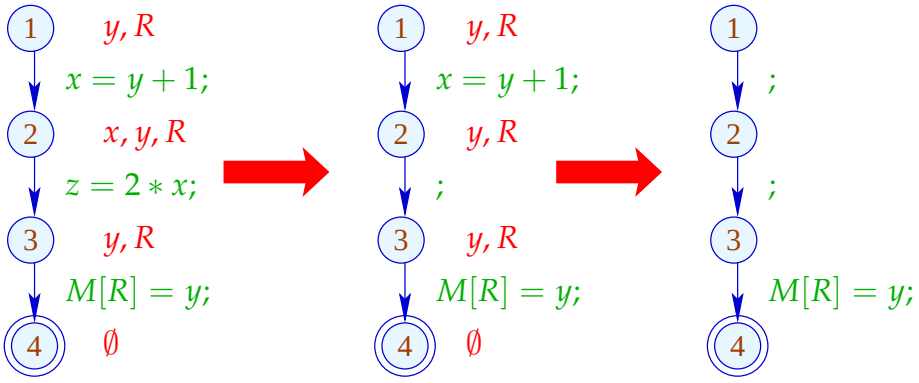


	1	2
7	$\emptyset$	
6	$\{y, R\}$	
2	$\{x, y, R\}$	dito
5	$\{x, y, R\}$	
4	$\{x, y, R\}$	
3	$\{x, y, R\}$	
1	$\{x, R\}$	
0	$\{I, R\}$	

Bei keiner Zuweisung ist die linke Variable tot :-)
Da alle Variablen lebendig sind, kann keine Verbesserung vorgenommen werden.

Achtung:

Beseitigung von Zuweisungen an tote Variablen kann weitere Variablen töten:



Das Programm mehrmals zu analysieren, ist hässlich :-(

Idee: Analysiere echte Lebendigkeit!

x heißt echt lebendig an u entlang eines Pfads π, falls sich π zerlegen lässt in $\pi = \pi_1\, \pi_2\, k\, \pi_3$ so dass gilt:

- π_1 erreicht u;

- k ist eine echte Benutzung von x ;
- π_2 enthält keine Überschreibung von x.

Die Menge der an einer Kante $k = (\_, lab, v)$ echt benutzten Variablen ist gegeben durch:

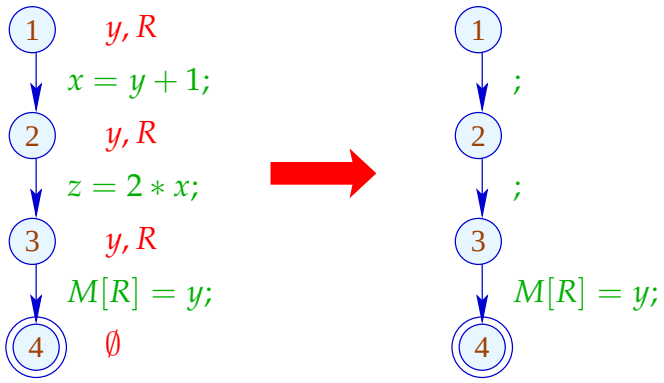
lab	echt benutzt
$;$	$\emptyset$
$\text{Pos}\,(e)$	$Vars\,(e)$
$\text{Neg}\,(e)$	$Vars\,(e)$
$x = e;$	$Vars\,(e)$ $(*)$
$x = M[R];$	$\{R\}$ $(*)$
$M[R_1] = R_2;$	$\{R_1, R_2\}$

$(*)$ – sofern x an v echt lebendig ist :-)

Dies ist der wichtige Unterscheid zur vorhergehenden Definition!

Die Variablen rechts auf der Zuweisung werden nur als echt benutzt betrachtet, sofern die echte Lebendigkeit der Variablen auf der linken Seite gegeben ist.

Beispiel:



z ist am Programmpunkt 2 nicht lebendig (auch nicht echt lebendig). Somit sind die Variablen auf der rechten Seite (x) auch nicht echt benutzt. x ist daher nicht echt benutzt und somit nicht echt lebendig. Am Programmpunkt 1 ist x nicht lebendig propagiert vom Programmpunkt 2 ,daher sind die Variablen hier auf der rechten Seite nicht echt benutzt. Aber das y hat eine echte Benutzung im Programmpunkt

3.

Die Kanten-Effekte:

$$
\begin{array}{lcl}
[\![;]\!]^\sharp\, L & = & L \\
[\![\mathrm{Pos}(e)]\!]^\sharp\, L & = & [\![\mathrm{Neg}(e)]\!]^\sharp\, L \;=\; L \cup \mathit{Vars}(e) \\
[\![x = e;]\!]^\sharp\, L & = & (L \backslash \{x\}) \cup (x \in L)\,?\,\mathit{Vars}(e) : \emptyset \\
[\![R_1 = M[R_2];]\!]^\sharp\, L & = & (L \backslash \{R_1\}) \cup (R_1 \in L)\,?\,\{R_2\} : \emptyset \\
[\![M[R_1] = R_2;]\!]^\sharp\, L & = & L \cup \{R_1, R_2\}
\end{array}
$$

Beachte:

- Die Kanten-Effekte für echt lebendige Variablen sind komplizierter als für lebendige Variablen :-)
- Sie sind aber immer noch distributiv !!

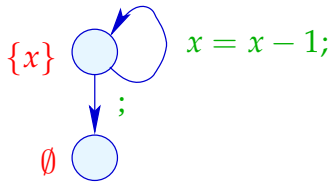
  Dazu betrachten wir für $\mathbb{D} = 2^U$, $f\, y = (u \in y)\,?\,b : \emptyset$ Wir überprüfen:

$$
\begin{array}{rcl}
f\,(y_1 \cup y_2) & = & (u \in y_1 \cup y_2)\,?\,b : \emptyset \\
& = & (u \in y_1 \vee u \in y_2)\,?\,b : \emptyset \\
& = & (u \in y_1)\,?\,b : \emptyset \cup (u \in y_2)\,?\,b : \emptyset \\
& = & f\, y_1 \cup f\, y_2
\end{array}
$$

$\Longrightarrow$ Constraint-System liefert MOP :-))

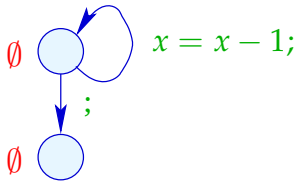
- Echte Lebendigkeit findet mehr überflüssige Zuweisungen als wiederholte Lebendigkeit !!!

Lebendigkeit:



In der Schleife wird x als lebendig identifiziert. Also eine Zuweisung an eine lebendige Variable. Es gäbe also keine Optimierung.

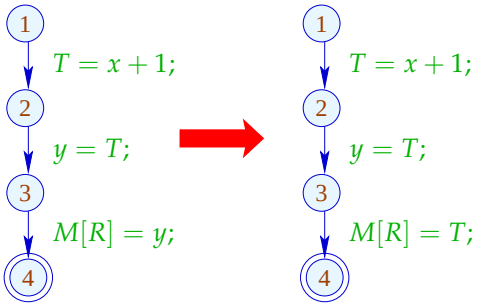
Echte Lebendigkeit:



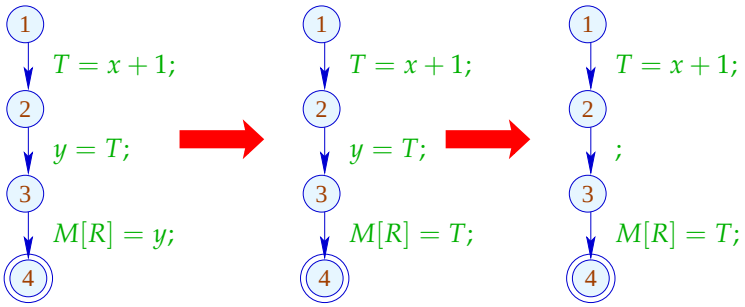
x wird als tot identifiziert. Die Zuweisung wird gestrichen.
Problem: Falls nun die Zuweisung x= x/0; wäre, würde dies einen Fehler versursachen, hätte also einen Seiteneffekt; Auch ein „underflow “durch x=x-1; erzeugt könnte vom Programmierer durchaus gewollt sein und sollte eventuell nicht durch Optimierung entfernt werden.

1.3 Beseitigung überflüssiger Umspeicherungen

Beispiel:



Offenbar ist die Umspeicherung nutzlos :-(
Statt y könnten wir auch T abspeichern :-)



Vorteil: Jetzt ist y tot :-))

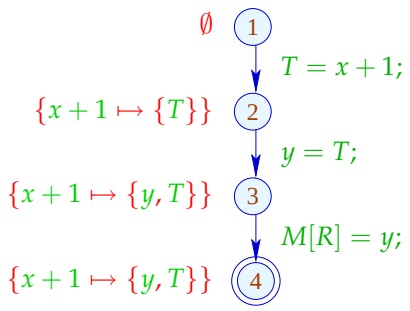
Idee:

Für jeden Ausdruck merken wir uns die Variablen, die gegenwärtig seinen Wert enthalten :-)

Wir benutzen: $\mathbb{V} = Expr \to 2^{Vars}$ und definieren:

$$
\begin{array}{lcl}
[\![;]\!]^\sharp\, V & = & V \\
[\![\mathrm{Pos}(e)]\!]^\sharp\, V & = & [\![\mathrm{Neg}(e)]\!]^\sharp\, V \;=\; V \\
[\![x = e;]\!]^\sharp\, V\, e' & = & \begin{cases} \{x\} & \text{falls} \quad e' = e \\ (V\, e')\backslash\{x\} & \text{sonst} \end{cases} \\
[\![x = y;]\!]^\sharp\, V\, e & = & \begin{cases} (V\, e) \cup \{x\} & \text{falls} \quad y \in V\, e \\ (V\, e)\backslash\{x\} & \text{sonst} \end{cases} \\
[\![x = M[R];]\!]^\sharp\, V\, e & = & (V\, e)\backslash\{x\} \\
[\![M[R_1] = R_2;]\!]^\sharp\, V & = & V
\end{array}
$$

Im Beispiel:



→ Wir propagieren die Information vorwärts :-)
An *start* haben wir $V_0\, e = \emptyset$ für alle e

→ $\sqsubseteq \subseteq \mathbb{V} \times \mathbb{V}$ definieren wir durch:

$$V_1 \sqsubseteq V_2 \quad \text{gdw.} \quad V_1\, e \quad \supseteq \quad V_2\, e \qquad \text{für alle} \quad e$$

Beobachtung:

Man kommt nun mit V_1 und V_2 zu einem Programmpunkt. Als Ordnung wird nun der Durchschnitt dieser Informtionen gewählt, dies ergibt eine sichere Information.

Die neuen Kanten-Effekte sind distributiv:

Dazu zeigen wir, dass die folgenden Funktionen distributiv sind:

(1) $f_1\, V\, e = (V\, e) \backslash \{x\}$

(2) $f_2\, V = V \oplus \{e \mapsto \{x\}\}$

(3) $f_3\, V\, e = (y \in V\, e)\, ?\, (V\, e \cup \{x\}) : ((V\, e) \backslash \{x\})$

Offenbar gilt:

$$\begin{array}{lcl} [\![x = e;]\!]^\sharp & = & f_2 \circ f_1 \\ [\![x = y;]\!]^\sharp & = & f_3 \\ [\![x = M[R];]\!]^\sharp & = & f_1 \end{array}$$

Distributivität ist unter Komposition abgeschlossen. Damit folgt die Behauptung :-))
(1) Für $f\, V\, e = (V\, e) \backslash \{x\}$ gilt:

$$\begin{array}{lcll} f\, (V_1 \sqcup V_2)\, e & = & ((V_1 \sqcup V_2)\, e) \backslash \{x\} & \\ & = & ((V_1\, e) \cap (V_2\, e)) \backslash \{x\} & \\ & = & ((V_1\, e) \backslash \{x\}) \cap ((V_2\, e) \backslash \{x\}) & \\ & = & (f\, V_1\, e) \cap (f\, V_2\, e) & \\ & = & (f\, V_1 \sqcup f\, V_2)\, e & \text{:-)} \end{array}$$

Beobachtung:

Die neuen Kanten-Effekte sind distributiv:

Dazu zeigen wir, dass die folgenden Funktionen distributiv sind:

(1) $f_1\, V\, e = (V\, e) \backslash \{x\}$

(2) $f_2\, V = V \oplus \{e \mapsto \{x\}\}$

(3) $f_3\, V\, e = (y \in V\, e)\ ?\ (V\, e \cup \{x\}) : ((V\, e) \backslash \{x\})$

Offenbar gilt:

$$\begin{array}{lcl} [\![x = e;]\!]^\sharp & = & f_2 \circ f_1 \\ [\![x = y;]\!]^\sharp & = & f_3 \\ [\![x = M[R];]\!]^\sharp & = & f_1 \end{array}$$

Distributivität ist unter Komposition abgeschlossen. Damit folgt die Behauptung :-))

(1) Für $f\, V\, e = (V\, e) \backslash \{x\}$ gilt:

$$\begin{array}{lcll} f\,(V_1 \sqcup V_2)\, e & = & ((V_1 \sqcup V_2)\, e) \backslash \{x\} & \\ & = & ((V_1\, e) \cap (V_2\, e)) \backslash \{x\} & \\ & = & ((V_1\, e) \backslash \{x\}) \cap ((V_2\, e) \backslash \{x\}) & \\ & = & (f\, V_1\, e) \cap (f\, V_2\, e) & \\ & = & (f\, V_1 \sqcup f\, V_2)\, e & \text{:-)} \end{array}$$

(2) Für $f\, V = V \oplus \{e \mapsto a\}$ gilt:

$$\begin{array}{lcll} f\,(V_1 \sqcup V_2)\, e' & = & ((V_1 \sqcup V_2) \oplus \{e \mapsto a\})\, e' & \\ & = & (V_1 \sqcup V_2)\, e' & \\ & = & (f\, V_1 \sqcup f\, V_2)\, e' & \text{sofern} \quad e \neq e' \end{array}$$

$$\begin{array}{lcll} f\,(V_1 \sqcup V_2)\, e & = & ((V_1 \sqcup V_2) \oplus \{e \mapsto a\})\, e & \\ & = & a & \\ & = & ((V_1 \oplus \{e \mapsto a\})\, e) \cap ((V_2 \oplus \{e \mapsto a\})\, e) & \\ & = & (f\, V_1 \sqcup f\, V_2)\, e & \text{:-)} \end{array}$$

(3) Für $f\,V\,e = (y \in V\,e)\,?\,(V\,e \cup \{x\}) : ((V\,e)\backslash\{x\})$ gilt:

$$\begin{aligned}
f\,(V_1 \sqcup V_2)\,e &= (((V_1 \sqcup V_2)\,e)\backslash\{x\}) \cup (y \in (V_1 \sqcup V_2)\,e)\,?\,\{x\} : \emptyset \\
&= ((V_1\,e \cap V_2\,e)\backslash\{x\}) \cup (y \in (V_1\,e \cap V_2\,e))\,?\,\{x\} : \emptyset \\
&= ((V_1\,e \cap V_2\,e)\backslash\{x\}) \cup \\
&\quad ((y \in V_1\,e)\,?\,\{x\} : \emptyset) \cap ((y \in V_2\,e)\,?\,\{x\} : \emptyset) \\
&= (((V_1\,e)\backslash\{x\}) \cup (y \in V_1\,e)\,?\,\{x\} : \emptyset) \cap \\
&\quad (((V_2\,e)\backslash\{x\}) \cup (y \in V_2\,e)\,?\,\{x\} : \emptyset) \\
&= (f\,V_1 \sqcup f\,V_2)\,e \qquad \text{:-)}
\end{aligned}$$

Wir schließen:

→ Lösen des Constraint-Systems liefert die MOP-Lösung :-)

→ Sei $\mathcal{V}$ diese Lösung.

Gilt $x \in \mathcal{V}[u]\,e$, enthält x an u den Wert von e —
welchen wir in T_e abgespeichert haben

$\Longrightarrow$

der Zugriff auf x kann durch Zugriff auf T_e ersetzt werden :-)

Für $V \in \mathbb{V}$ sei V^- die Variablen-Substitution mit:

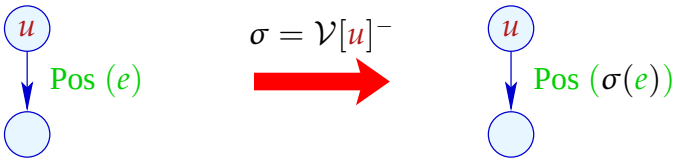
$$V^-\,x = \begin{cases} T_e & \text{falls } x \in V\,e \\ x & \text{sonst} \end{cases}$$

Man nimmt an, dass keine der Variablen die Werte von 2 verschiedenen Ausdrücken enthalten kann.
$V\,e \cap V\,e' = \emptyset$
Die Funktion ist also nicht definiert, falls x in $V\,e$ und $V\,e'$ vorkommt.
Dies ist aber ausgeschlossen, da die Transferfunktionen diese Situation nicht herbeiführen.
Falls eine Variable hinzugefügt wird, wird sie in allen anderen Bildern entfernt.

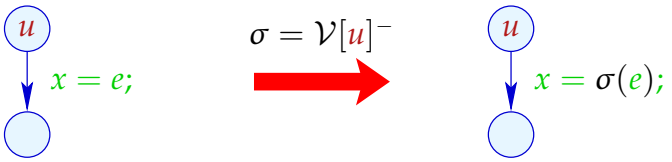
d.h. also:

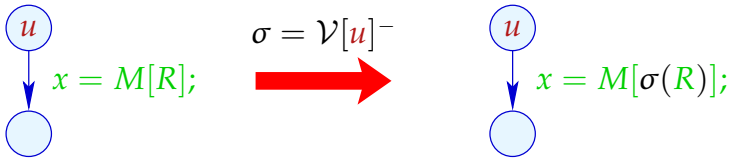
falls $V\,e \cap V\,e' = \emptyset$ für $e \neq e'$. Andernfalls: $V^-\,x = x$:-)

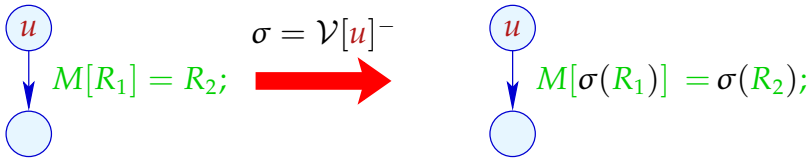
Transformation 4:



... analog für Kanten mit Neg (e)







Vorgehen insgesamt:

Sinnvolle Reihenfolge der uns bisher bekannten Optimierungen bzw. deren Transformationen.

(1) Verfügbarkeit von Ausdrücken: T1 + T2

- \+ verringert arithmetische Operationen
- – fügt überflüssige Umspeicherungen ein

(2) Werte von Variablen: T4

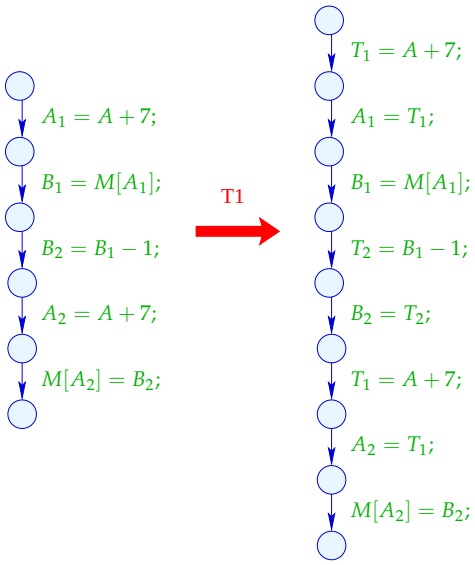
- \+ erzeugt tote Variablen

(3) (wahre) Lebendigkeit von Variablen: T3

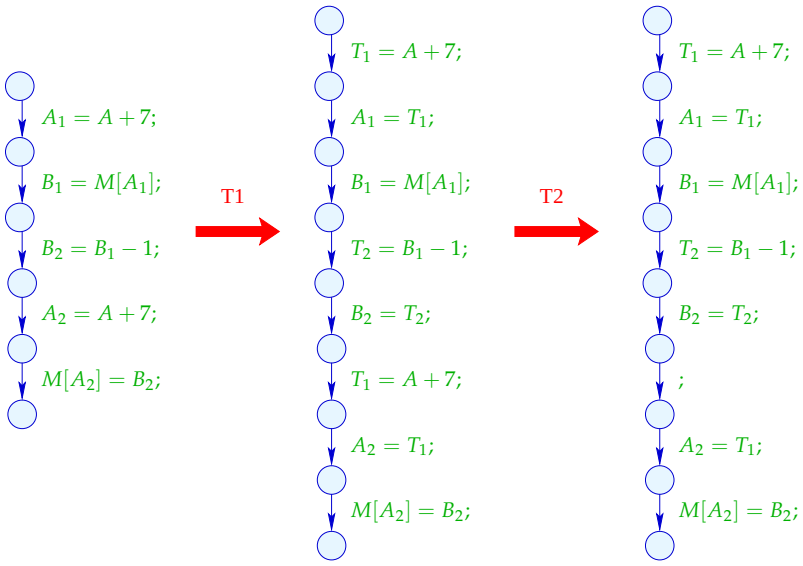
\+ beseitigt Zuweisungen an tote Variablen

Durchführung der Transformationen T1, T2, T4 und T3 am Statement `a[7]--`
Zuerst erfolgt eine Überführung in die Zwischensprache und es ergibt sich :

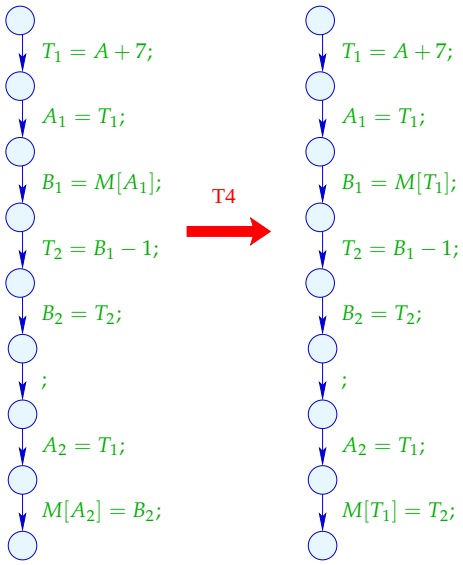
Beispiel: `a[7]--;` T1



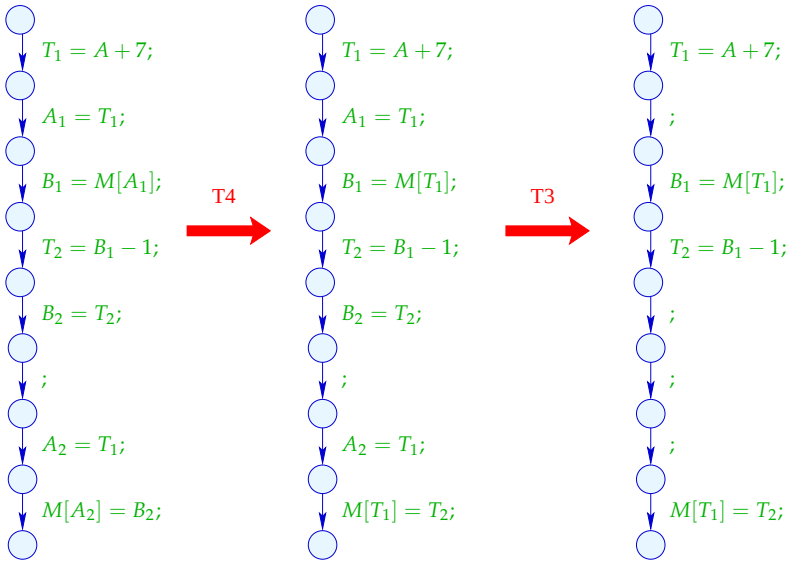
Beispiel: `a[7]--;` T2 *Verfügbare Ausdrücke*



Beispiel: `a[7]--;` T4



Beispiel : `a[7]--;` T3



Man hat nun die Anzahl der Zuweisungen gegenüber zu Beginn der Transformation um eine Zuweisung verringert.
Es zeigt sich, dass nur durch eine sinnvolle Reihenfolge der Transformationen eine wirkliche Optimierung erreicht werden kann.
NOP-Ketten können natürlich gestrichen werden.

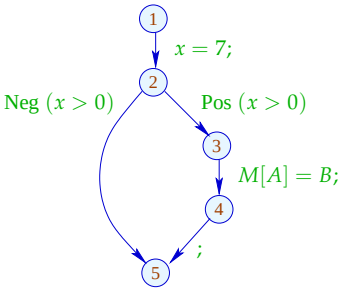
1.4 Konstanten-Propagation

Idee:

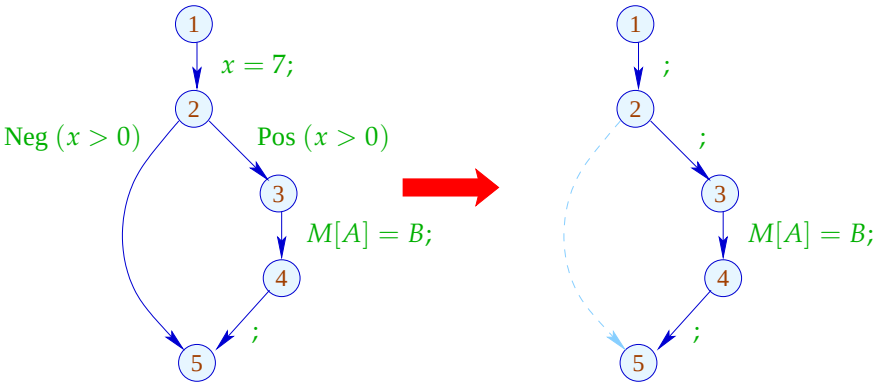
Führe möglichst große Teile des Codes bereits zur Compilezeit aus!

Beispiel:

$x = 7;$
if $(x > 0)$
$\quad M[A] = B;$



Offenbar hat x stets den Wert 7 :-)
Deshalb wird stets der Speicherzugriff durchgeführt :-))



Falls nun z.B. in der Kante (negativer Zweig) ein grösseres Codestück steht, wird dieses nicht mehr benötigt, da es ja nie durchlaufen wird und fällt weg. Man spart also Code.

Wo fallen nun derartige Ineffizienzen an?
z.B.Wenn Konstanten nicht in Variablen abgespeichert werden,
sondern jeweils direkt angegeben werden.
Oder:
JAVA : Hier dürfen Interfaces Konstanten enthalten.
Oder:
z.B. Debugging:
Ein Programm wird für verschiedene Anwendungen programmiert.
Dies wird durch Flags und deren Abfrage gesteuert.

Verallgemeinerung: Partielle Auswertung

Neil D. Jones, DIKU, Kopenhagen

Idee:

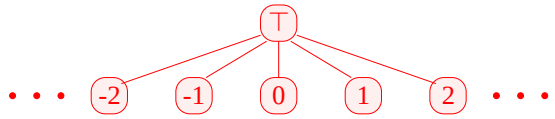
Entwerfe eine Analyse, die für jedes u

- die Werte ermittelt, die Variablen sicher haben;
- mitteilt, ob u überhaupt erreichbar ist :-)

Den vollständigen Verband konstruieren wir in zwei Schritten.

(1) Die möglichen Werte für Variablen:

$$\mathbb{Z}^\top = \mathbb{Z} \cup \{\top\} \qquad \text{mit} \quad x \sqsubseteq y \quad \text{gdw. } y = \top \text{ oder } x = y$$



Achtung: $\mathbb{Z}^\top$ ist selbst kein vollständiger Verband :-(*Kein Bottom-element!*

(2) $\mathbb{D} = (\mathit{Vars} \to \mathbb{Z}^\top)_\bot = (\mathit{Vars} \to \mathbb{Z}^\top) \cup \{\bot\}$

// $\bot$ heißt: "nicht erreichbar" :-))

mit $D_1 \sqsubseteq D_2$ gdw. $\bot = D_1$ oder

$D_1\, x \sqsubseteq D_2\, x \quad (x \in \mathit{Vars})$

Bemerkung: $\mathbb{D}$ ist ein vollständiger Verband :-)

Betrachte dazu $X \subseteq \mathbb{D}$. O.E. $\bot \notin X$.
Dann $X \subseteq \mathit{Vars} \to \mathbb{Z}^\top$.
Ist $X = \emptyset$, dann $\bigsqcup X = \bot \in \mathbb{D}$:-)
Ist $X \neq \emptyset$, dann ist $\bigsqcup X = D$ mit

$$D\, x = \bigsqcup\{f\, x \mid f \in X\} = \begin{cases} z & \text{falls} \quad f\, x = z \quad (f \in X) \\ \top & \text{sonst} \end{cases}$$

:-))

Zu jeder Kante $k = (\_, \mathit{lab}, \_)$ konstruieren wir eine Effekt-Funktion $[\![k]\!]^\sharp = [\![\mathit{lab}]\!]^\sharp : \mathbb{D} \to \mathbb{D}$, die die konkrete Berechnung simuliert.

Offenbar ist $[\![\mathit{lab}]\!]^\sharp \bot = \bot$ für alle lab :-)
Sei darum nun $\bot \neq D \in \mathit{Vars} \to \mathbb{Z}^\top$.

Idee:

- Wir benutzen D , um die Werte von Ausdrücken zu ermitteln.
- Für manche Teilausdrücke erhalten wir $\top$:-)

  $\Longrightarrow$

  Wir müssen die konkreten Operatoren $\Box$ durch abstrakte Operatoren $\Box^\sharp$ ersetzen, die mit $\top$ umgehen können:

$$a \,\Box^\sharp\, b = \begin{cases} \top & \text{falls} \quad a = \top \text{ oder } b = \top \\ a \,\Box\, b & \text{sonst} \end{cases}$$

  Wenn eines oder beide Argumente $\top$ ist, ist natürlich auch das Ergebnis $\top$.
  Ansonsten wird der Ausdruck mit den konkreten Operatoren berechnet.

- Mit den abstrakten Operatoren können wir
  eine abstrakte Ausdrucks-Auswertung definieren:

$$[\![e]\!]^\sharp \;:\; (\mathit{Vars} \to \mathbb{Z}^\top) \to \mathbb{Z}^\top$$

Abstrakte Ausdrucksauswertung ist wie konkrete Ausdrucksauswertung,
aber mit abstrakten Werten und Operatoren.
Hier:

$$\begin{array}{lcl} [\![c]\!]^\sharp\, D & = & c \\ [\![e_1 \,\square\, e_2]\!]^\sharp\, D & = & [\![e_1]\!]^\sharp\, D \,\square^\sharp\, [\![e_2]\!]^\sharp\, D \end{array}$$

... analog für unäre Operatoren :-)

Beispiel: $D = \{x \mapsto 2, y \mapsto \top\}$

$$\begin{array}{lcl} [\![x+7]\!]^\sharp\, D & = & [\![x]\!]^\sharp\, D \,+^\sharp\, [\![7]\!]^\sharp\, D \\ & = & 2 \,+^\sharp\, 7 \\ & = & 9 \\ [\![x-y]\!]^\sharp\, D & = & 2 \,-^\sharp\, \top \\ & = & \top \end{array}$$

Damit erhalten wir für die Kanten-Effekte $[\![\mathit{lab}]\!]^\sharp$:

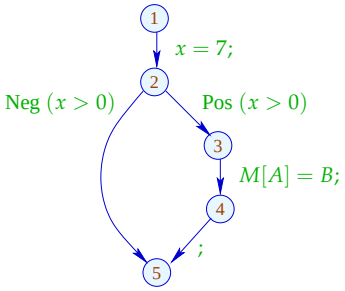
$$\begin{array}{lcl} [\![;]\!]^\sharp\, D & = & D \\ [\![\mathrm{Pos}\,(e)]\!]^\sharp\, D & = & \begin{cases} \bot & \text{falls} \quad 0 = [\![e]\!]^\sharp\, D \\ D & \text{sonst} \end{cases} \\ [\![\mathrm{Neg}\,(e)]\!]^\sharp\, D & = & \begin{cases} D & \text{falls} \quad 0 \sqsubseteq [\![e]\!]^\sharp\, D \\ \bot & \text{sonst} \end{cases} \\ [\![x = e;]\!]^\sharp\, D & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\ [\![x = M[R];]\!]^\sharp\, D & = & D \oplus \{x \mapsto \top\} \\ [\![M[R_1] = R_2;]\!]^\sharp\, D & = & D \end{array}$$

... sofern $D \neq \bot$:-)

Vor dem Programmstart ist nichts über die Belegung der Variablen bekannt.
In manchen Programmiersprachen sind die Variablen mit 0 initialisiert.

An *start* gilt $D_\bot = \{x \mapsto \top \mid x \in \mathit{Vars}\}$.

Somit ergibt sich:



1	$\{x \mapsto \top\}$
2	$\{x \mapsto 7\}$
3	$\{x \mapsto 7\}$
4	$\{x \mapsto 7\}$
5	$\bot \sqcup \{x \mapsto 7\} = \{x \mapsto 7\}$

Die abstrakten Kanten-Effekte $[\![k]\!]^\sharp$ setzen wir wieder zu den Effekten von Pfaden $\pi = k_1 \ldots k_r$ zusammen durch:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp \quad : \mathbb{D} \to \mathbb{D}$$

Idee zur Korrektheit:

Abstrakte Interpretation

Cousot, Cousot 1977

Aufstellen einer Beschreibungsrelation Δ zwischen konkreten Werten und deren Beschreibungen mit:

$$x \,\Delta\, a_1 \quad \wedge \quad a_1 \sqsubseteq a_2 \quad \Longrightarrow \quad x \,\Delta\, a_2$$

Konkretisierung: $\gamma\, a = \{x \mid x \,\Delta\, a\}$

// liefert Menge der beschriebenen Werte :-)

Patrick Cousot, ENS, Paris

Abstrakte Interpretation liefert eine Theorie zum Nachweis der Korrektheit von Programmanalysen. Statt der Interpretation der Syntax mit konkreter operationeller Semantik, wird mit abstrakter operationelle Semantik interpretiert.

Die abstrakten Kanten-Effekte $[\![k]\!]^\sharp$
setzen wir wieder zu den Effekten von Pfaden $\pi = k_1 \ldots k_r$ zusammen durch:

$$[\![\pi]\!]^\sharp = [\![k_r]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp \quad : \mathbb{D} \to \mathbb{D}$$

Idee zur Korrektheit:

Abstrakte Interpretation

Cousot, Cousot 1977

Wir betrachten nun abstrakte Interpretation am Beispiel der Konstantenpropagation.
Der Begriff Abstraktion nach Cousot:
Man wendet eine Abstraktion an, wenn man einen konkreten Wert hat und zu seiner Beschreibung übergehen möchte.
Δ ist eine Relation, die natürlich je nach Beschreibung eine andere sein kann.

Aufstellen einer Beschreibungsrelation Δ zwischen konkreten Werten und deren Beschreibungen mit:

$$x \,\Delta\, a_1 \quad \wedge \quad a_1 \sqsubseteq a_2 \quad \Longrightarrow \quad x \,\Delta\, a_2$$

Wenn man bei den abstrakten Werten, die wir in die Analyse einführen, zu grösseren Werten übergehen, geht Information verloren. Der grösste Wert ist dann Top ($\top$). Hier hat man dann keine Information.

Konkretisierung: $\gamma\, a = \{x \mid x \,\Delta\, a\}$
// liefert Menge der beschriebenen Werte :-)

(1) Werte: $\Delta \subseteq \mathbb{Z} \times \mathbb{Z}^\top$

$$z \,\Delta\, a \quad \text{gdw.} \quad z = a \vee a = \top$$

Konkretisierung:

$$\gamma\, a = \begin{cases} \{a\} & \text{falls} \quad a \sqsubset \top \\ \mathbb{Z} & \text{falls} \quad a = \top \end{cases}$$

(2) Variablenbelegungen: $\Delta \subseteq (\mathit{Vars} \to \mathbb{Z}) \times (\mathit{Vars} \to \mathbb{Z}^\top)_\perp$

$$(\rho, \mu) \,\Delta\, D \quad \text{gdw.} \quad D \neq \perp \;\wedge\; \rho\, x \sqsubseteq D\, x \quad (x \in \mathit{Vars})$$

Konkretisierung:

$$\gamma\, D = \begin{cases} \emptyset & \text{falls} \quad D = \perp \\ \{\rho \mid \forall\, x :\ (\rho\, x) \,\Delta\, (D\, x)\} & \text{sonst} \end{cases}$$

Beispiel: $\{x \mapsto 1, y \mapsto -7\} \;\Delta\; \{x \mapsto \top, y \mapsto -7\}$

(3) Zustände:

$$\Delta \subseteq ((\mathit{Vars} \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})) \times (\mathit{Vars} \to \mathbb{Z}^\top)_\bot$$
$$(\rho, \mu)\ \Delta\ D \qquad \text{gdw.} \qquad \rho\ \Delta\ D$$

Konkretisierung:

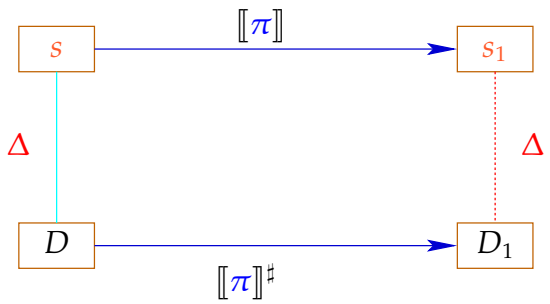
$$\gamma\, D = \begin{cases} \emptyset & \text{falls} \quad D = \bot \\ \{(\rho, \mu) \mid \forall\, x :\ (\rho\, x)\ \Delta\ (D\, x)\} & \text{sonst} \end{cases}$$

Wir zeigen:

Man kann also nun zwischen den konkreten Zuständen s und den abstrakten Werten D eine Beschreibungsrelation Δ herstellen.
Die Relation Δ ist je nach Anwendung natürlich verschieden.

$(*)$ Gilt $s\ \Delta\ D$ und ist $[\![\pi]\!]\, s$ definiert, dann gilt auch:

$$([\![\pi]\!]\, s)\ \Delta\ ([\![\pi]\!]^\sharp\, D)$$



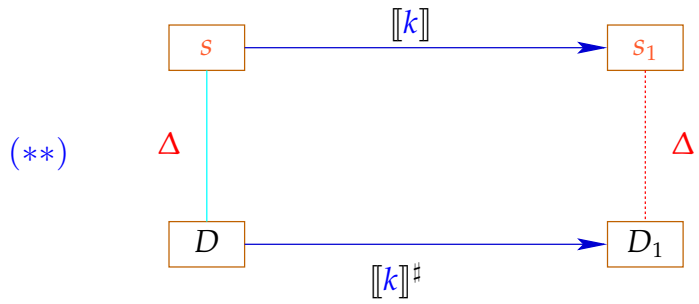
$(*)$ Die abstrakte Semantik simuliert die konkrete :-)
Insbesondere gilt:

$$[\![\pi]\!]\, s \in \gamma\, ([\![\pi]\!]^\sharp\, D)$$

Praktisch heißt das z.B., dass für $D\, x = -7$ gilt:

$$\begin{array}{rrcll} & \rho'\, x & = & -7 & \text{für alle} \quad \rho' \in \gamma\, D \\ \Longrightarrow & \rho_1\, x & = & -7 & \text{für} \quad (\rho_1, \_) = [\![\pi]\!]\, s \end{array}$$

Zum Beweis von $(*)$ zeigen wir für jede Kante k :



Dann folgt $(*)$ mittels Induktion :-)

Zum Beweis von $(**)$ zeigen wir für jeden Ausdruck e :

$(***)$ $([\![e]\!]\,\rho)\ \Delta\ ([\![e]\!]^\sharp\, D)$ sofern nur $\rho\ \Delta\ D$

Zum Beweis von $(***)$ zeigen wir für jeden Operator $\Box$:

$$(x \,\Box\, y)\ \Delta\ (x^\sharp \,\Box^\sharp\, y^\sharp) \qquad \text{sofern} \quad x\ \Delta\ x^\sharp \wedge y\ \Delta\ y^\sharp$$

So hatten wir die Operatoren $\Box^\sharp$ aber gerade definiert :-)

Nun zeigen wir $(**)$ durch Fallunterscheidung nach der Kanten-Beschriftung lab .

Sei $s = (\rho, \mu)\ \Delta\ D$. Insbesondere ist $\bot \neq D\ :\ Vars \to \mathbb{Z}^\top$

Zuweisung $\boxed{x = e;}$:

$$\begin{array}{lcl} \rho_1 & = & \rho \oplus \{x \mapsto [\![e]\!]\,\rho\} \qquad \mu_1 \;=\; \mu \\ D_1 & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \end{array}$$

$$\Longrightarrow \qquad (\rho_1, \mu_1)\ \Delta\ D_1$$

Load $\boxed{x = M[R];}$:

$$\begin{array}{lcl} \rho_1 & = & \rho \oplus \{x \mapsto \mu\,(\rho\, R)\} \qquad \mu_1 \;=\; \mu \\ D_1 & = & D \oplus \{x \mapsto \top\} \end{array}$$

$$\Longrightarrow \qquad (\rho_1, \mu_1)\ \Delta\ D_1$$

Store $\boxed{M[R_1] = R_2;}$:

$$\begin{array}{lcl} \rho_1 & = & \rho \\ D_1 & = & D \end{array} \qquad \mu_1 \;=\; \mu \oplus \{\rho\, R_1 \mapsto \rho\, R_2\}$$

$$\Longrightarrow \qquad (\rho_1, \mu_1)\ \Delta\ D_1$$

Bedingung $\boxed{\mathrm{Neg}(e)}$: $\qquad (\rho_1, \mu_1) = s\,,$ wobei:

$$\begin{array}{rcl} 0 & = & [\![e]\!]\,\rho \\ & \Delta & [\![e]\!]^\sharp\, D \\ \Longrightarrow\ 0 & \sqsubseteq & [\![e]\!]^\sharp\, D \\ \Longrightarrow\ \bot & \neq & D_1 \;=\; D \\ \Longrightarrow & & (\rho_1, \mu_1)\ \Delta\ D_1 \end{array}$$

Bedingung $\boxed{\mathrm{Pos}(e)}$: $\qquad (\rho_1, \mu_1) = s\,,$ wobei:

$$\begin{array}{rcl} 0 & \neq & [\![e]\!]\,\rho \\ & \Delta & [\![e]\!]^\sharp\, D \\ \Longrightarrow\ 0 & \neq & [\![e]\!]^\sharp\, D \\ \Longrightarrow\ \bot & \neq & D_1 \;=\; D \\ \Longrightarrow & & (\rho_1, \mu_1)\ \Delta\ D_1 \end{array}$$

:-)

Wir schließen: Die Behauptung $(*)$ stimmt :-))

Die MOP-Lösung:

Kleinste obere Schranke über alle möglichen Beiträge welche die Pfade liefern können, welche in einem Programmpunkt ankommen.

$$\mathcal{D}^*[v] \;=\; \bigsqcup\{[\![\pi]\!]^\sharp\, D_0 \mid \pi : \mathit{start} \to^* v\}$$

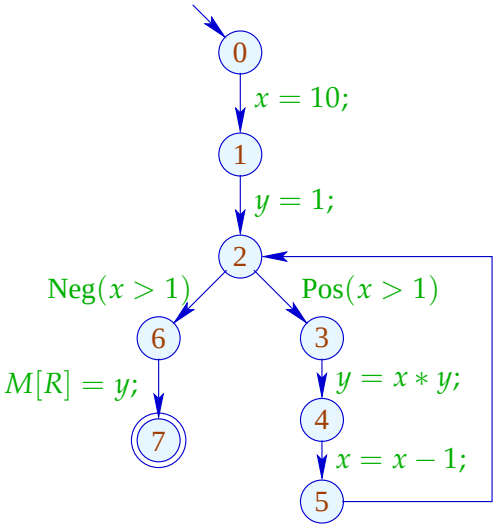
wobei $\qquad D_0\, x = \top \qquad (x \in \mathit{Vars})\,.$

Wegen $(*)$ gilt für alle Anfangszustände s
und alle Berechnungen π , die v erreichen:

$$([\![\pi]\!]\, s)\ \Delta\ (\mathcal{D}^*[v])$$

Zur Approximation des MOP benutzen wir unser Constraint-System :-))

Beispiel: *Das Fakultätsprogramm mit Anfangswerten*



	1		2		3	
	x	y	x	y	x	y
0	$\top$	$\top$	$\top$	$\top$		
1	10	$\top$	10	$\top$		
2	10	1	$\top$	$\top$		
3	10	1	$\top$	$\top$		
4	10	10	$\top$	$\top$	dito	
5	9	10	$\top$	$\top$		
6	$\bot$		$\top$	$\top$		
7	$\bot$		$\top$	$\top$		

Fazit:

Obwohl wir mit konkreten Zahlen rechnen, kriegen wir nicht alles raus :-(
Dafür terminiert die Fixpunkt-Iteration garantiert:
Für n Programmpunkte und m Variablen benötigen wir maximal:
$n \cdot (m + 1)$ Runden :-)

Das Verfahren terminiert i.A. schneller (s. Beispiel)

Achtung:

Die Kanten-Effekte sind nicht distributiv !!!
D.h. Die kleinste Lösung des Ungleichungssystems liefert nur eine Approximation des MOP

Gegenbeispiel: $f = [\![x = x + y;]\!]^\sharp$

Was passiert, wenn man:
Zuerst die Funktion anwendet und dann die kleinste obere Schranke ermittelt, bzw. zuerst die kleinste obere Schranke ermittelt und dann die Funktion anwendet?

$$
\begin{array}{lrcl}
\text{Sei} & D_1 & = & \{x \mapsto 2, y \mapsto 3\} \\
& D_2 & = & \{x \mapsto 3, y \mapsto 2\} \\
\\
\text{Dann} & f\,D_1 \sqcup f\,D_2 & = & \{x \mapsto 5, y \mapsto 3\} \sqcup \{x \mapsto 5, y \mapsto 2\} \\
& & = & \{x \mapsto 5, y \mapsto \top\} \\
& & \neq & \{x \mapsto \top, y \mapsto \top\} \\
& & = & f\,\{x \mapsto \top, y \mapsto \top\} \\
& & = & f\,(D_1 \sqcup D_2)
\end{array}
$$

:-((

Wir schließen:

Die kleinste Lösung $\mathcal{D}$ des Constraint-Systems liefert i.a. nur eine obere Approximation des MOP, d.h.:

$$\mathcal{C}^*[v] \;\sqsubseteq\; \mathcal{C}[v]$$

Als obere Approximation beschreibt $\mathcal{C}[v]$ trotzdem das Ergebnis jeder Berechnung π, die in v endet:

$$(\llbracket \pi \rrbracket\,(\rho, \mu)) \;\Delta\; (\mathcal{C}[v])$$

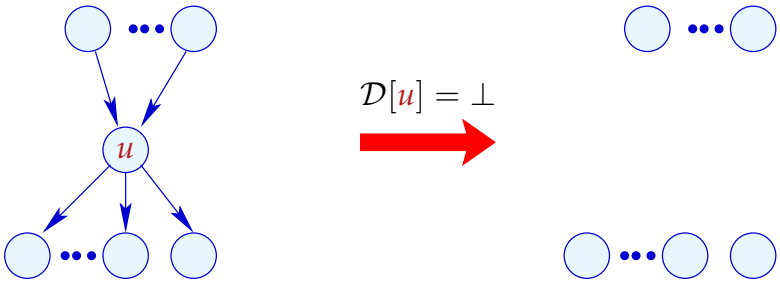
wann immer $\llbracket \pi \rrbracket\,(\rho, \mu)$ definiert ist ;-))

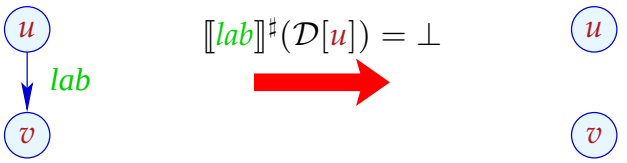
Transformation 5: Beseitigung von totem Code

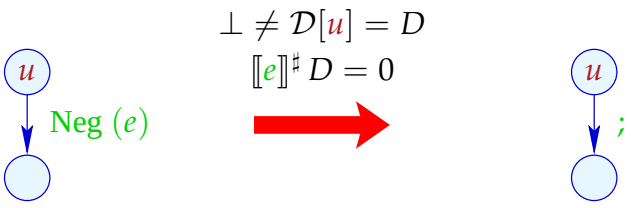
Man erkennt toten Code, falls der Wert an diesem Programmpunkt Bottom $\bot$ ist. Diesen nicht erreichbaren Knoten und alle in ihn ein- und ausgehenden Kanten kann man streichen.
Eine andere Möglichkeit ist, dass die eingehende Kante in den Knoten den Wert Bottom $\bot$ liefert. Hier in unserem Beispiel die Kante vom Knoten u in den Knoten v. Diese Kante kann dann entfernt werden.

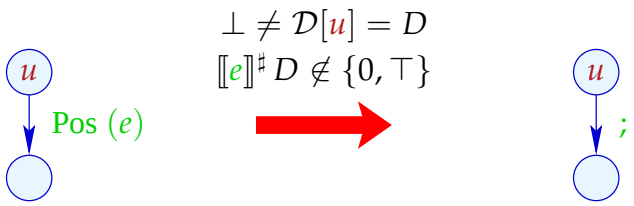
Auch wenn z.B. eine Bedingung immer erfüllt ist, kann diese Bedingung durch NOP ersetzt werden.

Vereinfachte Zuweisungen: Falls der Ausdruck der an eine Variable zugewiesen wird immer den gleichen Wert liefert, kann dieser Wert direkt an die Variable zugewiesen werden.

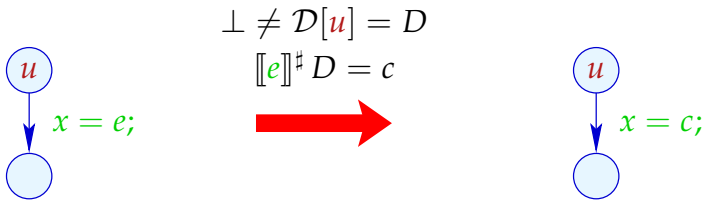








Vereinfachte Zuweisungen



Erweiterungen:

- Statt ganzer rechter Seiten kann man auch Teilausdrücke vereinfachen:

$$x + (3 * y) \xrightarrow{\{x \mapsto \top, y \mapsto 5\}} x + 15$$
$$y * (x + 3) \xrightarrow{\{x \mapsto \top, y \mapsto 5\}} 5 * x + 15$$

  ... und weitere Vereinfachungsregeln anwenden, etwa:

$$\begin{aligned} x * 0 &\Longrightarrow 0 \\ x * 1 &\Longrightarrow x \\ x + 0 &\Longrightarrow x \\ x - 0 &\Longrightarrow x \\ &\dots \end{aligned}$$

- Bisher haben wir die Information von Bedingungen nicht optimal ausgenutzt:

$$\begin{aligned} &\text{if } (x == 7) \\ &\qquad y = x + 3; \end{aligned}$$

Selbst wenn wir den Wert von x vor der if-Abfrage nicht kennen, wissen wir doch, dass bei Betreten des then-Teils x stets den Wert 7 hat :-)

Wir könnten darum definieren:

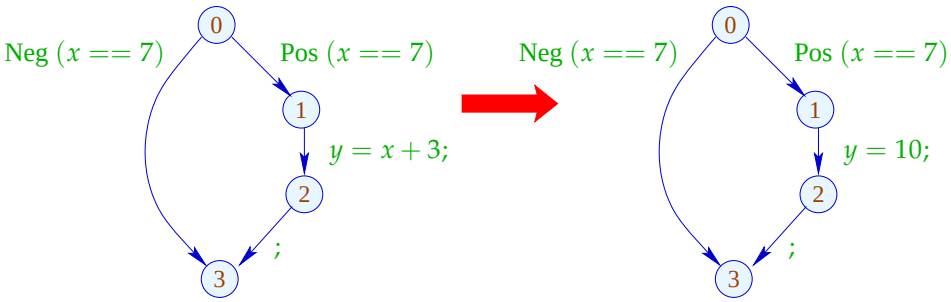
$$[\![\text{Pos}\,(x == e)]\!]^\sharp\, D = \begin{cases} D & \text{falls} \quad [\![x == e]\!]^\sharp\, D = 1 \\ \bot & \text{falls} \quad [\![x == e]\!]^\sharp\, D = 0 \\ D_1 & \text{sonst} \end{cases}$$

wobei

$$D_1 = D \oplus \{x \mapsto (D\, x \sqcap [\![e]\!]^\sharp\, D)\}$$

Analog sieht der Kanteneffekt für $\text{Neg}\,(x \neq e)$ aus :-)

Unser Beispiel:



1.5 Intervall-Analyse

Der exakte Wert einer Variablen ist i.A. nicht bekannt. Man kann aber oft ein sicheres Intervall für den Wert der Variablen angeben.

Beobachtung:

- Programmiererinnen benutzen oft globale Konstanten, um Debug-Code ein oder aus zu schalten

  $\Longrightarrow$

  Konstantenpropagation ist hilfreich :-)

- Im allgemeinen wird aber der Wert von Variablen nicht bekannt sein — möglicherweise aber ein Intervall !!!

Beispiel:

```
for (i = 0; i < 42; i++)
    if (0 ≤ i ∧ i < 42){
        A_1 = A + i;
        M[A_1] = i;
    }
// A Anfangsadresse eines Felds
// if ist Array-Bound-Check
```

Offenbar ist die innere Abfrage überflüssig :-)

Idee 1:

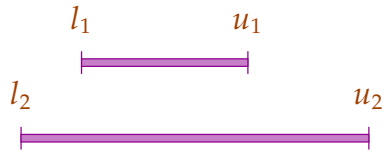
Die Intervalle haben die Grenzen l (lower) und u (upper). Sie entsprechen dem $\mathbb{Z}^\top$.
Diese Intervalle, sowie auch das $\mathbb{Z}^\top$ *sind kein vollständiger Verband, da das Bottom-element fehlt.*

Bestimme für jede Variable x ein (möglichst kleines :-) Intervall für die möglichen Werte:

$$\mathbb{I} = \{[l,u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$$

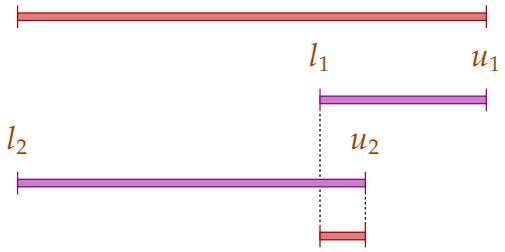
Partielle Ordnung:

$$[l_1,u_1] \sqsubseteq [l_2,u_2] \qquad \text{gdw.} \qquad l_2 \leq l_1 \wedge u_1 \leq u_2$$



Damit:

$$\begin{array}{lcll} [l_1,u_1] \sqcup [l_2,u_2] & = & [l_1 \sqcap l_2, u_1 \sqcup u_2] & \\ [l_1,u_1] \sqcap [l_2,u_2] & = & [l_1 \sqcup l_2, u_1 \sqcap u_2] & \text{sofern } (l_1 \sqcup l_2) \leq (u_1 \sqcap u_2) \end{array}$$



Achtung:

→ $\mathbb{I}$ ist kein vollständiger Verband :-)

→ $\mathbb{I}$ besitzt unendliche aufsteigende Ketten, z.B.

$$[0,0] \sqsubset [0,1] \sqsubset [-1,1] \sqsubset [-1,2] \sqsubset \ldots$$

Man kann ein Ungleichungssystemangeben, aber da es unendlich aufsteigende Ketten gibt, kann dieses System nicht mit der normalen Fixpunktiteration gelöst werden.

Beschreibungsrelation:

$$z \;\Delta\; [l,u] \qquad \text{gdw.} \qquad l \leq z \leq u$$

Konkretisierung:

$$\gamma\,[l,u] = \{z \in \mathbb{Z} \mid l \leq z \leq u\}$$

Beispiel:

$$\begin{aligned} \gamma\,[0,7] &= \{0,\ldots,7\} \\ \gamma\,[0,\infty] &= \{0,1,2,\ldots,\} \end{aligned}$$

Rechnen mit Intervallen: Intervall-Arithmetik :-)

Damit nun die normale Semantik mit Intervallen simuliert werden kann, müssen die entsprechenden abstrakten Operatoren definiert werden.

Addition: *Addition der unteren, bzw. oberen Schranken*

$$\begin{aligned} [l_1,u_1] \;+^\sharp\; [l_2,u_2] &= [l_1+l_2, u_1+u_2] \qquad \text{wobei} \\ -\infty + \_ &= -\infty \\ +\infty + \_ &= +\infty \\ // &\quad -\infty+\infty \quad \text{kommt nicht vor} \quad :\text{-)} \end{aligned}$$

Negation: *Vertauschen der unteren, bzw. oberen Schranke und negieren*

$$-^\sharp [l,u] \;=\; [-u,-l]$$

Multiplikation: *Bildung aller möglichen paarweisen Produkte. Untere Schranke ist das kleinste, obere Schranke ist das grösste Element.*

$$\begin{aligned} [l_1,u_1] *^\sharp [l_2,u_2] &= [a,b] \qquad \text{wobei} \\ a &= l_1l_2 \sqcap l_1u_2 \sqcap u_1l_2 \sqcap u_1u_2 \\ b &= l_1l_2 \sqcup l_1u_2 \sqcup u_1l_2 \sqcup u_1u_2 \end{aligned}$$

Beispiel:

$$\begin{aligned} [0,2] *^\sharp [3,4] &= [0,8] \\ [-1,2] *^\sharp [3,4] &= [-4,8] \\ [-1,2] *^\sharp [-3,4] &= [-6,8] \\ [-1,2] *^\sharp [-4,-3] &= [-8,4] \end{aligned}$$

Division: $[l_1,u_1] /^\sharp [l_2,u_2] = [a,b]$

- Ist 0 nicht im Nenner-Intervall enthalten, sei:

$$\begin{aligned} a &= l_1/l_2 \sqcap l_1/u_2 \sqcap u_1/l_2 \sqcap u_1/u_2 \\ b &= l_1/l_2 \sqcup l_1/u_2 \sqcup u_1/l_2 \sqcup u_1/u_2 \end{aligned}$$

  Paarweise Quotientenbildung und kleinstes Element als untere und grösstes Element als obere Schranke

- Gilt: $l_2 \leq 0 \leq u_2$, setzen wir:

$$[a,b] \;=\; [-\infty,+\infty]$$

Gleichheit:

$$[l_1,u_1] ==^\sharp [l_2,u_2] \;=\; \begin{cases} [1,1] & \text{falls} \quad l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{falls} \quad u_1 < l_2 \vee u_2 < l_1 \\ [0,1] & \text{sonst} \end{cases}$$

Beispiel:

$$\begin{aligned} [42,42] ==^\sharp [42,42] &= [1,1] \\ [0,7] ==^\sharp [0,7] &= [0,1] \\ [1,2] ==^\sharp [3,4] &= [0,0] \end{aligned}$$

Intervallanalyse:
Array-bound-checks sollen beseitigt werden.
Falls man nun ein Intervall berechnen kann, in dem ein Index enthalten ist, und dieses mit dem Intervall der erlaubten Indizes übereinstimmt, kann auf die Überprüfung der Feldgrenzen verzichtet werden.
Operationen aus der konkreten Semantik werden mithilfe von Intervall-Arithmetik modelliert. Man benötigt abstrakte Operatoren, insbesonders Vergleichsoperatoren.

Kleiner:

$$[l_1, u_1] <^\sharp [l_2, u_2] = \begin{cases} [1,1] & \text{falls} \quad u_1 < l_2 \\ [0,0] & \text{falls} \quad u_2 \leq l_1 \\ [0,1] & \text{sonst} \end{cases}$$

Beispiel:

$$\begin{aligned} [1,2] <^\sharp [9,42] &= [1,1] \\ [0,7] <^\sharp [0,7] &= [0,1] \\ [3,4] <^\sharp [1,2] &= [0,0] \end{aligned}$$

Mithilfe von $\mathbb{I}$ konstruieren wir den vollständigen Verband:

$$\mathbb{D}_\mathbb{I} = (\mathit{Vars} \to \mathbb{I})_\perp$$

Beschreibungsrelation:

Die konkrete Variablenbelegung ordnet Zahlen zu, die abstrakte Variablenbelegung Intervalle.
Eine konkrete Variablenbelegung ist in einer abstrakten Variablenbelegung genau dann beschrieben, wenn für jedes x der Wert der darin enthalten ist, in dem Intervall liegt.

$$\rho \ \Delta \ D \qquad \text{gdw.} \qquad D \neq \perp \quad \wedge \quad \forall x \in \mathit{Vars} : (\rho\, x) \ \Delta \ (D\ x)$$

Die abstrakte Ausdrucksauswertung definieren wir analog Konstantenpropagation. Wir finden:

$$([\![e]\!]\ \rho) \ \Delta \ ([\![e]\!]^\sharp\ D) \qquad \text{sofern} \qquad \rho \ \Delta \ D$$

Die Kanteneffekte:

$$
\begin{array}{lcl}
[\![;]\!]^\sharp\, D & = & D \\
[\![x = e;]\!]^\sharp\, D & = & D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\} \\
[\![x = M[R];]\!]^\sharp\, D & = & D \oplus \{x \mapsto \top\} \\
[\![M[R_1] = R_2;]\!]^\sharp\, D & = & D \\
[\![\mathrm{Pos}\,(e)]\!]^\sharp\, D & = & \begin{cases} \bot & \text{falls} \quad [0,0] = [\![e]\!]^\sharp\, D \\ D & \text{sonst} \end{cases} \\
[\![\mathrm{Neg}\,(e)]\!]^\sharp\, D & = & \begin{cases} D & \text{falls} \quad [0,0] \sqsubseteq [\![e]\!]^\sharp\, D \\ \bot & \text{sonst} \end{cases}
\end{array}
$$

... sofern $D \neq \bot$:-)

$\top$ *ist das Intervall von* $-\infty$ *bis* ∞*.. (Keine Information)*

Bessere Ausnutzung von Bedingungen:

Man erhält hier genauere Informationen als bei der Konstantenpropagation.

$$
[\![\mathrm{Pos}\,(e)]\!]^\sharp\, D = \begin{cases} \bot & \text{falls} \quad [0,0] = [\![e]\!]^\sharp\, D \\ D_1 & \text{sonst} \end{cases}
$$

wobei :

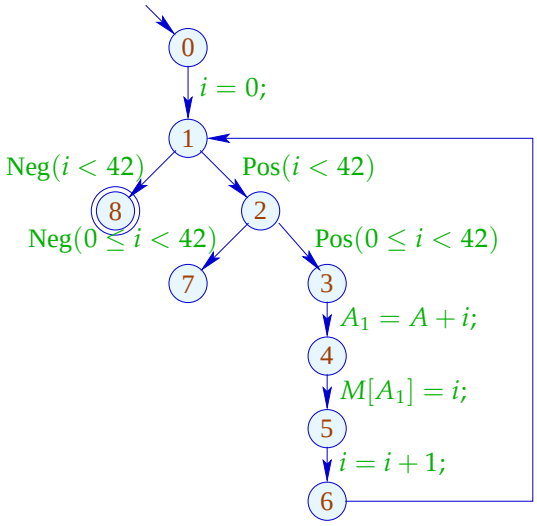
$$
D_1 = \begin{cases} D \oplus \{x \mapsto (D\,x) \sqcap ([\![e_1]\!]^\sharp\, D)\} & \text{falls } e \equiv x == e_1 \\ D \oplus \{x \mapsto (D\,x) \sqcap [-\infty, u]\} & \text{falls } e \equiv x \leq e_1, [\![e_1]\!]^\sharp\, D = [\_, u] \\ D \oplus \{x \mapsto (D\,x) \sqcap [l, \infty]\} & \text{falls } e \equiv x \geq e_1, [\![e_1]\!]^\sharp\, D = [l, \_] \end{cases}
$$

$$
[\![\mathrm{Neg}\,(e)]\!]^\sharp\, D = \begin{cases} \bot & \text{falls} \quad [0,0] \not\sqsubseteq [\![e]\!]^\sharp\, D \\ D_1 & \text{sonst} \end{cases}
$$

wobei :

$$
D_1 = \begin{cases} D \oplus \{x \mapsto (D\,x) \sqcap ([\![e_1]\!]^\sharp\, D)\} & \text{falls } e \equiv x \neq e_1 \\ D \oplus \{x \mapsto (D\,x) \sqcap [-\infty, u]\} & \text{falls } e \equiv x > e_1, [\![e_1]\!]^\sharp\, D = [\_, u] \\ D \oplus \{x \mapsto (D\,x) \sqcap [l, \infty]\} & \text{falls } e \equiv x < e_1, [\![e_1]\!]^\sharp\, D = [l, \_] \end{cases}
$$

Beispiel: *Array-Bound-Check*



	i	
	l	u
0	$-\infty$	$+\infty$
1	0	42
2	0	41
3	0	41
4	0	41
5	0	41
6	1	42
7	$\bot$	
8	42	42

Problem:

In unserem Beispiel terminiert diese Iteration, das muss aber nicht so sein.

→ Die Lösung lässt sich mit RR-Iteration berechnen —
nach ca. 42 Runden :-(

→ Auf manchen Programmen terminiert die Iteration nie :-((

Man sucht nun eine Strategie, die es erlaubt mit Bereichen umzugehen die unendlich aufsteigende Ketten haben.

Idee 1: Widening

- Iteriere beschleunigt — unter Preisgabe von Präzision :-)
- Erlaube nur beschränkt oft die Modifikation eines Werts !!!

... im Beispiel:

- verbiete Updates von Intervall-Grenzen in $\mathbb{Z}$ ...

  $\Longrightarrow$ eine maximale Kette:

$$[3, 17] \sqsubset [3, +\infty] \sqsubset [-\infty, +\infty]$$

Formalisierung dieses Vorgehens:

Die x_i sind hier die Informationen für einen Programmpunkt (die Variablen des Constraint-Systems)
f_i sind die Kanteneffekte.

Sei $x_i \sqsupseteq f_i(x_1, \ldots, x_n)\,, \quad i = 1, \ldots, n$ (1)

ein Ungleichungssystem über $\mathbb{D}$, wobei die f_i nicht notwendigerweise monoton sind. Trotzdem können wir eine akkumulierende Iteration definieren.

Betrachte das Gleichungssystem:

$$x_i = x_i \sqcup f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (2)$$

Wir bilden hier die grösste obere Schranke, indem wir zu dem Wert den das f_i liefert, den alten Wert x_i dazugeben

Offenbar gilt:

(a) $\underline{x}$ ist Lösung von (1) gdw. $\underline{x}$ Lösung von (2) ist.

(b) Die Funktion $G : \mathbb{D}^n \to \mathbb{D}^n$ mit $G(x_1, \ldots, x_n) = (y_1, \ldots, y_n), \quad y_i = x_i \sqcup f_i(x_1, \ldots, x_n)$
ist vergrößernd, d.h. $\underline{x} \sqsubseteq G\,\underline{x}$ für alle $\underline{x} \in \mathbb{D}^n$.

(c) Die Folge $G^k \underline{\bot}, \quad k \geq 0,$ ist eine aufsteigende Kette:

$$\underline{\bot} \sqsubseteq G\underline{\bot} \sqsubseteq \ldots \sqsubseteq G^k\underline{\bot} \sqsubseteq \ldots$$

(d) Gilt $G^k \underline{\bot} = G^{k+1}\underline{\bot} = \underline{y}$ ist $\underline{y}$ eine Lösung von (1).

(e) Hat $\mathbb{D}$ unendliche aufsteigende Ketten, ist uns mit (d) noch nicht viel gedient ...
aber: wir könnten statt Gleichungssystem (2) ein Gleichungssystem:

$$x_i = x_i \sqcup f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (3)$$

betrachten für eine binäre Operation Widening:

$$\sqcup : \mathbb{D}^2 \to \mathbb{D} \quad \text{mit} \quad v_1 \sqcup v_2 \sqsubseteq v_1 \sqcup v_2$$

Dann berechnet (RR)-Iteration für (3) immer noch eine Lösung von (1) :-)

... für die Intervall-Analyse:

Man bestimmt nicht mehr die kleinste obere Schranke, sonderjn „eine“obere Schranke.

- Der vollständige Verband ist: $\mathbb{D}_{\mathbb{I}} = (\mathit{Vars} \to \mathbb{I})_\bot$
- Das Widening $\sqcup$ definieren wir als:

$$\bot \sqcup D = D \sqcup \bot = D \qquad \text{und für} \quad D_1 \neq \bot \neq D_2\text{:}$$

$$(D_1 \sqcup D_2)\,x = (D_1\,x) \sqcup (D_2\,x) \qquad \text{wobei}$$

$$[l_1, u_1] \sqcup [l_2, u_2] = [l, u] \qquad \text{mit}$$

$$l = \begin{cases} l_1 & \text{falls} \quad l_1 \leq l_2 \\ -\infty & \text{sonst} \end{cases}$$

$$u = \begin{cases} u_1 & \text{falls} \quad u_1 \geq u_2 \\ +\infty & \text{sonst} \end{cases}$$

$\Longrightarrow$ $\sqcup$ ist nicht kommutativ !!!

Beispiel:

$$\begin{aligned} [0,2] \sqcup [1,2] &= [0,2] \\ [1,2] \sqcup [0,2] &= [-\infty,2] \\ [1,5] \sqcup [3,7] &= [1,+\infty] \end{aligned}$$

$\rightarrow$ Widening liefert schneller größere Werte.

$\rightarrow$ Es sollte so gewählt werden, dass es die Terminierung der Iteration garantiert :-)

$\rightarrow$ Bei Intervall-Analyse begrenzt es die Anzahl der Iterationen auf:

$$\#Punkte \cdot (1 + 2 \cdot \#Vars)$$

Fazit:

- Um eine Lösung von (1) über einem vollständigen Verband mit unendlichen aufsteigenden Ketten zu bestimmen, definieren wir ein geeignetes Widening und lösen dann (3) :-)

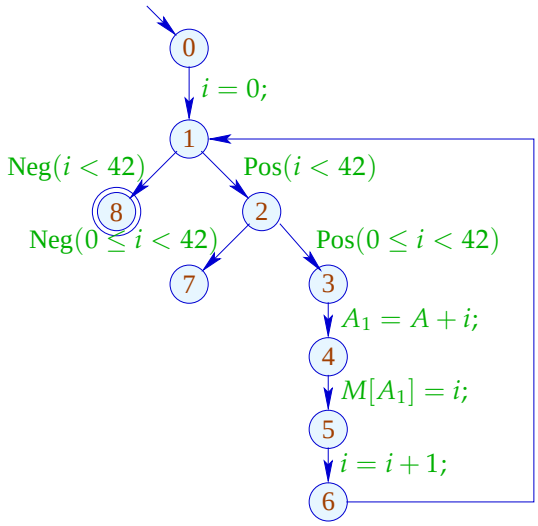
- Achtung: Die Konstruktion geeigneter Widenings ist eine schwarze Kunst !!!

  Oft wählt man $\sqcup$ ganz pragmatisch dynamisch während der Iteration, so dass

  $\rightarrow$ die abstrakten Werte nicht zu kompliziert werden;

  $\rightarrow$ die Anzahl der Updates fest beschränkt bleibt ...

Unser Beispiel:



	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$		
1	0	0	0	$+\infty$		
2	0	0	0	$+\infty$		
3	0	0	0	$+\infty$		
4	0	0	0	$+\infty$	dito	
5	0	0	0	$+\infty$		
6	1	1	1	$+\infty$		
7	$\bot$		42	$+\infty$		
8	$\bot$		42	$+\infty$		

... offenbar ist das Ergebnis enttäuschend :-(

Alle oberen Schranken gehen verloren. Der Algorithmus liefert ein ungenaues unbefriedigendes Ergebnis.

Idee 2:

Man wird nun versuchen, Widening so anzuwenden, dass eine sichere Terminierung erreicht wird. Man wendet Widening dann nur noch an manchen geeigneten Stellen an.

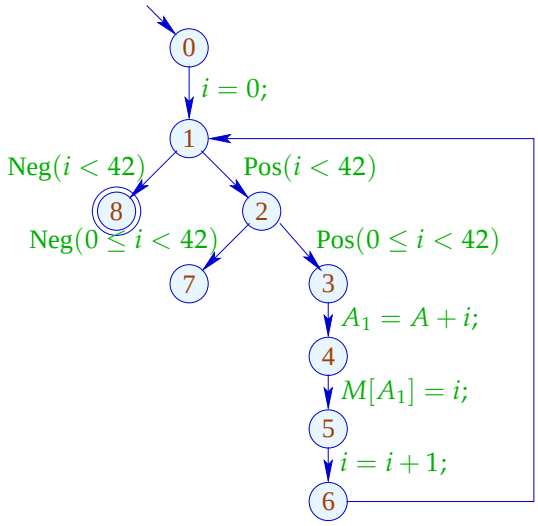
Eigentlich reicht es, die Beschleunigung mittels $\sqcup$ nur an genügend vielen Stellen anzuwenden!

Eine Menge I heißt Loop Separator (Kreis-Trenner), falls jeder Kreis mindestens einen Punkt aus I enthält :-)

Nur an diesen Schleifenseparatoren wird nun das Widening angewandt an anderen Punkten nicht!

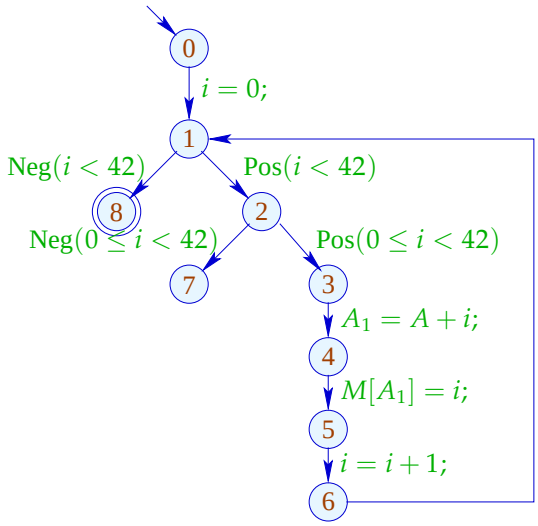
Wenden wir Widening nur an den Punkten aus einer solchen Menge I, terminiert RR-Iteration immer noch !!!

In unserem Beispiel:



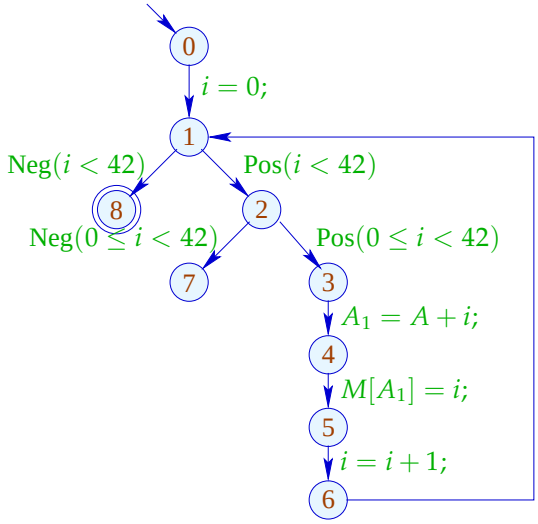
$I_1 = \{1\}$ oder auch:
$I_2 = \{2\}$ oder auch:
$I_3 = \{3\}$

Die Analyse mit $I = \{1\}$:



	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$		
1	0	0	0	$+\infty$		
2	0	0	0	41		
3	0	0	0	41		
4	0	0	0	41	dito	
5	0	0	0	41		
6	1	1	1	42		
7	$\bot$		$\bot$			
8	$\bot$		42	$+\infty$		

Die Analyse mit $I = \{2\}$:



	1		2		3	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$		
1	0	0	0	42		
2	0	0	0	$+\infty$		
3	0	0	0	41		
4	0	0	0	41	dito	
5	0	0	0	41		
6	1	1	1	42		
7	$\bot$		42	$+\infty$		
8	$\bot$		42	42		

Diskussion:

Man sieht·
Es ist von grösster Bedeutung welcher Knoten nun als Loop-Separator gewählt und hier dann das Widening angewandt wird.

- Beide Analysen-Läufe berechnen interessante Informationen :-)
- Der Lauf mit $I = \{2\}$ belegt, dass nach Verlassen der Schleife stets $i = 42$ gilt.

- Nur der Lauf mit $I = \{1\}$ belegt aber, dass der äußere Test den inneren überflüssig macht :-(

Wie findet man einen geeigneten Loop Separator I ???

Idee 3: Narrowing

Eine ungenaue Lösung, die durch widening verursacht wird, wird verbessert, indem man f auf sie anwendet, somit bekommt man eine absteigende Folge und die Lösung wird kleiner und somit besser.

Sei $\underline{x}$ irgend eine Lösung von (1), d.h.

$$x_i \sqsupseteq f_i\,\underline{x}\ , \qquad i = 1, \ldots, n$$

Dann gilt für monotone f_i,

$$\underline{x} \sqsupseteq F\,\underline{x} \sqsupseteq F^2\,\underline{x} \sqsupseteq \ldots \sqsupseteq F^k\,\underline{x} \sqsupseteq \ldots$$

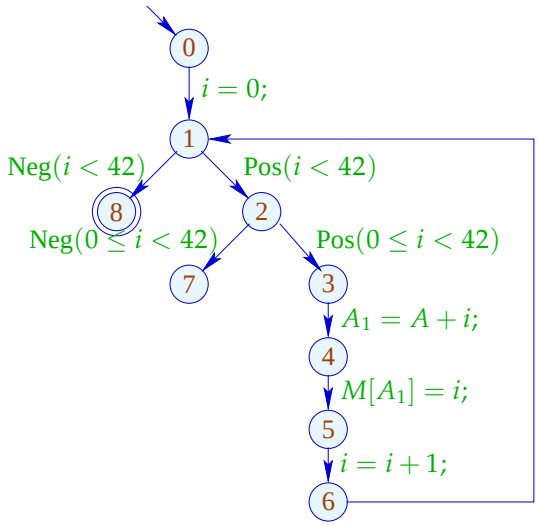
// Narrowing Iteration

Jeder der Tupel $F^k\,\underline{x}$ ist eine Lösung von (1) :-)
$\Longrightarrow$
Terminierung ist kein Problem mehr:
wir stoppen, wenn wir keine Lust mehr haben :-))
// Analoges gilt für RR-Iteration.

Fixpunktapproximation:
Der vollständige Verband hat keine endliche Höhe. Es gibt aufsteigende Ketten von unendlicher Länge.
Die Fixpunktiteration terminiert demnach normalerweise nicht.
Abhilfe: Durch Widening:
Beschleunigung der Fixpunktiteration, indem man zu grösseren Werten übergeht.
Aber das Ergebnis bei der Intervallanalyse ist unbefriedigend.
Man verwirft eventuell die Grenzen und bekommt $\bot$ oder $\top$, (Keine Information).

Durch eine anschliessende zweite Iteration können die Ergebnisse nachträglich wieder verbessert werden.
Abhilfe Narrowing:
Es wurde bereits eine Lösung gefunden, die grösser als die kleinste Lösung ist. Wenn man nun auf das x nochmals die rechten Seiten anwendet, bekommt man möglicherweise kleinere, bessere Werte.
Dieses Narrowing funktioniert nur, wenn die rechten Seiten monoton sind!

Narrowing Iteration im Beispiel:



	0		1		2	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$	0	42
2	0	$+\infty$	0	41	0	41
3	0	$+\infty$	0	41	0	41
4	0	$+\infty$	0	41	0	41
5	0	$+\infty$	0	41	0	41
6	1	$+\infty$	1	42	1	42
7	42	$+\infty$	$\bot$		$\bot$	
8	42	$+\infty$	42	$+\infty$	42	42

Diskussion:

→ Wir beginnen mit einer sicheren Approximation.

→ Wir finden, dass die innere Abfrage redundant ist :-)

→ Wir finden, dass nach der Iteration gilt: $i = 42$:-))

→ Dazu war nicht erforderlich, einen optimalen Loop Separator zu berechnen :-)))

Letzte Frage:

Rechnen mit Intervallen:
Ist es möglich, dass es unendlich absteigende Ketten gibt?
Eben Intervalle, die immer kleiner werden?
Dies ist dann möglich, wenn die obere Grenze ∞ und die untere Grenze von $-\infty$ mit negativen Zahlen nach oben geht.
z.B. Iteration : $[-\infty, \infty]$, $[-100, \infty]$. $[-99, \infty]$....

Müssen wir hinnehmen, dass Narrowing möglicherweise nicht terminiert ???

4. Idee: Beschleunigtes Narrowing

Nehmen wir an, wir hätten eine Lösung $\underline{x} = (x_1, \ldots, x_n)$ des Constraint-Systems:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (1)$$

Dann schreiben betrachten wir das Gleichungssystem:

$$x_i = x_i \sqcap f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (4)$$

Offenbar gilt für monotone f_i : $H^k \underline{x} = F^k \underline{x}$:-)

wobei $H(x_1,\ldots,x_n) = (y_1,\ldots,y_n)\,, \quad y_i = x_i \sqcap f_i(x_1,\ldots,x_n)$.

In (4) ersetzen wir $\sqcap$ durch den neuen Operator $\boxminus$ mit:

$$a_1 \sqcap a_2 \;\sqsubseteq\; a_1 \boxminus a_2 \;\sqsubseteq\; a_1$$

... für die Intervall-Analyse:

Man verbessert nun nur einmal die Intervalgrenzen zu einem richtigen Wert, falls diese $-\infty$ bzw. ∞ sind. Dies sind also zwei Änderungen pro Variable

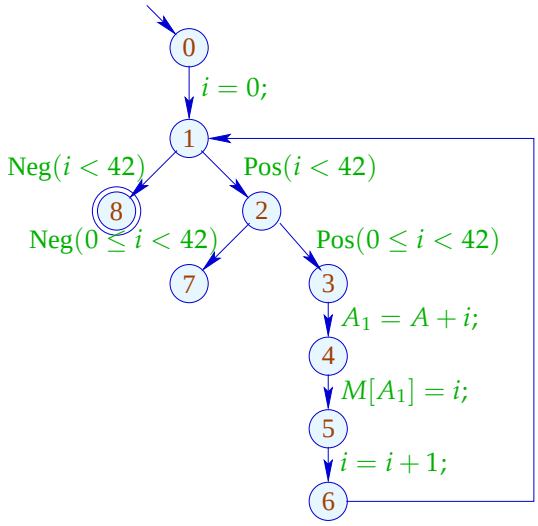
Wir konservieren endliche Intervall-Grenzen :-)

Deshalb $\bot \boxminus D = D \boxminus \bot = \bot$ und für $D_1 \neq \bot \neq D_2$:

$$\begin{aligned}
(D_1 \boxminus D_2)\, x &= (D_1\, x) \boxminus (D_2\, x) \qquad \text{wobei}\\
[l_1,u_1] \boxminus [l_2,u_2] &= [l,u] \qquad \text{mit}\\
l &= \begin{cases} l_2 & \text{falls} \quad l_1 = -\infty \\ l_1 & \text{sonst} \end{cases}\\
u &= \begin{cases} u_2 & \text{falls} \quad u_1 = \infty \\ u_1 & \text{sonst} \end{cases}
\end{aligned}$$

$\Longrightarrow$ $\boxminus$ ist nicht kommutativ !!!

Beschleunigtes Narrowing im Beispiel:



	0		1		2	
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$	0	42
2	0	$+\infty$	0	41	0	41
3	0	$+\infty$	0	41	0	41
4	0	$+\infty$	0	41	0	41
5	0	$+\infty$	0	41	0	41
6	1	$+\infty$	1	42	1	42
7	42	$+\infty$	$\bot$		$\bot$	
8	42	$+\infty$	42	$+\infty$	42	42

Diskussion:

- → Achtung: Widening liefert für nicht-monotone f_i eine Lösung.
  Narrowing liefert dagegen nur für monotone f_i eine Lösung!!
- → Das beschleunigte Narrowing liefert (im Beispiel) das richtige Ergebnis :-)
- → Erlaubt der neue Operator $\sqcap$ nur endlich viele Verbesserungen bei jedem Wert, kann Narrowing bis zur Stabilisierung durchgeführt werden.
- → Für die Intervall-Analyse sind das maximal

$$\#Punkte \cdot (1 + 2 \cdot \#Vars)$$

Für die Analyselaufzeit ergibt sich somit ein Faktor 2. Dies ist akzeptabel.

Historie:
Es wurde schon bei der Entstehung der abstrakten Interpretation (Cousot) die Programmanalyse mit nicht endlichen Bereichen behandelt. Der daraus resultierenden Nichtterminierung dieser Iterationen wurde durch Widening und Narrowing begegnet.

1.6 Pointer-Analyse

Es gibt ein sehr grosses Spektrum an Arbeiten über diese Analyse.
In der Vorlesung wird diese Thematik nicht sehr vertieft.
Es werden nur Grundansätze betrachtet.

Fragen:

Man unterscheidet zwei Arten von Aliases:
Je nach Programmlauf könnten ja die Variablen x und y die gleichen, bzw. verschiedene Adressen belegen. (May Alias)

Bei definitiver Gleichheit (Must Alias)

- → Sind zwei Adressen möglicherweise gleich? — May Alias
- → Sind zwei Adressen definitiv gleich? — Must Alias

$\Longrightarrow$ Alias-Analyse

Die bisherigen Analysen ohne Alias-Information:

(1) Verfügbare Ausdrücke:

- Erweitere die Menge $Expr$ der Ausdrücke um die vorkommenden Loads $M[R]$.
- Erweitere die Kanten-Effekte:

$$\begin{array}{lcl} [\![x = e;]\!]^\sharp\, A & = & (A \cup \{e\}) \backslash Expr_x \\ [\![x = M[R];]\!]^\sharp\, A & = & (A \cup \{M[R]\}) \backslash Expr_x \\ [\![M[R] = x;]\!]^\sharp\, A & = & A \backslash Loads \end{array}$$

Loads sind also an einem Programmpunkt nur verfügbar, wenn zwischendurch kein schreibender Zugriff auf den Speicher erfolgt ist.
Ein Store beseitigt also die Verfügbarkeit aller Loads.

(2) Werte von Variablen:

- Erweitere die Menge $Expr$ der Ausdrücke um die vorkommenden Loads $M[R]$.
- Erweitere die Kanten-Effekte:

$$[\![x = M[R];]\!]^\sharp\, V\, e = \begin{cases} \{x\} & \text{falls} \quad e = M[R] \\ V\, e \backslash \{x\} & \text{sonst} \end{cases}$$

$$[\![M[R] = x;]\!]^\sharp\, V\, e = \begin{cases} \emptyset & \text{falls} \quad e = M[R'] \\ V\, e & \text{sonst} \end{cases}$$

(3) Konstantenpropagation:

- Erweitere den abstrakten Zustand um einen abstrakten Speicher M
- Führe Speicher-Operationen mit bekannten Adressen aus!

$$
\begin{aligned}
[\![x = M[R];]\!]^\sharp\,(D,M) &= \begin{cases} (D \oplus \{x \mapsto M\,a\}, M) & \text{falls} \quad D\,R = a \sqsubset \top \\ (D \oplus \{x \mapsto \top\}, M) & \text{sonst} \end{cases} \\
[\![M[R] = x;]\!]^\sharp\,(D,M) &= \begin{cases} (D, M \oplus \{a \mapsto D\,x\}) & \text{falls} \quad D\,R = a \sqsubset \top \\ (D, \underline{\top}) & \text{sonst} \end{cases} \qquad \text{wobei} \\
\underline{\top}\,a &= \top \qquad (a \in \mathbb{N})
\end{aligned}
$$

Problem :
$M : \mathbb{N} \longrightarrow \mathbb{Z}^\top$
Man bekommt eine Konstantenpropagation mit einem vollständigen Verband der unendlich aufsteigende Ketten hat.
Man könnte also im Prinzip die Konstantenpropagation erweitern, mit dem Problem, dass man bei solchen Store-Operationen den Wert $\top$ bekommt und somit den gesamten Speicherzustand verliert.
Konstantenpropagation könnte durch Einführung von Intervallen verbessert werden.

Probleme:

- Adressen sind aus $\mathbb{N}$:-(
- Exakte Adressen sind zur Compilezeit selten bekannt :-(
  Bei der realen Codegenerierung werden i.A. niemals absolute Adressen vergeben, sondern immer relative Adressen in Bezug zu einem fiktiven Anfangspunkt, da der Code verschiebbar ist. Man hat also eine Basisadresse die unbekannt ist.
- Am selben Programmpunkt wird i.a. auf mehrere Adressen zugegriffen ...
- Abspeichern an unbekannter Adresse zerstört alle Information M :-(

$\Longrightarrow$ Konstanten-Propagation versagt :-(
$\Longrightarrow$ Speicherzugriffe/Pointer zerstören Präzision :-(

Vereinfachung:

Man betrachtet nur JAVA-Style Pointer.
Diese zeigen auf ein Record mit Feldern von konstanter Grösse.
Nur innerhalb dieses Bereichs kann durch Selektion mithilfe von Komponenten modelliert werden.

- Wir betrachten Pointer auf Strukturen mit Komponenten a, b :-)
- Wir verzichten auf Wohl-Getyptheit dieser Komponenten.
- Neue Statements: *Vorher : Speicherzugriff als :* $M[R]$

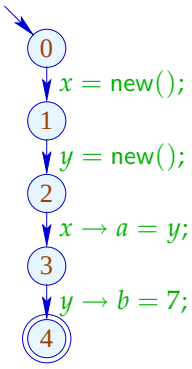
$x = \mathsf{new}();$ // Allokation eines neuen Paars
$x = R \rightarrow a;$ // Laden einer Komponente
$R \rightarrow a = x;$ // Setzen einer Komponente

- Wir verzichten auf Pointer-Arithmetik :-)

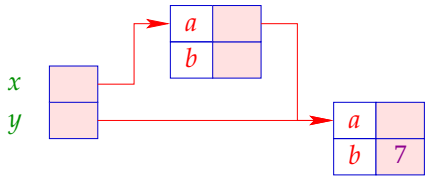
Bemerkung : new hat kein Argument, da in unseren Beispielen immer eine Struktur mit zwei Argumenten allokiert wird.
Die durch new hergestellten Komponenten werden immer im heap erzeugt.
R enthält die Adresse , a die Komponente. Andere Darstellung wäre R.a

Einfaches Beispiel:

$$
\begin{aligned}
&x = \mathsf{new}();\\
&y = \mathsf{new}();\\
&x \to a = y;\\
&y \to b = 7;
\end{aligned}
$$

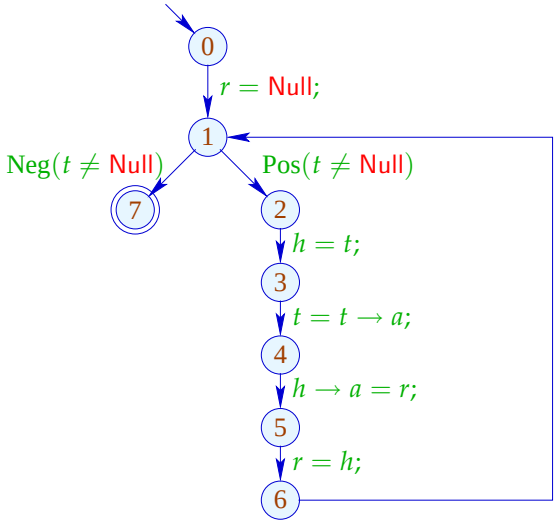


Die Semantik:



Schwierigeres Beispiel: *list reverse*

```
r = Null;
while (t ≠ Null) {
        h = t;
        t = t → a;
        h → a = r;
        r = h;
}
```



Konkrete Semantik:

Ein Speicher ist jetzt eine endliche Ansammlung von Paaren.
Nach h new-Operationen haben wir:

In einer Reihe von Programmiersprachen werden Adressen durch sog. tags gekennzeichnet und damit von integern unterschieden. (Hier Kennzeichnung durch ref)

$$
\begin{array}{lcll}
Addr_h & = & \{\mathsf{ref}\ a \mid 0 \leq a < h\} & // \quad \text{Adressen} \\
Val_h & = & Addr_h \cup \mathbb{Z} & // \quad \text{Werte} \\
Store_h & = & (Addr_h \times \{a, b\}) \rightharpoonup Val_h & // \quad \text{Speicher} \\
State_h & = & (Vars \rightharpoonup Val_h) \times Store_h & // \quad \text{Zustände}
\end{array}
$$

Der Einfachheit setzen wir: $0 = \mathsf{Null}$

Sei $(\rho, \mu) \in State_h$. Dann erhalten wir für die neuen Kanten:

$$
\begin{array}{lcl}
[\![x = \mathsf{new}();]\!]\ (\rho, \mu) & = & (\rho \oplus \{x \mapsto \mathsf{ref}\ h\}, \\
 & & \quad \mu \oplus \{(\mathsf{ref}\ h).a \mapsto 0, (\mathsf{ref}\ h).b \mapsto 0\}) \\
[\![x = R \rightarrow a;]\!]\ (\rho, \mu) & = & (\rho \oplus \{x \mapsto \mu\,((\rho\, R).a)\}, \mu) \\
[\![R \rightarrow a = x;]\!]\ (\rho, \mu) & = & (\rho, \mu \oplus \{(\rho\, R).a \mapsto \rho\, x\})
\end{array}
$$

Ein neues Objekt wird angelegt, indem man den Speicher mit (ref h).a und mit (ref h).b erweitert und mit 0 initialisiert.
Andernfalls würde hier ein undefinierter Wert stehen und die Semantik wäre nichtdeterministisch.

Achtung:

Diese Semantik ist zu detailliert, weil sie mit absoluten Adressen rechnet. Die beiden Programme:

$$
\begin{array}{ll}
x = \mathsf{new}(); & y = \mathsf{new}(); \\
y = \mathsf{new}(); & x = \mathsf{new}();
\end{array}
$$

werden nicht als äquivalent betrachtet !!?

Ausweg:

Definiere Äquivalenz bis auf Permutation von Adressen :-)

Anstatt der konkreten Ausführung des Programms, eine abstrakte Ausführung des Programms, dementsprechend:

Abstrakter Speicher anstatt konkretem Speicher, dieser besteht aus endlich vielen Elementen, somit hat man endlich viele abstrakte Adressen.
Man benutzt nun Kanten an denen ein new steht als abstrakte Adressen. Die Werte sind Mengen von Kanten, also Mengen von Adressen.
Also abstrakte Semantik anstatt konkreter Semantik.

Alias-Analyse 1. Idee:

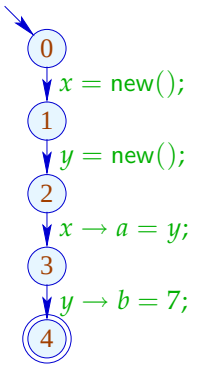
- Unterscheide endlich viele verschiedene Objekte im Speicher.
- Benutze Mengen von Adressen als abstrakte Werte!

  $\Longrightarrow$ Points-to-Analyse

$$
\begin{array}{lcll}
Addr^\sharp & = & Edges & // \text{ Erzeugungs-Kanten} \\
Val^\sharp & = & 2^{Addr^\sharp} & // \text{ Abstrakte Werte} \\
Store^\sharp & = & (Addr^\sharp \times \{a,b\}) \to Val^\sharp & // \text{ abstrakter Speicher} \\
State^\sharp & = & (Vars \to Val^\sharp) \times Store^\sharp & // \text{ Zustände}
\end{array}
$$

Die Menge aller Zustände bilden einen vollständigen Verband.

... im einfachen Beispiel:



	x	y	$(0,1).a$
0	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{(0,1)\}$	$\emptyset$	$\emptyset$
2	$\{(0,1)\}$	$\{(1,2)\}$	$\emptyset$
3	$\{(0,1)\}$	$\{(1,2)\}$	$\{(1,2)\}$
4	$\{(0,1)\}$	$\{(1,2)\}$	$\{(1,2)\}$

Die Kanten-Effekte:

$$
\begin{array}{lcl}
[\![(\_, ;, \_)]\!]^\sharp\,(D, M) & = & (D, M) \\
[\![(\_, \mathrm{Pos}(e), \_)]\!]^\sharp\,(D, M) & = & (D, M) \\
[\![(\_, x = y;, \_)]\!]^\sharp\,(D, M) & = & (D \oplus \{x \mapsto D\,y\}, M) \\
[\![(\_, x = e;, \_)]\!]^\sharp\,(D, M) & = & (D \oplus \{x \mapsto \emptyset\}, M) \qquad , \qquad e \notin \mathit{Vars}
\end{array}
$$

$$
\begin{array}{lcl}
[\![(u, x = \mathsf{new}();, v)]\!]^\sharp\,(D, M) & = & (D \oplus \{x \mapsto \{(u, v)\}\}, M) \\
[\![(\_, x = R \to a;, \_)]\!]^\sharp\,(D, M) & = & (D \oplus \{x \mapsto \bigcup\{M(f.a) \mid f \in D\,R\}\}, M) \\
[\![(\_, R \to a = x;, \_)]\!]^\sharp\,(D, M) & = & (D, M \oplus \{f \mapsto (M\,f \cup D\,x) \mid f \in D\,R\})
\end{array}
$$

Alias-Analyse 1. Idee:

- Unterscheide endlich viele verschiedene Klassen von Objekten im Speicher.
- Benutze Mengen von Adressen als abstrakte Werte!

  $\Longrightarrow$ Points-to-Analyse

Alle Objekte im Speicher, die an derselben Kante erzeugt werden, werden dementsprechend identifiziert.
Die Werte sind Teilmengen von Adressen.
Diese Analyse kann für keinen Programmpunkt feststellen, dass er nicht erreicht wird, deshalb :
Kleinstes Element : $\bot \quad = \quad (\lambda x.\emptyset\ ,\ \lambda(k,a].\emptyset)$

$$
\begin{array}{lcll}
Addr^\sharp & = & Edges & // \text{ Erzeugungs-Kanten} \\
Val^\sharp & = & 2^{Addr^\sharp} & // \text{ Abstrakte Werte} \\
Store^\sharp & = & (Addr^\sharp \times \{a,b\}) \to Val^\sharp & // \text{ abstrakter Speicher} \\
State^\sharp & = & (Vars \to Val^\sharp) \times Store^\sharp & // \text{ Zustände}
\end{array}
$$

// vollständiger Verband !!!

Die Kanten-Effekte:

$$
\begin{array}{lcl}
[\![(\_, ;, \_)]\!]^\sharp\,(D,M) & = & (D,M) \\
[\![(\_, \mathrm{Pos}(e), \_)]\!]^\sharp\,(D,M) & = & (D,M) \\
[\![(\_, x = y;, \_)]\!]^\sharp\,(D,M) & = & (D \oplus \{x \mapsto D\,y\}, M) \\
[\![(\_, x = e;, \_)]\!]^\sharp\,(D,M) & = & (D \oplus \{x \mapsto \emptyset\}, M) \quad , \quad e \notin Vars
\end{array}
$$

$$
\begin{array}{lcl}
[\![(u, x = \mathsf{new}();, v)]\!]^\sharp\,(D,M) & = & (D \oplus \{x \mapsto \{(u,v)\}\}, M) \\
[\![(\_, x = R \to a;, \_)]\!]^\sharp\,(D,M) & = & (D \oplus \{x \mapsto \bigcup\{M(f.a) \mid f \in D\,R\}\}, M) \\
[\![(\_, R \to a = x;, \_)]\!]^\sharp\,(D,M) & = & (D, M \oplus \{f.a \mapsto (M\,(f.a) \cup D\,x) \mid f \in D\,R\})
\end{array}
$$

Es gibt kein „destructive update“ im Speicher, da die abstrakte Adresse f nicht nur für ein Speicherelement , sondern für mehrere steht.

Achtung:

- Den Wert Null haben wir nicht mit-modelliert. Dereferenzieren von Null kann darum nicht entdeckt werden :-(
- Destruktive Updates sind nur von Variablen möglich, nicht im Speicher!

  $\Longrightarrow$ keine Information, falls Speicher-Objekte nicht vorinitialisiert sind :-((

  Ohne Vorinitialisierung ergäbe sich $\top$ *Bei Hinzufügen von Information ändert sich nichts, es bleibt* $\top$.
- Die Kanten-Effekte hängen jetzt von der ganzen Kante ab.

  Die Analyse lässt sich so nicht gegenüber der Referenz-Semantik als korrekt erweisen :-(

  Zur Korrektheit muss die konkrete Semantik mit zusätzlicher Information instrumentiert werden, die vermerkt, an welchem Programmpunkt eine Adresse erzeugt wurde.

Der konkrete Speicher nach h Instruktionen :
$Store = [0, h-1] \times \{a, b\} \longrightarrow Val\ h$

- ...
- Wir berechnen mögliche Points-to-Information.
  Man berechnet also nicht, ob zwei Variable möglicherweise auf das Gleiche zeigen, sondern die Menge der Objekte auf die sie zeigen können.
- Daraus können wir May-Alias-Information gewinnen.

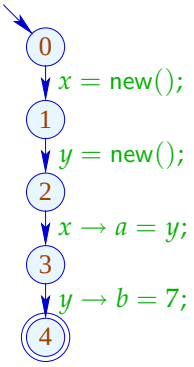
  Zwei Variablen haben einen May-Alias, wenn der Durchschnitt ihrer Werte nicht leer ist, dann zeigen sie möglicherweise auf das Gleiche.

- Die Analyse kann jedoch ziemlich aufwendig sein (ohne viel raus zu kriegen :-(
  Man muss an jedem Programmpunkt den abstrakten Speicher und auch für jede Speicherzelle im heap die Menge der Adressen verwalten.
- Separate Information für jeden Programmpunkt ist möglicherweise nicht nötig ??

Alias-Analyse 2. Idee:

Berechne für jede Variable und jede Adresse einen Wert, der die Werte an sämtlichen Programmpunkten sicher approximiert!

... im einfachen Beispiel:



x	$\{(0,1)\}$
y	$\{(1,2)\}$
$(0,1).a$	$\{(1,2)\}$
$(0,1).b$	$\emptyset$

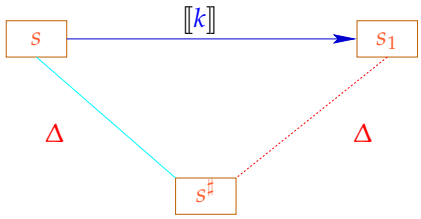
Jede Kante (u, lab, v) gibt Anlass zu Constraints:

lab	*Constraints*
$x = y;$	$\mathcal{P}[x] \supseteq \mathcal{P}[y]$
$x = \mathsf{new}();$	$\mathcal{P}[x] \supseteq \{(u,v)\}$
$x = R \to a;$	$\mathcal{P}[x] \supseteq \bigcup\{\mathcal{P}[f.a] \mid f \in \mathcal{P}[R]\}$
$R \to a = x;$	$\mathcal{P}[f.a] \supseteq (f \in \mathcal{P}[R])\,?\,\mathcal{P}[x] : \emptyset$ für alle $f \in Addr^\sharp$

Andere Kanten haben keinen Effekt :-)
Auch hier kein destruktives Update, die Informationen sind kumulativ.

Diskussion:

- Das resultierende Constraint-System ist $\mathcal{O}(k \cdot n)$ bei k abstrakten Adressen und n Kanten :-(
- Die Anzahl eventuell notwendiger Iterationen ist $\mathcal{O}(k)$ ...
  n ist die Grösse des Programms. Zur Laufzeit ergibt sich : $\mathcal{O}(k^2 \cdot n)$ *man kann aber auch* $\mathcal{O}(k \cdot n)$ *erreichen.*
- Die berechnete Information ist möglicherweise immer noch zu präzise !!?
- Zur Korrektheit einer Lösung $s^\sharp \in States^\sharp$ des Constraint-Systems zeigt man:

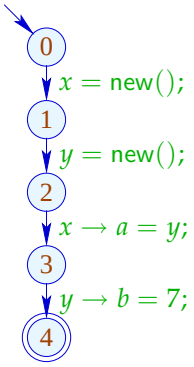


Alias-Analyse 3. Idee:

Wir berechnen die Points-to Information, also die Menge der möglichen Adressen für Variablen. Man ist nur an der Alias-Information interessiert, ob eben zwei Variablen möglicherweise auf das gleiche Objekt zeigen. Man berechnet eine Äquivalenzrelation auf Variable und Selektoren.
Es ergibt sich somit folgende Idee:

Berechne eine Äquivalenzrelation $\equiv$ auf Variablen x und Selektoren $y \to a$ mit $s_1 \equiv s_2$ falls an irgendeinem u s_1, s_2 die gleiche Adresse enthalten ...

... im einfachen Beispiel:



$$\equiv \;=\; \{\{x\},\ \{y, x \to a\},\ \{x \to b\},\ \{y \to b\}\}$$

Diskussion:

- $\to$ Wir berechnen eine Information für das ganze Programm.
- $\to$ Die Berechnung dieser Information verwaltet Partitionen $\pi = \{P_1, \ldots, P_m\}$:-)
  Das Konzept einer Datenstruktur (Union-find) zur Verwaltung von Partitionen ist ein bekanntes Problem der Informatik.
- $\to$ Einzelne Mengen P_i identifizieren wir durch einen Repräsentanten $p_i \in P_i$.
- $\to$ Die Operationen auf einer Partition π sind:

$$
\begin{array}{lcl}
\mathsf{find}\,(\pi, p) & = & p_i \qquad \text{falls } p \in P_i \\
 & // & \text{liefert den Repräsentanten} \\
\mathsf{union}\,(\pi, p_{i_1}, p_{i_2}) & = & \{P_{i_1} \cup P_{i_2}\} \cup \{P_j \mid i_1 \neq j \neq i_2\} \\
 & // & \text{vereinigt repräsentierte Klassen}
\end{array}
$$

→ Sind $x_1, x_2 \in \mathit{Vars}$ äquivalent,
müssen auch $x_i \to a$ und $x_i \to b$ äquivalent sein :-)

→ Ist $P_i \cap \mathit{Vars} \neq \emptyset$, soll auch $p_i \in \mathit{Vars}$ gelten.
Dann können wir union rekursiv anwenden :

$$
\begin{array}{lll}
\mathsf{union}^*\,(\pi, q_1, q_2) = \mathsf{let} & p_{i_1} = \mathsf{find}\,(\pi, q_1) & \\
 & p_{i_2} = \mathsf{find}\,(\pi, q_2) & \\
 & \mathsf{in}\ \ \mathsf{if}\ p_{i_1} == p_{i_2}\ \mathsf{then}\ \pi & \\
 & \quad \mathsf{else}\ \ \mathsf{let}\ \pi = \mathsf{union}\,(\pi, p_{i_1}, p_{i_2}) & \\
 & \qquad \mathsf{in}\ \ \mathsf{if}\ p_{i_1}, p_{i_2} \in \mathit{Vars}\ \mathsf{then} & \\
 & \qquad\quad \mathsf{let}\ \pi = \mathsf{union}^*\,(\pi, p_{i_1} \to a, p_{i_2} \to a) & \\
 & \qquad\quad \mathsf{in}\ \mathsf{union}^*\,(\pi, p_{i_1} \to b, p_{i_2} \to b) & \\
 & \qquad \mathsf{else}\ \pi &
\end{array}
$$

Die Analyse iteriert einmal über alle Kanten:

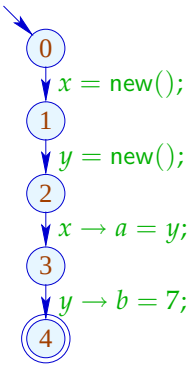
$$
\begin{array}{l}
\pi = \{\{x\}, \{x \to a\}, \{x \to b\} \mid x \in \mathit{Vars}\}; \\
\mathsf{forall}\, k = (\_, \mathit{lab}, \_)\ \mathsf{do} \quad \pi = [\![\mathit{lab}]\!]^\sharp\, \pi;
\end{array}
$$

Dabei ist:

$$
\begin{array}{lcl}
[\![x = y;]\!]^\sharp\, \pi & = & \mathsf{union}^*\,(\pi, x, y) \\
[\![x = R \to a;]\!]^\sharp\, \pi & = & \mathsf{union}^*\,(\pi, x, R \to a) \\
[\![R \to a = x;]\!]^\sharp\, \pi & = & \mathsf{union}^*\,(\pi, x, R \to a) \\
[\![\mathit{lab}]\!]^\sharp\, \pi & = & \pi \qquad \text{sonst}
\end{array}
$$

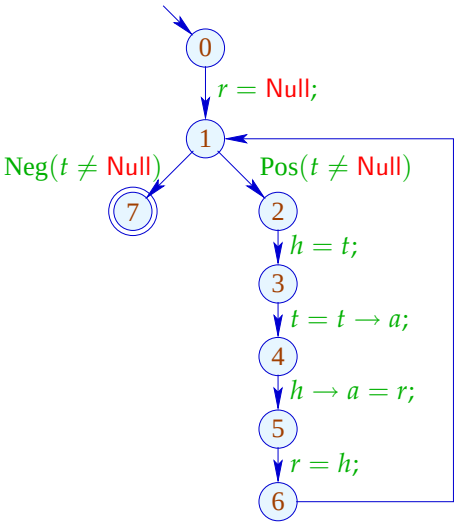
Anzahl der finds *ist Anzahl der Variablen* ($\#find = \mathcal{O}(\#Vars)$)
Anzahl der unions *ist die Grösse des Programms* ($\#union = \mathcal{O}(n)$)

... im einfachen Beispiel:



	$\{\{x\},\{y\},\{x \to a\},\{y \to a\},\ldots\}$
$(0,1)$	$\{\{x\},\{y\},\{x \to a\},\{y \to a\},\ldots\}$
$(1,2)$	$\{\{x\},\{y\},\{x \to a\},\{y \to a\},\ldots\}$
$(2,3)$	$\{\{x\},\boxed{\{y, x \to a\}},\{y \to a\},\ldots\}$
$(3,4)$	$\{\{x\},\{y, x \to a\},\{y \to a\},\ldots\}$

... im komplizierten Beispiel:



	$\{\{h\},\{r\},\{t\},\{h \to a\},\{t \to a\}\}$
$(2,3)$	$\{\boxed{\{h,t\}},\{r\},\boxed{\{h \to a, t \to a\}}\}$
$(3,4)$	$\{\boxed{\{h,t,h \to a, t \to a\}},\{r\}\}$
$(4,5)$	$\{\boxed{\{h,t,r,h \to a, t \to a\}}\}$
$(5,6)$	$\{\{h,t,r,h \to a, t \to a\}\}$

Ergebnis: Es werden keine interessanten Informationen gefunden.

Achtung:

Um überhaupt etwas heraus zu kriegen, müssen wir annehmen, dass alle Variablen anfangs auf verschiedene Adressen zeigen. *Oder auf Null zeigen*

Zur Komplexität:

Wir haben:

$\mathcal{O}(\#\,Kanten)$	Aufrufe von	union*
$\mathcal{O}(\#\,Kanten)$	Aufrufe von	find
$\mathcal{O}(\#\,Vars)$	Aufrufe von	union

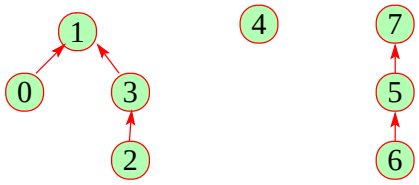
$\Longrightarrow$ Wir benötigen effiziente Union-Find-Datenstruktur :-)

Idee:

Repräsentiere Partition von U als gerichteten Wald:

- Zu $u \in U$ verwalten wir einen Vater-Verweis $F[u]$.
- Elemente u mit $F[u] = u$ sind Wurzeln. *(Vater-Verweis zeigt auf sich selbst.)*

Einzelne Bäume sind Äquivalenzklassen.
Ihre Wurzeln sind die Repräsentanten ...



0	1	2	3	4	5	6	7

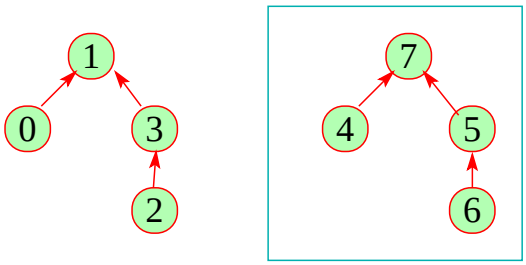
1	1	3	1	4	7	5	7

Das zweite arra repräsentiert die Väter.
Zur Implementierung benützt man am einfachsten Hash-tabellen

$\rightarrow$ find (π, u) folgt den Vater-Verweisen :-)

$\rightarrow$ union (π, u_1, u_2) hängt den Vater-Verweis eines u_i um ...

Nach union



0	1	2	3	4	5	6	7

1	1	3	1	7	7	5	7

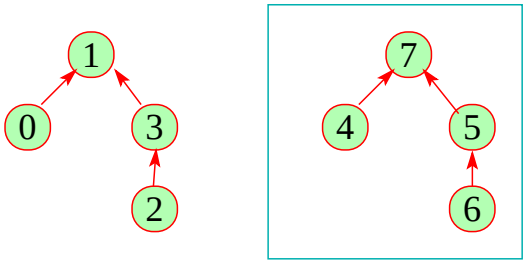
Die Kosten:

union	:	$\mathcal{O}(1)$	:-)
find	:	$\mathcal{O}(depth(\pi))$	:-(

Strategie zur Vermeidung tiefer Bäume:

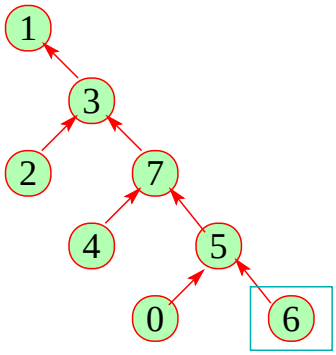
- Hänge den kleineren Baum unter den größeren ! *Bäume werden i.A. nicht tiefer. Aufwand: Feststellen des kleineren Baumes*
- Benutze find , um Pfade zu komprimieren ...

Strategie : Nach union



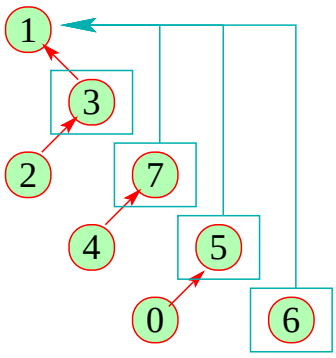
0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7

Strategie : find *Komprimierung der Pfade*

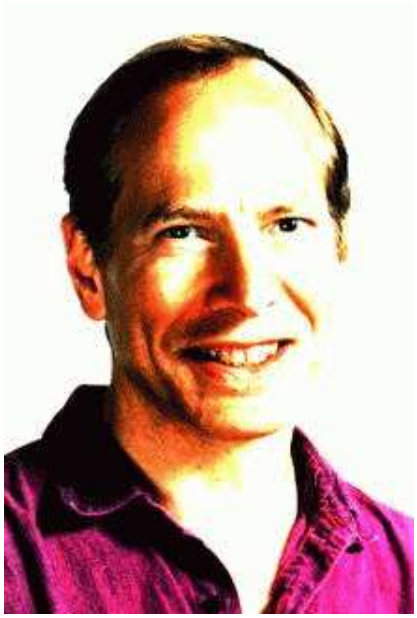


0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3

Ergebnis : find *Komprimierung der Pfade*



0	1	2	3	4	5	6	7
5	1	3	1	1	7	1	1



Robert Endre Tarjan, Princeton

Beachte:

- Mit dieser Datenstruktur dauern n union- und m find-Operationen $\mathcal{O}(n + m \cdot \alpha(n, n))$
  // α die inverse Ackermann-Funktion :-)
- Für unsere Anwendung müssen wir union nur so modifizieren, dass an den Wurzeln nach Möglichkeit Elemente aus *Vars* stehen.
- Diese Modifikation vergrößert die asymptotische Laufzeit nicht :-)

Fazit:

Die Analyse ist blitzschnell — findet aber nicht sehr viel heraus.

Exkurs 3: Fixpunkt-Algorithmen

Beobachtung:

RR-Iteration ist ineffizient:

→ Wir benötigen eine ganze Runde, um Terminierung festzustellen :-(

→ Ändert sich in einer Runde der Wert nur einer Variable, berechnen wir trotzdem alle neu :-(

→ Die praktische Laufzeit hängt von der Reihenfolge der Variablen ab :-(

Idee: Workset-Iteration

Auch Work-List-Algorithmus. Workset-Iteration ändert Variable dann, wenn man annehmen muss, dass sie ihre Werte unter falschen Annahmen bekommen haben.

Ändert eine Variable x_i ihren Wert, werten wir alle Variablen neu aus, die von x_i abhängen.

Technisch benötigen wir:

→ die Mengen $Dep\ f_i$ der Variablen, auf die die Auswertung von f_i zugreift. Daraus berechnen wir:

$$I[x_i] = \{x_j \mid x_i \in Dep\ f_j\}$$

d.h. die Menge der x_j, die von x_i abhängen.

→ die Werte $D[x_i]$ der x_i, wobei anfangs $D[x_i] = \bot$;

→ Eine Menge W der Variablen, deren Wert neu berechnet werden muss ...

Der Algorithmus:

$W = \{x_1, \ldots, x_n\}$;
while $(W \neq \emptyset)$ {
  x_i = extract W;
  t = f_i eval;
  if $(t \not\sqsubseteq D[x_i])$ {
    $D[x_i]$ = $D[x_i] \sqcup t$;
    W = $W \cup I[x_i]$;
  }
}

wobei :

$eval\ x_j = D[x_j]$

Beispiel:

$$
\begin{aligned}
x_1 &\supseteq \{a\} \cup x_3 \\
x_2 &\supseteq x_3 \cap \{a, b\} \\
x_3 &\supseteq x_1 \cup \{c\}
\end{aligned}
$$

	I
x_1	$\{x_3\}$
x_2	$\emptyset$
x_3	$\{x_1, x_2\}$

$D[x_1]$	$D[x_2]$	$D[x_3]$	W
$\emptyset$	$\emptyset$	$\emptyset$	$\boxed{x_1}, x_2, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_2}, x_3$
$\{a\}$	$\emptyset$	$\emptyset$	$\boxed{x_3}$
$\{a\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_1}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_3}, x_2$
$\{a, c\}$	$\emptyset$	$\{a, c\}$	$\boxed{x_2}$
$\{a, c\}$	$\{a\}$	$\{a, c\}$	$\emptyset$

In diesem Beispiel wurde gegenüber der RR-Iteration eine Iteration gespart.

Theorem

Sei $x_i \sqsupseteq f_i(x_1, \ldots, x_n)$, $i = 1, \ldots, n$
ein Constraint-System über dem vollständigen Verband $\mathbb{D}$ der Höhe $h > 0$.

(1) Der Algorithmus terminiert nach maximal $h \cdot N$ Auswertungen rechter Seiten
(RR-Iteration : $(n \times h) \times N$ wobei dann $N = n$, somit $n^2 \times h$)
, wobei

$$N \geq \sum_{i=1}^{n} (1 + \#(Dep\ f_i)) \qquad // \quad \text{Größe des Systems} \quad :\text{-)}$$

(2) Der Algorithmus liefert eine Lösung.
Sind alle f_i monoton, liefert er die kleinste.

Beweis:

Zu (1):

Jede Variable x_i kann nur h mal ihren Wert ändern :-)
Dann wird die Menge $I[x_i]$ zu W hinzu gefügt.
Damit ist die Anzahl an Auswertungen:

$$\begin{aligned}
&\leq n + \textstyle\sum_{i=1}^{n} (h \cdot \#(I[x_i])) \\
&= n + h \cdot \textstyle\sum_{i=1}^{n} \#(I[x_i]) \\
&= n + h \cdot \textstyle\sum_{i=1}^{n} \#(Dep\ f_i) \\
&\quad \textit{Durch eine Erweiterung kommt man zu :} \\
&\quad \leq h \cdot n + h \cdot \textstyle\sum_{i=1}^{n} (\cdots \\
&\quad \textit{und erhält somit die Abschätzung :} \\
&\leq h \cdot \textstyle\sum_{i=1}^{n} (1 + \#(Dep\ f_i)) \\
&= h \cdot N
\end{aligned}$$

Der Algorithmus terminiert also nur, wenn die Anzahl der aufsteigenden Ketten endlich ist.
Der Algorithmus terminiert nicht, wenn der vollständige Verband unendlich aufsteigende Ketten hat.

Zu (2):

wir betrachten nur die Aussage für monotone f_i.
Sei D_0 die kleinste Lösung. Man zeigt:

- $D_0[x_i] \sqsupseteq D[x_i]$ (zu jedem Zeitpunkt)
- $D[x_i] \not\sqsupseteq f_i\,\mathsf{eval} \Longrightarrow x_i \in W$ (am Ende des Rumpfs)
- Bei Terminierung liefert der Algo eine Lösung :-))

Diskussion:

- Im Beispiel werden tatsächlich weniger Auswertungen rechter Seiten benötigt als bei RR-Iteration :-)
- Der Algo funktioniert auch für nicht-monotone f_i :-)
- Für monotone f_i kann man den Algo vereinfachen:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \Longrightarrow \boxed{D[x_i] = t;}$$

- Für Widening ersetzt man:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \Longrightarrow \boxed{D[x_i] = D[x_i] \sqcup t;}$$

- Für Narrowing ersetzt man:

$$\boxed{D[x_i] = D[x_i] \sqcup t;} \Longrightarrow \boxed{D[x_i] = D[x_i] \sqcap t;}$$

Man sieht, dass hier, nur der jeweilige Operator durch den Widening/Narrowing-Operator ersetzt werden muss. Bei Narrowing muss ausserdem mit dem richtigen Startwert begonnen werden.

Achtung:

- Der Algorithmus benötigt die Variablen-Abhängigkeiten $Dep\ f_i$.
  In unseren bisherigen Anwendungen waren diese Abhängigkeiten offensichtlich.
  Das muss nicht immer so sein :-(
- Wir benötigen eine Strategie für extract ,
  die festlegt, welche Variable als nächstes auszuwerten ist.
  W ist eine Menge; Die Implementierung des Algorithmus erfolgt mit einer Liste, wobei z.B. durch ein Bit (is-in-worklist) eine Mehrfachbearbeitung in der Worklist vermieden wird. Die Strategien zur Effizienz der Bearbeitung: stack- oder queue-Strategie zur Auswertung der Variablen in der Liste. Die Effizienz der Strategien ist in etwa gleich. Der Algorithmus kann aber durch eine verbesserte Strategie bei der Auswertung effizienter gemacht werden. (Rekursive Auswertung!)
- Am besten wäre es, wenn wir erst auswerten,
  dann auf das Ergebnis zugreifen ... :-)

  $\Longrightarrow$ rekursive Auswertung ...

Idee:

→ Greifen wir in f_i auf ein x_j zu, werten wir erst rekursiv aus. Dann fügen wir x_i zu $I[x_j]$ hinzu :-)

$$\begin{aligned}\mathsf{eval}\ x_i\ x_j\ =\ & \mathsf{solve}\ x_j;\\ & I[x_j] = I[x_j] \cup \{x_i\};\\ & D[x_j];\end{aligned}$$

→ Damit die Rekursion nicht unendlich absteigt,
verwalten wir die Menge *Stable* von Variablen, für die solve den Wert nachschlägt :-)
z.B. $x_1 \sqsupseteq x_1 \cup \{a\}$ *hier Rekursion!*

Anfangs ist $Stable = \emptyset$ ...

Die Funktion solve :

$$\begin{aligned}\mathsf{solve}\ x_i\ =\ & \mathsf{if}\ (x_i \notin Stable)\ \{\\ & \quad Stable = Stable \cup \{x_i\};\\ & \quad t = f_i\ (\mathsf{eval}\ x_i);\\ & \quad \mathsf{if}\ (t \not\sqsubseteq D[x_i])\ \{\\ & \quad\quad W = I[x_i];\quad I[x_i] = \emptyset;\\ & \quad\quad D[x_i] = D[x_i] \sqcup t;\\ & \quad\quad Stable = Stable \backslash W;\\ & \quad\quad \mathsf{app}\ \mathsf{solve}\ W;\\ & \quad \}\\ & \}\end{aligned}$$

Der Algorithmus ist tailrekursiv.
Definition von app:
$app[\,] = ()$
$app\ f(x, x_s) = fx\ ;\ app\ f x_s$

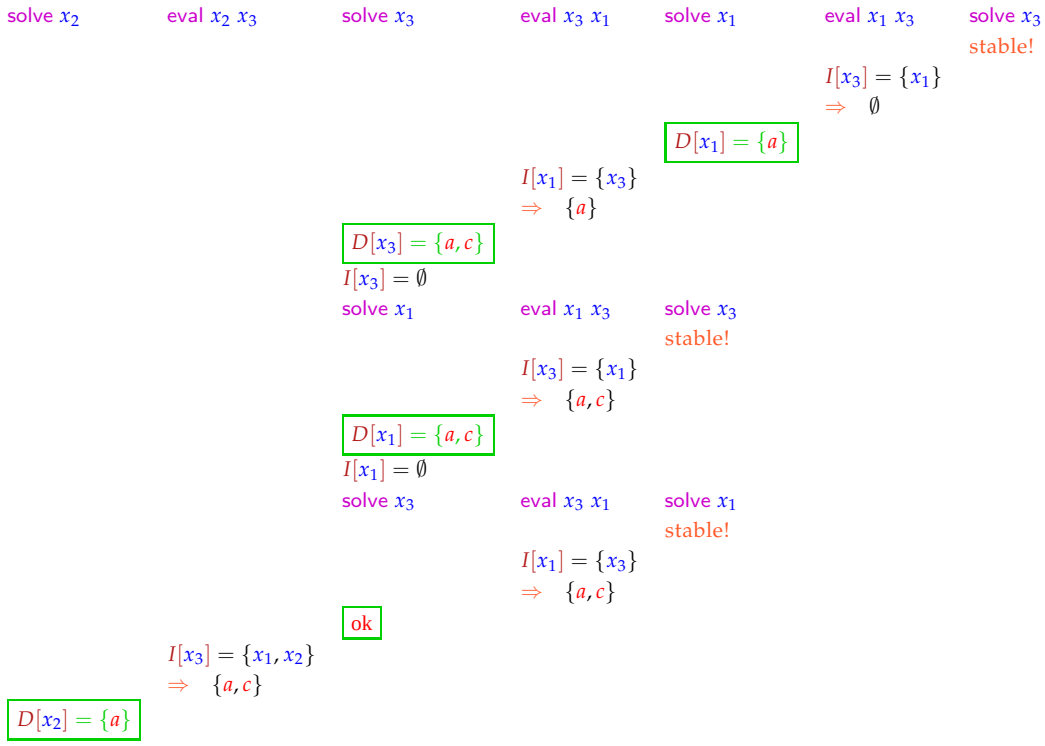
Helmut Seidl, TU München ;-)

Beispiel:

Betrachte unser Standard-Beispiel:

$$\begin{array}{lcl} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Dann sieht ein Trace des Fixpunkt-Algorithmus etwa so aus:
Rekursionsstack $\longrightarrow$



Der Algorithmus lässt sich elegant mit einer Programmiersprache die partielle Anwendungen unterstützt, implementieren. z.B: ML

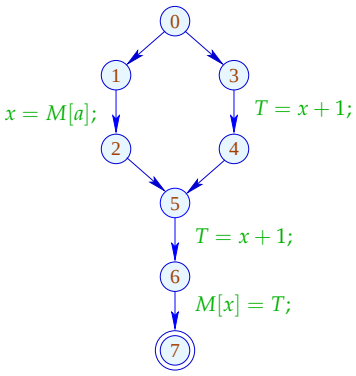
- $\rightarrow$ Die Auswertung startet mit einer interessierenden Variable x_i (z.B. dem Wert für *stop*) *Endpunkt des Programms.*
- $\rightarrow$ Es werden automatisch alle Variablen ausgewertet, die x_i beeinflussen :-)
- $\rightarrow$ Die Anzahl der Auswertungen ist i.a. kleiner als die bei normaler Iteration ;-)
- $\rightarrow$ Der Algorithmus ist komplizierter, benötigt aber keine Vorberechnung der Variablen-Abhängigkeiten :-))
- $\rightarrow$ Er funktioniert auch, wenn die Variablen-Abhängigkeiten sich während der Iteration ändern !!!

$\Longrightarrow$ interprozedurale Analyse

Es werden also NUR die Variablen betrachtet die x_i beeinflussen.

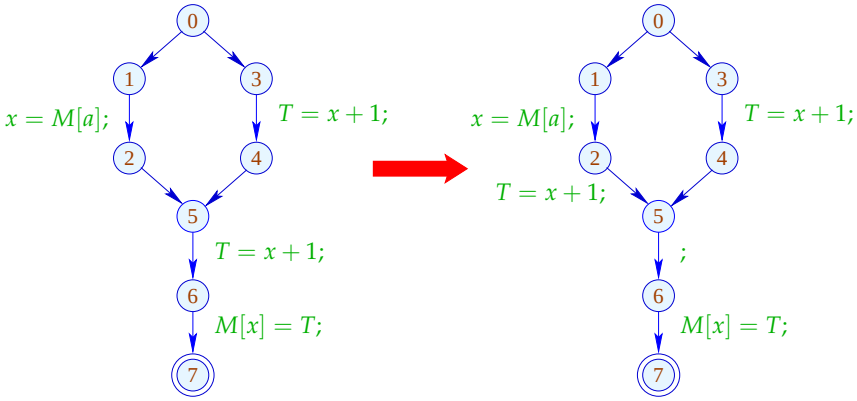
1.7 Beseitigung partieller Redundanzen

Beispiel:



// $e = x + 1$ wird auf jedem Pfad ausgewertet ...
// leider auf einem Pfad sogar zweimal :-(

Ziel:



code-motion

Idee:

(1) Wende T1 an, d.h. ersetze jede interessierende Zuweisung x=e;
durch: $T_e = e; x = T_e;$
Für Berechnungen werden Register gemäss Transformation T_1 eingeführt.

(2) Finde alle Stellen, an denen e sicher berechnet werden kann,
ohne die Semantik zu zerstören.

(3) Platziere (konzeptuell) $T_e = e;$ an allen diesen Plätzen.

Beseitige die redundanten Zuweisungen mittels T2.
Die entstandenen Mehrfachberechnungen werden mit der Transformation T_2 entfernt.

$\Longrightarrow$ wir benötigen eine neue Analyse :-))

Ein Ausdruck e heißt aktiv (busy) entlang eines Pfads π,
falls der Wert von e berechnet wird,
bevor eine der Variablen $x \in Vars(e)$ überschrieben wird.

// Rückwärtsanalyse!
Es geht um zukünftige Berechnungen!!

e heißt sehr aktiv (very busy) an u, falls e aktiv ist entlang jedes Pfads
$\pi : u \to^* stop$.

Entsprechend benötigen wir:

$$\mathcal{B}[u] \;=\; \bigcap \{ [\![\pi]\!]^\sharp \; \emptyset \mid \pi : u \to^* stop \}$$

wobei für $\pi = k_1 \ldots k_m$:

$$[\![\pi]\!]^\sharp \;=\; [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_m]\!]^\sharp$$

Unser vollständiger Verband ist:

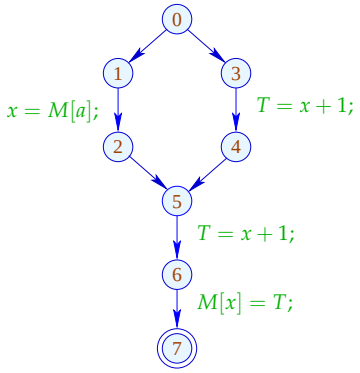
$$\mathbb{B} = 2^{Expr \setminus Vars} \qquad \text{mit} \quad \sqsubseteq \;=\; \supseteq$$

Der Effekt $[\![k]\!]^\sharp$ einer Kante $k = (u, lab, v)$ hängt nur von lab ab,
d.h. $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ wobei:

$$
\begin{array}{lcl}
[\![;]\!]^\sharp B & = & B \\
[\![Pos(e)]\!]^\sharp B & = & [\![Neg(e)]\!]^\sharp B \quad = \quad B \cup \{e\} \\
[\![T_e = e;]\!]^\sharp B & = & (B \backslash Expr_{T_e}) \cup \{e\} \\
[\![x = T;]\!]^\sharp B & = & B \backslash Expr_x \\
[\![x = M[R];]\!]^\sharp B & = & B \backslash Expr_x \\
[\![M[R] = x;]\!]^\sharp B & = & B
\end{array}
$$

Die Kanten-Effekte sind sämtlich distributiv.
Deshalb liefert die kleinste Lösung des Constraint-Systems exakt den MOP :-)

Beispiel: *Mit RR-Iteration*



7	$\emptyset$
6	$\emptyset$
5	$\{x+1\}$
4	$\{x+1\}$
3	$\{x+1\}$
2	$\{x+1\}$
1	$\emptyset$
0	$\emptyset$

Beachte:

- Im Beispiel enthalten die $\mathcal{B}[u]$ maximal ein Element.
- Enthält $\mathcal{B}[u]$ mehrere Ausdrücke $e_1 \neq e_2$,
  sind diese unabhängig, d.h. $T_{e_1} \notin Vars(e_2)$:-)
- Unabhängige Ausdrücke können in beliebiger Reihenfolge berechnet werden :-))

Beweis der Anordbarkeit:

→ Wir zeigen Beh. für $[\![\pi]\!]^\sharp \emptyset$. Die Beh. für u folgt,
da die Eigenschaft unter $\cap$ abgeschlossen ist.

→ Induktion über die Länge von π.

$\boxed{\pi = \epsilon}$ $[\![\pi]\!]^\sharp \emptyset = [\![\epsilon]\!]^\sharp \emptyset = \emptyset$:-)

$\boxed{\pi = k\ \pi'}$ Kanten-Effekte erhalten die Anordbarkeit :-))

Ein u heißt sicher für e, sofern $e \in \mathcal{A}[u] \cup \mathcal{B}[u]$;
d.h. e ist entweder verfügbar oder sehr aktiv.

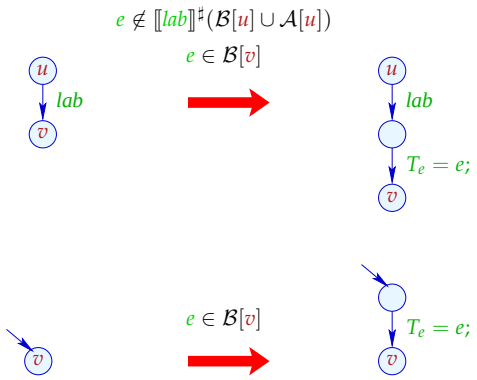
Ist u sicher, können wir dort e gefahrlos berechnen :-)

Idee:

- Wir berechnen e zum frühest möglichen Zeitpunkt :-)
- Wir platzieren die Berechnung von e am Ende von $k = (u, lab, v)$ falls:
  - $\rightarrow$ $e \in \mathcal{B}[v]$ sowie
  - $\rightarrow$ $e \notin [\![lab]\!]^\sharp(\mathcal{A}[u])$ (nicht verfügbar entlang k) und
  - $\rightarrow$ $e \notin [\![lab]\!]^\sharp(\mathcal{B}[u])$ (auch nicht nach Transformation)
- Weil alle $e \in \mathcal{B}[v]$ anordbar sind,
  betrachten wir die Transformation für jedes e gesondert:

Transformation 6.1: *Löschen der Berechnungen im Programm*

Transformation 6.2: *Einfügen der Berechnungen an bestimmten Stellen aufgrund folgender Voraussetzungen:*

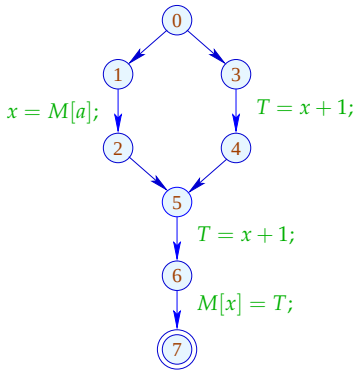


Bernhard Steffen, Dortmund

Jens Knoop, Wien

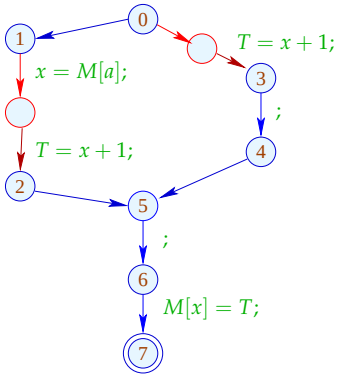
Im Beispiel:

Mit den beiden Analysen: A (Availability-Analyse, vorwärts) und B (Very-busy-Analyse, rückwärts)



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

Aufspaltung der Kanten durch Einführung von Zwischenknoten (Dummy-Knoten) und NOP-Kanten. Diese werden durch eine weitere Transformation wieder entfernt.



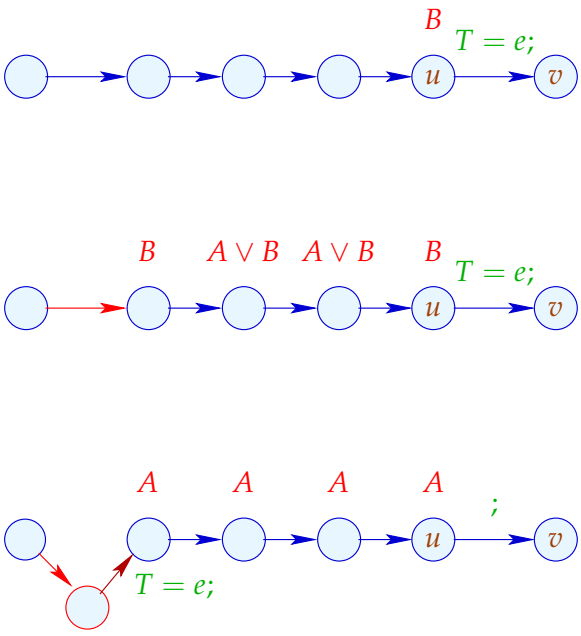
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\emptyset$
7	$\{x+1\}$	$\emptyset$

Zur Korrektheit:

Sei $\pi = \pi' k$ ein Pfad mit $k = (u, T = e;, v)$.

Dann gilt: $e \in \mathcal{B}[u]$ da u nur eine ausgehende Kante hat :-)

Wir haben:



Wir schließen:

- Überall, wo wir $T = e;$ gestrichen haben, ist e verfügbar :-)
  $\Longrightarrow$ Korrektheit der Transformation

- Jedem $T = e;$, das wir in einen Pfad einfügen,
  entspricht ein $T = e;$, das wir gestrichen haben :-))
  $\Longrightarrow$ Nicht-Verschlechterung der Transformation

1.8 Anwendung: Schleifen-invarianter Code

Beispiel:

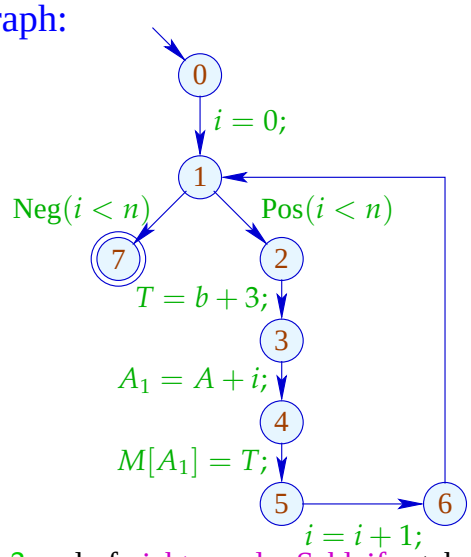
$$\text{for } (i = 0; i < n; i{+}{+})$$
$$a[i] = b + 3;$$

Oder auch z.B. : Ein Index wird durch konstante Teilausdrücke gebildet und jedesmal in der Schleife berechnet, oder er hängt von Variablen ab, die innerhalb der Schleife nicht modifiziert werden.

// Der Ausdruck $b + 3$ wird in jeder Iteration berechnet :-(

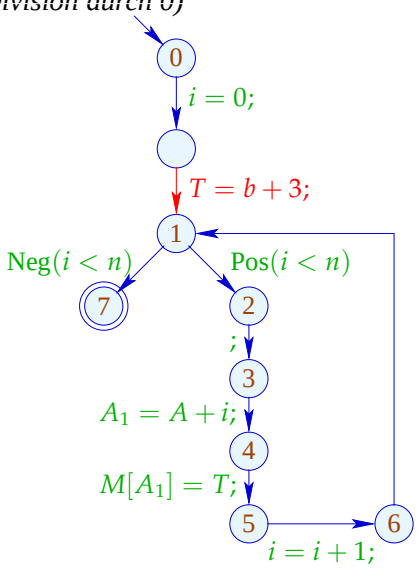
// Das wollen wir vermeiden :-)

Der Kontrollfluss-Graph:



Achtung: $T = b + 3;$ darf nicht vor der Schleife stehen :

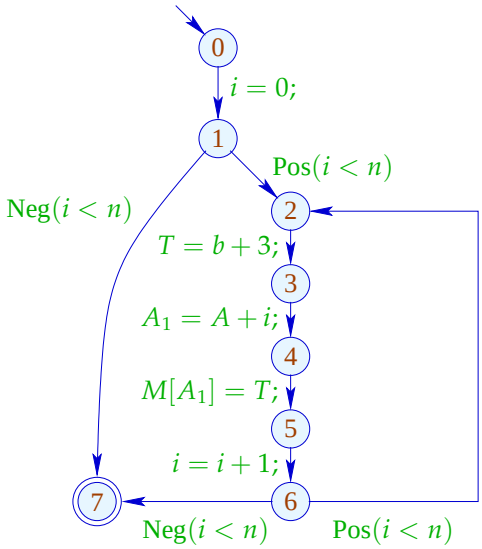
Der Ausdruck wird auch dann ausgewertet, wenn die Schleife NIE! durchlaufen wird;
(Oder auch : overflow oder Division durch 0)



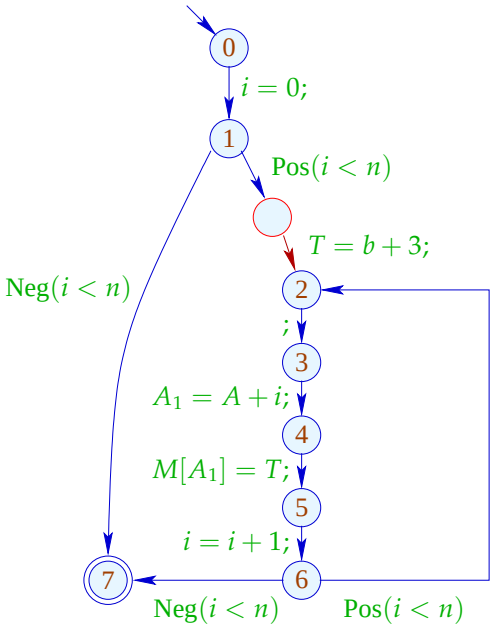
$\Longrightarrow$ Es gibt keinen guten Platz für $T = b + 3;$:-(

Idee: Transformiere in eine do-while-Schleife ...

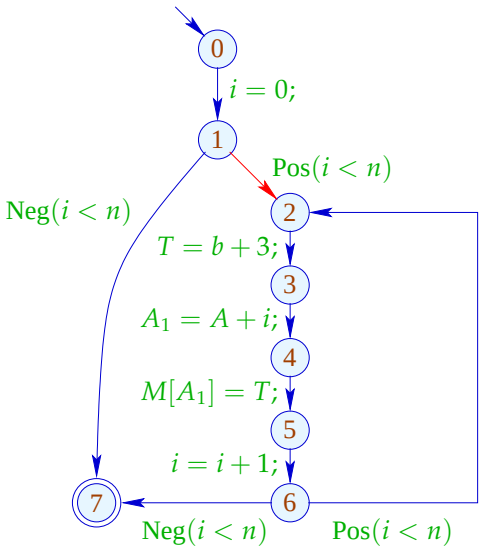
Aufteilung der Schleifenbedingung in 2 Teile (if (Vor erster Ausführung der Schleife) und do-while. (Die Abfrage ob der Rumpf weiter durchlaufen wird, steht nun am Ende.))



... jetzt gibt es eine Stelle für $T = e;$:-)



Anwendung von T6 (PRE) :



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{b+3\}$
3	$\{b+3\}$	$\emptyset$
4	$\{b+3\}$	$\emptyset$
5	$\{b+3\}$	$\emptyset$
6	$\{b+3\}$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$

Fazit:

- Beseitigung partieller Redundanzen kann loop-invarianten Code aus Schleifen heraus schieben :-))
- Das funktioniert nur für do-while-Scheifen :-(
- Um andere Scheifen zu optimieren, wandeln wir sie in do-while-Scheifen um:

$$\texttt{while}\ (b)\ stmt \quad \Longrightarrow \quad \texttt{if}\ (b)\ \texttt{do}\ stmt\ \texttt{while}\ (b);$$

$\Longrightarrow$ Schleifen-Rotation

Problem:

Haben wir das Quell-Programm nicht (mehr) zur Verfügung, müssen wir nachträglich die Schleifen (-köpfe) identifizieren ;-)

$\Longrightarrow$ Prädominatoren

u prädominiert v , falls jeder Pfad $\pi : start \rightarrow^* v$ Knoten u enthält.

Wir schreiben: $u \Rightarrow v$.

"$\Rightarrow$" ist reflexiv, transitiv und anti-symmetrisch :-)

Berechnung:

Wir sammeln die Knoten entlang Pfaden auf mithilfe der Analyse:

$$\mathbb{P} = 2^{Nodes} \quad , \qquad \sqsubseteq \; = \; \supseteq$$

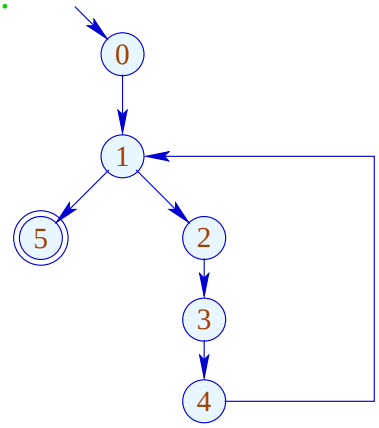
$$[\![(\_, \_, v)]\!]^\sharp \, P \;\; = \;\; P \cup \{v\}$$

Dann ist die Menge $\mathcal{P}[v]$ der Prädominatoren:

$$\mathcal{P}[v] = \bigcap \{ [\![\pi]\!]^\sharp \, \{start\} \mid \pi : start \rightarrow^* v \}$$

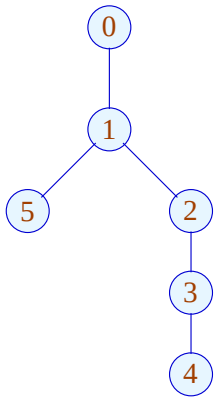
Da die $[\![k]\!]^\sharp$ distributiv sind, können wir die $\mathcal{P}[v]$ mithilfe von Fixpunkt-Iteration berechnen :-)

Beispiel:



	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Die partielle Ordnung "$\Rightarrow$" im Beispiel:



	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Offenbar ist das Ergebnis ein Baum :-)
Tatsächlich gilt:

Satz:

Jeder Punkt v hat maximal einen unmittelbaren Prädominator.
d.h. Es ist immer ein Baum!

Beweis:

Annahme:

Es gäbe $u_1 \neq u_2$, die v unmittelbar prädominieren.

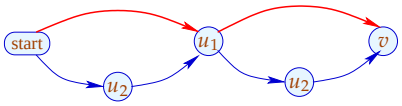
Gälte $u_1 \Rightarrow u_2$, wäre u_1 nicht unmittelbar.

Folglich müssen u_1, u_2 unvergleichbar sein :-)

Nun gilt für jedes $\pi : start \rightarrow^* v$:

$$\pi = \pi_1\ \pi_2 \qquad \text{mit} \qquad \pi_1 : start \rightarrow^* u_1 \qquad \pi_2 : u_1 \rightarrow^* v$$

Sind u_1, u_2 aber unvergleichbar, gibt es einen Pfad: $start \rightarrow^* v$ ohne u_2 :



Beobachtung:

Der Schleifenkopf einer while-Schleife dominiert jeden Knoten des Rumpfs.

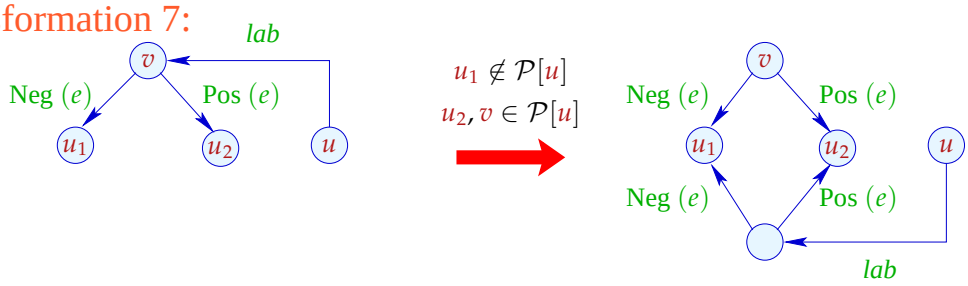
Einen Rücksprung vom Ende u zum Schleifenkopf v erkennt man daran, dass

$$v \in \mathcal{P}[u]$$

:-)

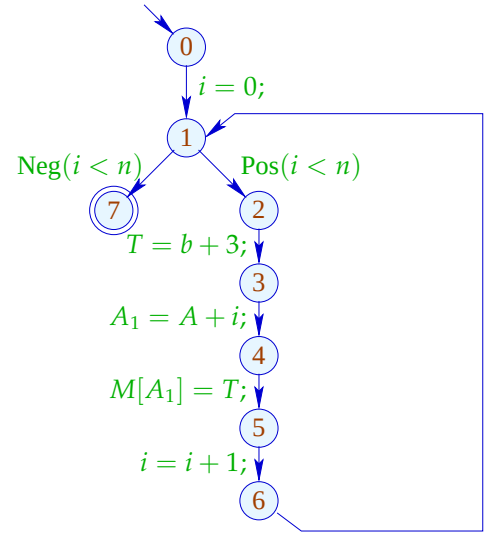
Damit definieren wir:

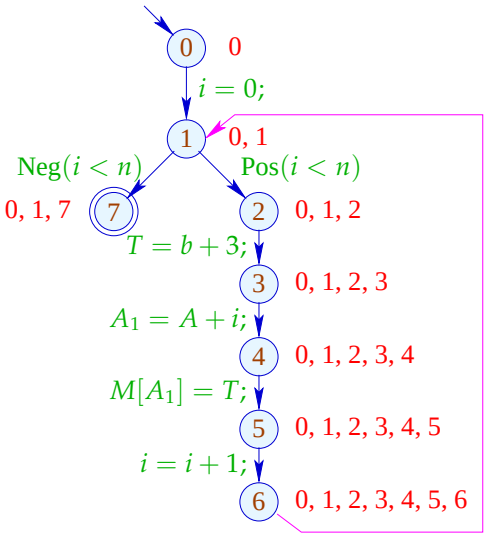
Transformation 7:

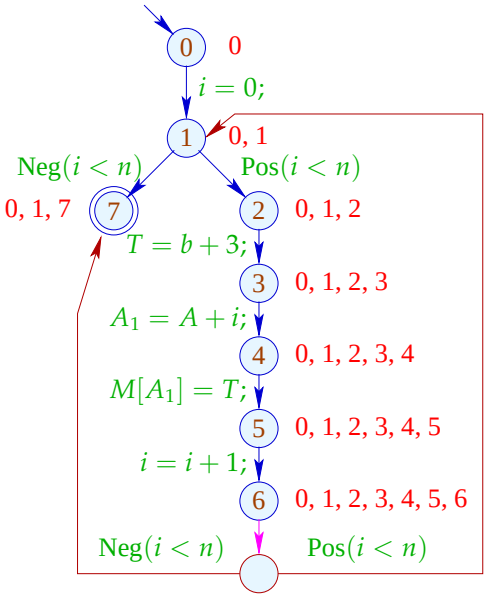


Wir duplizieren den Eintritts-Test an alle Rücksprung-Stellen :-)

... im Beispiel:

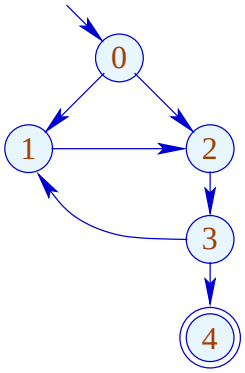




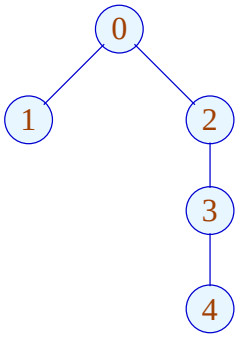


Achtung:

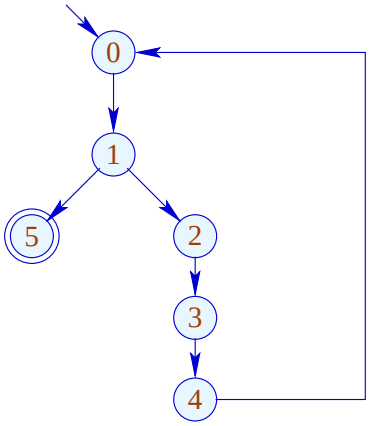
Es gibt ungewöhnliche Schleifen, die so nicht rotiert werden:

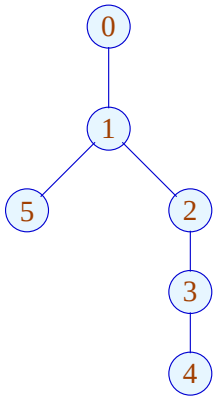


Prädominatoren:

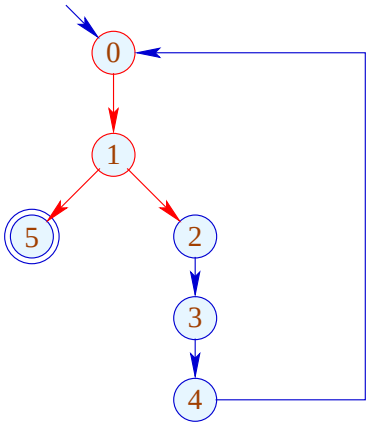


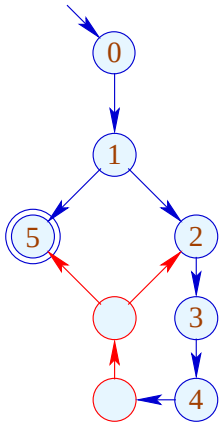
... leider aber auch gewöhnliche, die nicht rotiert werden:





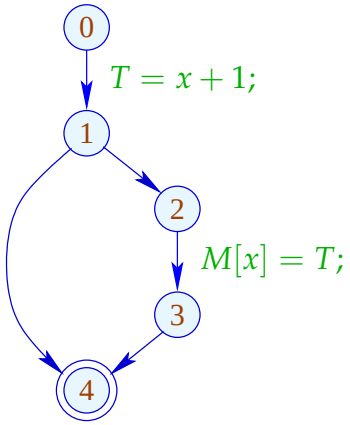
Hier müsste man den ganzen Pfad zwischen Rücksprung und Bedingung duplizieren :-(





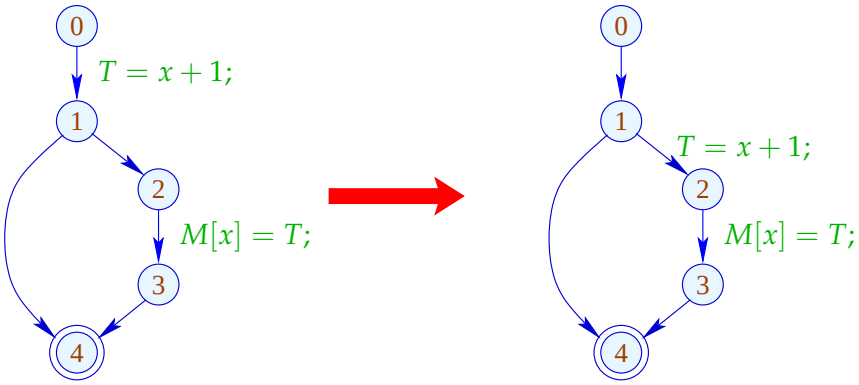
1.9 Beseitigung partiell toten Codes

Beispiel:



$x + 1$ muss nur auf einem der Pfade berechnet werden ;-(

Idee:



Problem:

- Die Definition $T_e = e;$ darf nur dorthin geschoben werden, wo sie eh verfügbar ist ;-)
- Die Definition muss weiterhin für Benutzungen zur Verfügung stehen ;-)

$\Longrightarrow$

Wir schieben sie an den Anfang einer Kante (u, lab, v) mit:

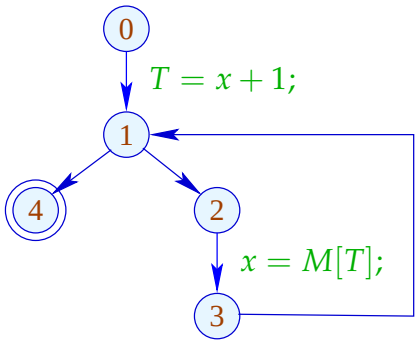
1. $e \in \mathcal{A}[u]$; und
2. $e \notin \mathcal{A}[v] \quad \vee \quad T_e \in Use(lab)$

Dabei ist: *Benutzung (Use) einer Variablen.*

lab	Use
$;$	$\emptyset$
$\mathrm{Pos}(e)$	$Vars(e)$
$\mathrm{Neg}(e)$	$Vars(e)$
$T_e = e;$	$Vars(e)$
$x = R;$	$\{R\}$
$x = M[R];$	$\{R\}$
$M[R] = x;$	$\{x, R\}$

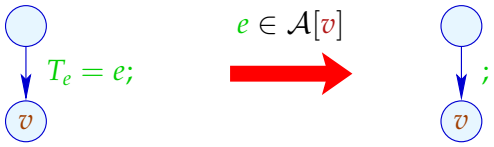
Achtung:

Wir können $T_e = e;$ nur verschieben, falls e am Ende der Kante verfügbar ist:

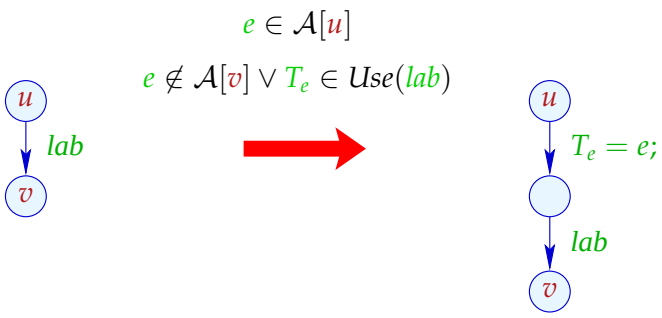


Offenbar ist $x + 1$ nicht an 1 verfügbar, darf also nicht verschoben werden !!!

Transformation 8.1:

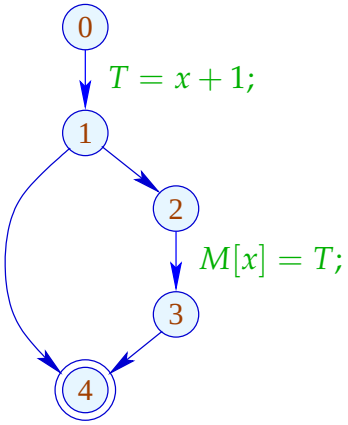


Transformation 8.2:

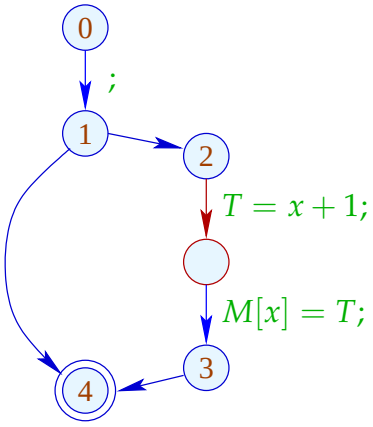


Beachte:

Die Transformation T8 ist nur sinnvoll, nachdem T1 gemacht wurde.
Im Beispiel beseitigt sie den partiell toten Code:



	$\mathcal{A}$
0	$\emptyset$
1	$\{x+1\}$
2	$\{x+1\}$
3	$\{x+1\}$
4	$\{x+1\}$



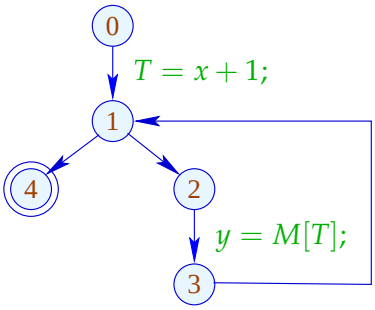
	$\mathcal{A}$
0	$\emptyset$
1	$\{x+1\}$
2	$\{x+1\}$
3	$\{x+1\}$
4	$\{x+1\}$

Fragen:

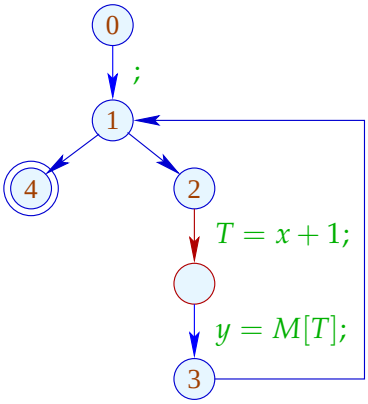
1. Ist die Transformation korrekt ???
2. Ist sie beweisbar nicht-verschlechternd ???

Beispiel:

Problem : Es wird schleifen-invarianter Code erzeugt!
Diese Verschlechterung könnte aber eventuell wieder durch Beseitigung partieller Redundanz behoben werden.



	$\mathcal{A}$
0	$\emptyset$
1	$\{x+1\}$
2	$\{x+1\}$
3	$\{x+1\}$
4	$\{x+1\}$



	$\mathcal{A}$
0	$\emptyset$
1	$\{x+1\}$
2	$\{x+1\}$
3	$\{x+1\}$
4	$\{x+1\}$

Offenbar

... kann schleifen-invarianter Code entstehen ;-(

... kann Code dupliziert werden :-(

... kann eine Verschlechterung eintreten !!!

... kann diese Verschlechterung im Beispiel durch
(Schleifen-Rotation und) PRE wieder beseitigt werden :-))

Immer?

Fazit:

→ Das Design einer sinnvollen Optimierung ist nicht ganz einfach.

→ Optimierungen, die ein Programm verbessern,
können andere dramatisch verschlechtern :-(

→ Manche Transformationen entfalten ihre Wirkung erst
in Verbindung mit weiteren Optimierungen :-)

→ Die Reihenfolge der angewandten Optimierungen ist entscheidend !!

→ Manche Optimierungen können iteriert angewandt werden !!!

... eine sinnvolle Abfolge:

T5	Konstanten-Propagation Intervall-Analyse Alias-Analyse
T7	Schleifen-Rotation
T1	Hilfsvariablen für Ausdrücke
T2	verfügbare Ausdrücke
T4	Move-Optimierung
T3	tote Zuweisungen
T8	partiell toter Code
T6	partiell redundanter Code

Bemerkung zu basic blocks: Diese sind eine Folge von Zuweisungen ohne Verzweigung.
Bei den Analysen eventuell schwieriger zu handhaben.
Zu der Abfolge der Analysen:
T5: Hier möglicherweise auch andere Abfolge dieser Analysen.
T7: Wenn möglich auf der Ebene des Programmcodes durchzuführen. Falls man auf dem Source-code arbeitet, dann evtl. zuerst.

2 Ersetzung teurer Berechnungen durch billigere

2.1 Reduction of Strength

(1) Polynom-Berechnung

$$f(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x + a_0$$

	Multiplikationen	Additionen
naiv	$\frac{1}{2}n(n+1)$	n
Wiederverwendung	$2n-1$	n
Horner-Schema	n	n

Strenght: Stärke des Operators. Reduction of strenght : Hier Ersetzen der Potenzierung durch schrittweise Multiplikation. Ersetzen eines teuren Operators durch einen billigeren.

Idee:

$$f(x) = (\ldots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2})\ldots) \cdot x + a_0$$

(2) Tabellierung eines Polynoms $f(x)$ vom Grad n :

→ Für jeden Wert $f(x)$ neu auszuwerten ist zu teuer :-)

→ Glücklicherweise sind die n-ten Differenzen konstant !!!

Beispiel: $f(x) = 3x^3 - 5x^2 + 4x + 13$

x	$f(x)$	Δ	Δ^2	Δ^3
0	13	2	8	18
1	15	10	26	
2	25	36		
3	61			
4	...			

Dabei ist die n-te Differenz stets

$$\Delta_h^n(f) = n! \cdot a_n \cdot h^n \qquad (h \text{ Schrittweite})$$

Kosten:

- n mal f auswerten;
- $\frac{1}{2} \cdot (n-1) \cdot n$ Subtraktionen, um die Δ^k zu berechnen;
- $2n-2$ Multiplikationen, um $\Delta_h^n(f)$ zu berechnen;
- n Additionen für jeden weiteren Wert :-)

$\Longrightarrow$

Anzahl der Multiplikationen hängt nur von n ab :-))

Einfacher Fall: $f(x) = a_1 \cdot x + a_0$

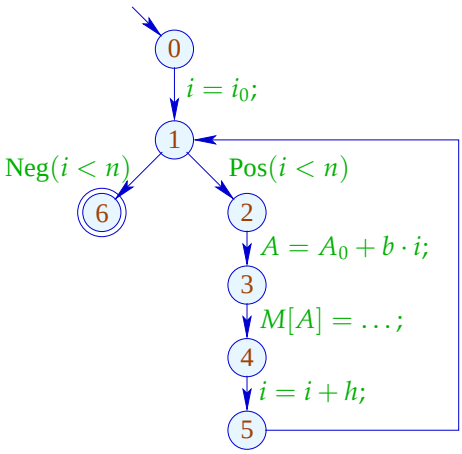
- ... kommt in vielen numerischen Schleifen vor :-)
  Diese Art von Funktionen kommen oft bei der Berechnung von Adressen vor.
  (Hier : a_1 Skalierung, x Laufindex, a_0 Anfangsadresse).
  Die Multiplikation würde man gerne sparen.
- Die ersten Differenzen sind bereits konstant:

$$f(x+h) - f(x) = a_1 \cdot h$$

- Anstelle einer Folge: $y_i = f(x_0 + i \cdot h), \quad i \geq 0$
  berechnen wir: $y_0 = f(x_0), \quad \Delta = a_1 \cdot h$
  $y_i = y_{i-1} + \Delta, \quad i > 0$

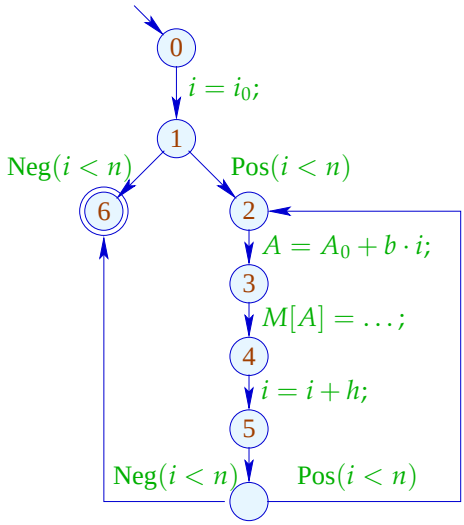
Beispiel:

for $(i = i_0; i < n; i = i + h)$ {
  $A = A_0 + b \cdot i;$
  $M[A] = \ldots;$
}



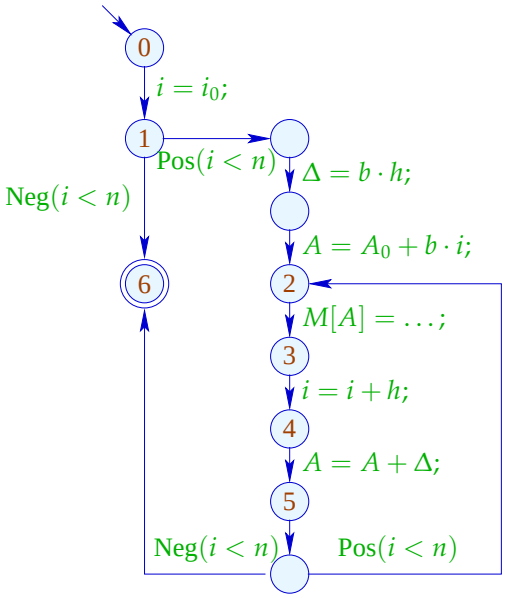
... bzw. nach Schleifen-Rotation:

$i = i_0;$
if $(i < n)$ do $\{$
$\quad A = A_0 + b \cdot i;$
$\quad M[A] = \ldots;$
$\quad i = i + h;$
$\}$ while $(i < n);$



... und Reduktion der Stärke:

$i = i_0;$
if $(i < n)$ $\{$
$\quad \Delta = b \cdot h;$
$\quad A = A_0 + b \cdot i_0;$
$\quad$ do $\{$
$\quad\quad M[A] = \ldots;$
$\quad\quad i = i + h;$
$\quad\quad A = A + \Delta;$
$\quad \}$ while $(i < n);$
$\}$



Achtung:

- Die Werte b, h, A_0 dürfen sich in der Schleife nicht ändern.
- i, A dürfen nur genau an einer Stelle in der Schleife modifiziert werden :-(

- Man könnte versuchen, die Variable i ganz einzusparen :

  → i darf sonst nicht weiter benutzt werden.
  → Man muss die Initialisierung transformieren in: $A = A_0 + b \cdot i_0$.
  → Man muss die Schleifenbedingung $i < n$ transformieren in: $A < N$ für $N = A_0 + b \cdot n$.
  → b muss ungleich Null sein !!!

Vorgehen:

Identifizieren von

... Schleifen;
... Iterations-Variablen;
... Konstanten;
... den richtigen Benutzungs-Strukturen.

Schleifen:

Praedomininierung: Ein Knoten v praedominiert u, wenn jeder Pfad zu u durch v geht.
Postdominierung : Ein Knoten u postdominiert v, wenn jeder Pfad von v durch u geht.

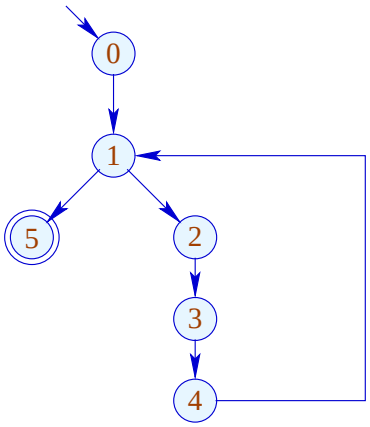
... identifizieren wir durch einen Punkt v, zu dem ein Rücksprung $(\_, \_, v)$ existiert :-)

Für den Teilgraphen G_v des CFG auf $\{w \mid v \Rightarrow w\}$ definieren wir:

$$\mathsf{Loop}[v] \quad = \quad \{w \mid w \rightarrow^* v \ \text{ in } \ G_v\}$$

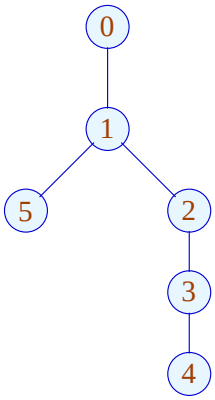
Beispiel:

Kontrollflussgraph



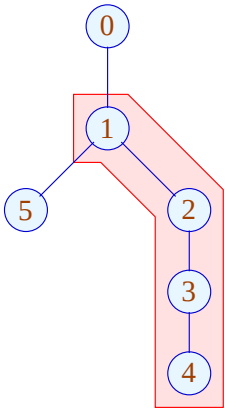
	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Zugehöriger Praedominatorbaum



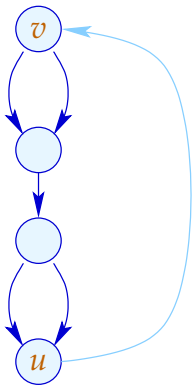
	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Somit findet man diesen Schleifenbereich



	$\mathcal{P}$
0	$\{0\}$
1	$\{0,1\}$
2	$\{0,1,2\}$
3	$\{0,1,2,3\}$
4	$\{0,1,2,3,4\}$
5	$\{0,1,5\}$

Wir sind an Kanten interessiert, die pro Iteration exakt einmal ausgeführt werden:



Das ist graphentheoretisch nicht ganz leicht auszudrücken :-(

Man könnte solche Kanten k selektieren, dass:

- der Teilgraph $G = \mathsf{Loop}[v] \backslash \{(\_, \_, v)\}$ zusammenhängend ist;
- der Graph $G \backslash \{k\}$ in zwei unverbundene Teilgraphen zerfällt.

Auf der Source-Programm-Ebene ist das dagegen trivial:

$$\begin{array}{l} \mathsf{do}\ \{\ s_1 \ldots s_k \\ \}\ \mathsf{while}\ (e); \end{array}$$

Die gesuchten Zuweisungen müssen unter den s_i sein :-)

Iterationsvariable:

i heißt Iterationsvariable, wenn die einzige Definition von i in der Schleife an einer Kante erfolgt, die den Rumpf separiert, und von der Form:

$$i = i + h;$$

ist für eine Schleifen-Konstante h .

Eine Schleifen-Konstante ist einfach eine Konstante (z.B. 42) oder, etwas liberaler, ein Ausdruck, der nur von Variablen abhängt, die innerhalb der Schleife nicht modifiziert werden :-)

Achtung: Die Modifikation der Schleifenvariablen muss am Ende des Rumpfes stattfinden. (Ansonsten : Fehler (sog. +1,- 1 -Fehler) in der Transformation möglich.)

(3) Differenzen für Mengen

Wir wenden nun ein Optimierungsprinzip auf die Fixpunktiteration an. Sie findet in der Informatik oft Anwendung, insbesondere bei deduktiven Datenbanken; (Datalog)
Hier sog. semi-naive Iteration
z.B. $x \supseteq Fx$
Von solch einer Ungleichnung möchte man den Fixpunkt berechnen.
Man rechnet in diesem Fall mit Mengen. F ist ein einstelliger Operator.

Betrachte die Fixpunkt-Berechnung:

$$\begin{array}{l} x = \emptyset; \\ \text{for } (t = F\,x; t \not\subseteq x; \boxed{t = F\,x;}) \\ \qquad x = x \cup t; \end{array}$$

Dies ist i.A. ineffizient, da F eventuell eine Funktion ist, deren Kosten proportional zur Menge sind.

F sollte eben nur auf die neu hinzugekommenen Elemente angewandt werden.
Man geht also inkrementell vo:
Die Nachfolger der in der letzen Iteration neu hinzugekommenen Elementen ermitteln .

Anwendung : Reachablity für endliche Automaten.
(Model-checking, Ermittlung aller erreichbaren Zustände)

Ist F distributiv, könnte man sie ersetzen durch:

$$\begin{array}{l} x = \emptyset; \\ \text{for } (\Delta = F\,x; \Delta \neq \emptyset; \boxed{\Delta = (F\,\Delta) \setminus x;}) \\ \qquad x = x \cup \Delta; \end{array}$$

Die Funktion F muss jetzt nur noch für die kleineren Mengen Δ ausgerechnet werden :-)
semi-naive Iteration

Statt der Folge: $\emptyset \subseteq F(\emptyset) \subseteq F^2(\emptyset) \subseteq \ldots$
berechnen wir: $\Delta_1 \cup \Delta_2 \cup \ldots$
wobei:

$$\begin{array}{rcl} \Delta_{i+1} & = & F(F^i(\emptyset)) \backslash F^i(\emptyset) \\ & = & F(\Delta_i) \backslash (\Delta_1 \cup \ldots \cup \Delta_i) \quad \text{mit } \Delta_0 = \emptyset \end{array}$$

Nehmen wir an, die Kosten von $F\,x$ seien $1 + \#x$.
Dann summieren sich die Kosten zu:

naiv	$1+2+\ldots+n+n \quad = \quad \frac{1}{2}n(n+3)$	
semi-naiv		$2n$

wobei n die Kardinalität des Ergebnisses ist.

$\Longrightarrow$ Man spart einen linearen Faktor :-)

Der Unterschied ist also ca. Faktor 4. Bei normaler Fixpuntiteration ergibt sich auch noch ein grösserer Verwaltungsaufwand als bei semi-naiver Iteration.
Also in der Praxis nur bei extremen Beispielen effiziente Verbesserung.
Bei Model-checking ergeben sich Vorteile bei semi-naiver Iteration

2.2 Peephole Optimierung

Idee:

- Schiebe ein kleines Fenster über das Programm.
- Optimiere agressiv innerhalb des Fensters. D.h.:
  - $\rightarrow$ Beseitige Redundanzen!
  - $\rightarrow$ Ersetze innerhalb des Fensters teure Operationen durch billige!

Beispiele:

$x = x + 1;$ $\Longrightarrow$ $x{+}{+};$

// sofern es dafür eine spezielle Instruktion gibt :-)

$z = y - a + a;$ $\Longrightarrow$ $z = y;$

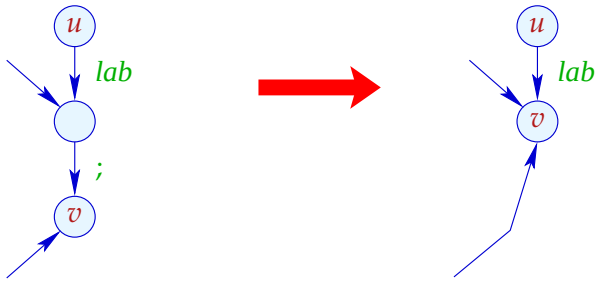
// algebraische Umformungen :-)

$x = x;$ $\Longrightarrow$ $;$

$x = 0;$ $\Longrightarrow$ $x = x \oplus x;$

$x = 2 \cdot x;$ $\Longrightarrow$ $x = x + x;$

Wichtiges Teilproblem: *nop*-Optimierung



→ Ist $(v_1, ;, v)$ eine Kante, hat v_1 keine weitere ausgehende Kante.

→ Folglich dürfen wir v_1 und v identifizieren :-)

→ Die Reihenfolge der Identifizierungen ist egal :-))

Implementierung:

Da sich NOP-kanten in Kontrollflussgraphen nicht verzweigen, kann next rekursiv definiert werden.

- Wir konstruieren eine Funktion $\mathsf{next} : \mathit{Nodes} \to \mathit{Nodes}$ mit:

$$\mathsf{next}\ u = \begin{cases} \mathsf{next}\ v & \text{falls } (u, ;, v) \text{ Kante} \\ u & \text{sonst} \end{cases}$$

  Achtung: Diese Definition ist nur rekursiv, wenn es ;-Schleifen gibt ???

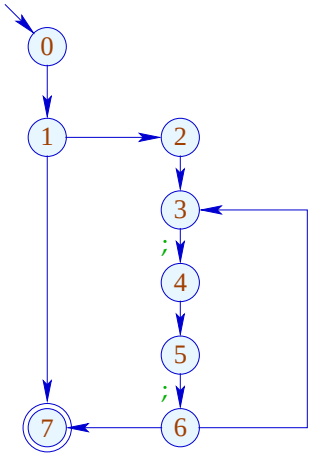
- Wir ersetzen jede Kante:

$$(u, lab, v) \quad \Longrightarrow \quad (u, lab, \mathsf{next}\ v)$$

  ... sofern $lab \neq ;$

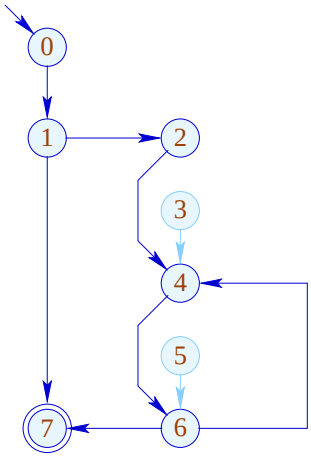
- Alle ;-Kanten werfen wir weg ;-)

Beispiel:



$$\begin{aligned} \mathsf{next}\ 1 &= 1 \\ \mathsf{next}\ 3 &= 4 \\ \mathsf{next}\ 5 &= 6 \end{aligned}$$

Nach der Transformation:
(auch die unerreichbaren Hilfsknoten werden eliminiert)



$$\begin{aligned} \mathsf{next}\ 1 &= 1 \\ \mathsf{next}\ 3 &= 4 \\ \mathsf{next}\ 5 &= 6 \end{aligned}$$

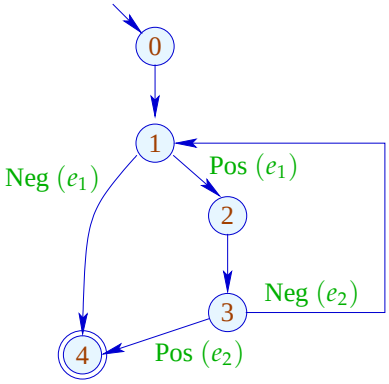
2. Teilproblem: Linearisierung

Der CFG muss nach der Optimierung wieder in eine lineare Abfolge von Instruktionen gebracht werden :-)

Achtung:

Nicht jede Linearisierung ist gleich gut !!!

Beispiel:

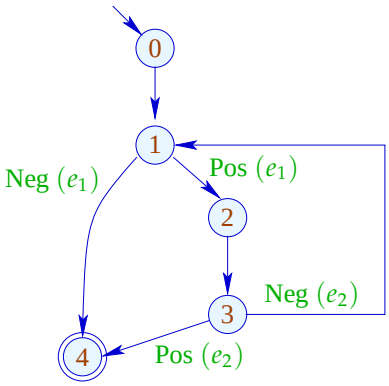



```
0:
1:   if (e1) goto 2;
4:   halt
2:   Rumpf
3:   if (e2) goto 4;
     goto 1;
```

Schlecht: Der Schleifen-Rumpf wird angesprungen :-(

Der Schleifenrumpf wird mit einem Sprung betreten.
Jede zusätzliche Operation im Schleifenrumpf ist zu vermeiden. Besser : Jump at false

Bessere Lösung:




```
0:
1:   if (!e1) goto 4;
2:   Rumpf
3:   if (!e2) goto 1;
4:   halt
```

// besseres Cache-Verhalten :-)

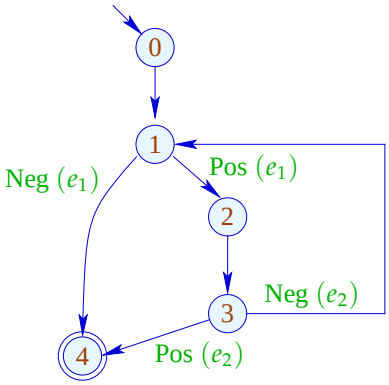
Wie erkenne ich, ob in die Schleife, oder aus der Schleife gesprungen wird ?

Idee:

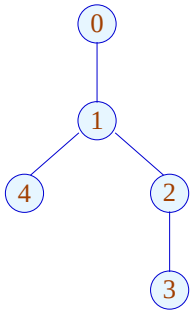
- Gib jedem Knoten eine Temperatur!
- Springe stets zu
  - (1) bereits behandelten Knoten;
  - (2) kälteren Knoten.
- Temperatur $\approx$ Schachtelungstiefe

Zur Berechnung benutzen wir den Prädominator-Baum
und starke Zusammenhangskomponenten ...

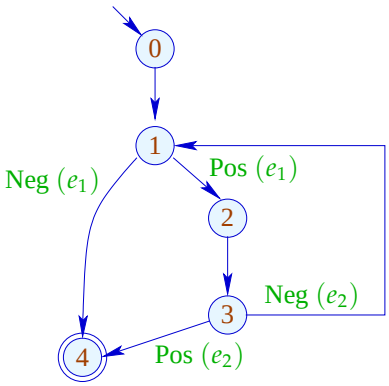
... Beispiel:



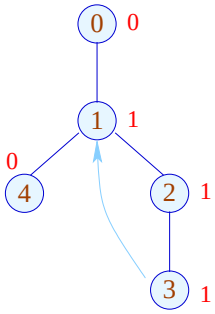
Praedominatorbaum



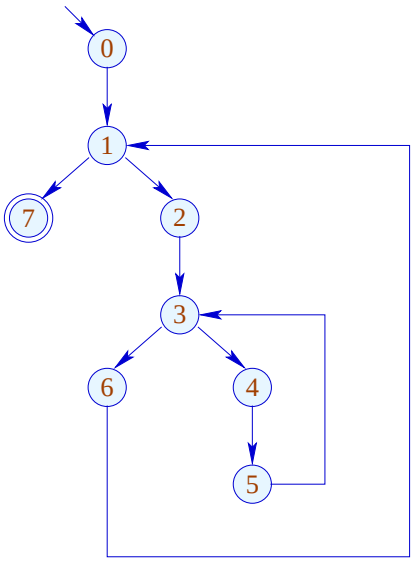
Der Teilbaum mit Rücksprung ist heißer ...



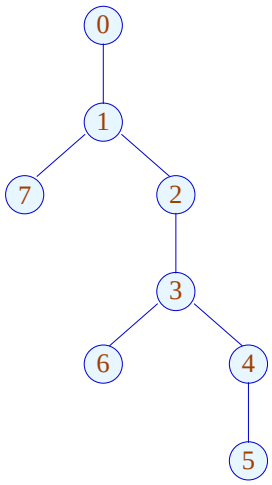
Praedominatorbaum

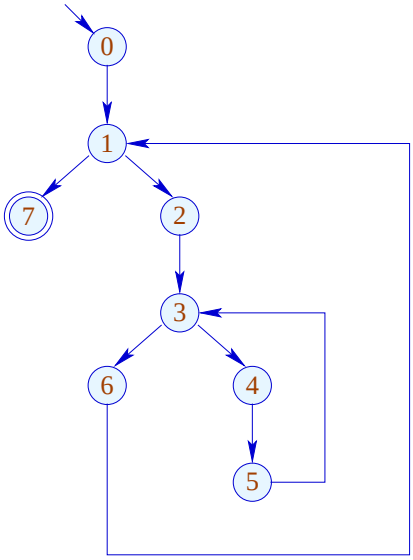


Komplizierteres Beispiel:

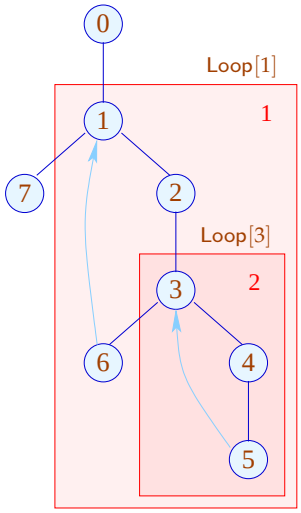


Praedominatorbaum

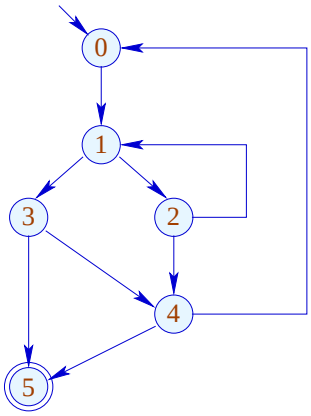




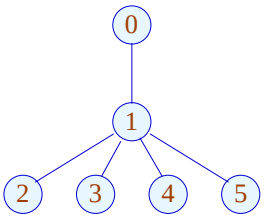
Praedominatorbaum



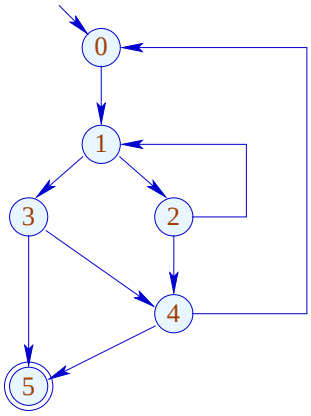
Kompliziertere Struktur mit do-while-Schleifen und break



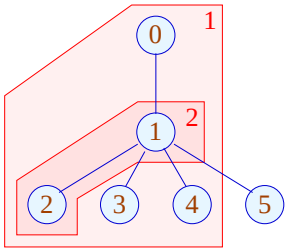
Praedominatorbaum



Unsere Definition von Loop sorgt dafür,
dass (erkannte) Schleifen geschachtelt auftreten :-)
Sie ist auch für do-while-Schleifen mit breaks vernünftig...



Praedominatorbaum



Man erkennt also tatsächlich mit dem Verfahren 2 geschachtelte Schleifen.

Zusammenfassung: Das Verfahren

(1) Ermittlung einer Temperatur für jeden Knoten;

(2) Prä-order-DFS über den CFG;

→ Führt eine Kante zu einem Knoten, für den wir bereits Code erzeugt haben, fügen wir einen Sprung ein.

→ Hat ein Knoten zwei Nachfolger unterschiedlicher Temperatur, fügen wir einen Sprung zum kälteren der beiden ein.

→ Hat ein Knoten zwei gleich warme Nachfolger, ist es egal ;-)

2.3 Funktionen

Wir erweitern unsere Mini-Programmiersprache um Funktionen und Funktions-Aufrufe.
Dazu führen wir neue Statements ein:

$$\mathsf{ret} = f(b_1, \ldots, b_k); \qquad \mathsf{return}\ e;$$

Jede Funktion f besitzt eine Definition:

$$f\,(a_1, \ldots, a_n)\ \{\ stmt^*\ \}$$

// a_i formale Parameter
// b_i aktuelle Parameter
// ret Register für Rückgabewert

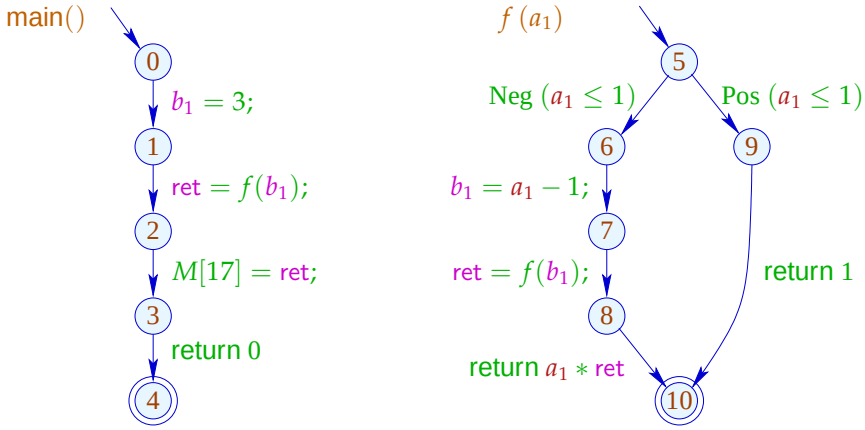
Die Programm-Ausführung startet mit dem
Aufruf einer (parameterlosen) Funktion main () .

Beispiel: *Fakultätsfunktion*

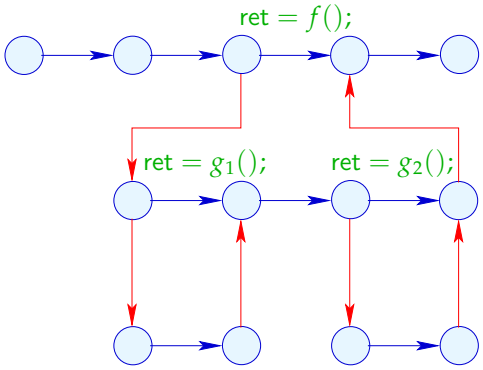
```
main () {                    f (a1) {
    b1 = 2;                      if (a1 ≤ 1) return 1;
    ret = f(b1);                   b1 = a1 − 1;
    M[17] = ret;                 ret = f(b1);
    return 0;                    return a1 · ret;
}                            }
```

Solche Programme lassen sich durch eine Menge von CFGs darstellen:
einem für jede Funktion ...

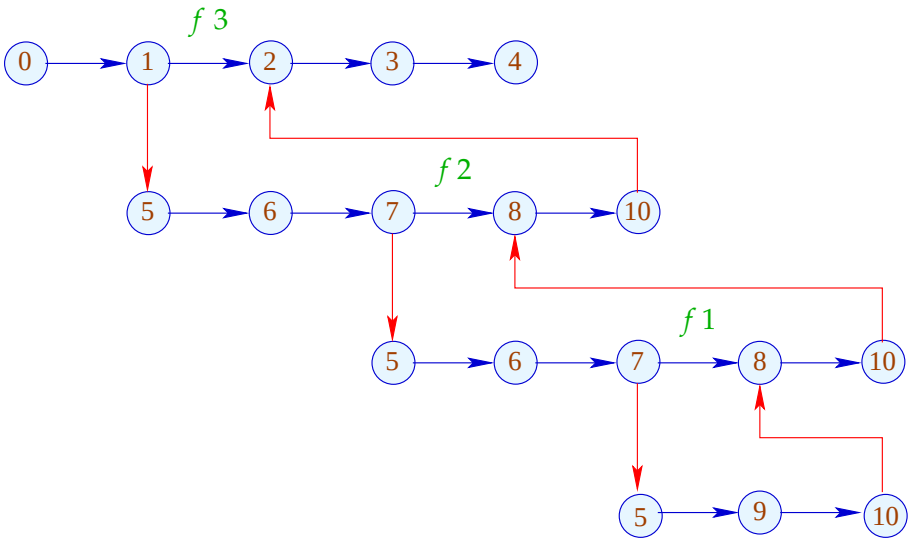
... im Beispiel:



Um solche Programme zu optimieren,
benötigen wir eine erweiterte operationelle Semantik ;-)
Programm-Ausführungen sind nicht mehr Pfade, sondern Wälder:
Man hat also eine Folge von Bäumen (diese sind Rekursionswälder, welche die rekursiven Aufrufe repräsentieren)



Beispiel:Fakultät



Die Funktion $[\![.]\!]$ erweitern wir auf Berechnungs-Wälder w :

$$[\![w]\!] : (Vars \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (Vars \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})$$

Für einen Aufruf $k = (u, \mathsf{ret} = f\ (b_1, \ldots, b_k);, v)$ müssen wir:

- die Anfangwerte der lokalen Variablen ermitteln:

$$\mathsf{enter}_k\ \rho\ x = \begin{cases} \rho\, b_i & \text{falls} \quad x = a_i \\ 0 & \text{sonst} \end{cases}$$

- ... den berechneten Rückgabe-Wert in ret ablegen:

$$\mathsf{combine}\ (\rho_1, \rho_2) = \rho_1 \oplus \{\mathsf{ret} \mapsto \rho_2\ \mathsf{ret}\}$$

enter ändert also die Variablenbelegung vor dem Aufruf in die Variablenbelegung beim Aufruf (Also beim Betreten des Rumpfs)
Spezielle Konvention: Alle anderen Variablen werden mit 0 initiiert.

- ... dazwischen den Berechnungs-Wald der Funktion auswerten:

$$\begin{aligned} [\![k\ \langle w \rangle]\!]\ (\rho, \mu) \quad = \quad & \\ & \mathsf{let} \quad (\rho_1, \mu_1) = [\![w]\!]\ (\mathsf{enter}_k\ \rho, \mu) \\ & \mathsf{in} \quad (\mathsf{combine}\ (\rho, \rho_1), \mu_1) \end{aligned}$$

Ein Return $k = (u, \mathsf{return}\ e;, v)$ ist eine Zuweisung an ret :

$$[\![k]\!]\ (\rho, \mu) \quad = \quad (\rho \oplus \{\mathsf{ret} \mapsto [\![e]\!]\ \rho\}, \mu)$$

Achtung:

- $[\![w]\!]$ ist i.a. nur partiell definiert :-)
- Die Benutzung von speziellen Registern a_i, b_i, ret repräsentiert eine bestimmte Aufruf-Konvention.
- Die normale operationelle Semantik arbeitet mit Konfigurationen, die Aufrufkeller verwalten.
- Berechnungs-Wälder eignen sich aber besser zur Konstruktion von Analysen und Korrektheitsbeweisen :-)

- Es ist eine lästige (aber nützliche) Aufgabe,
  die Äquivalenz der beiden Ansätze zu zeigen ...

Konfigurationen:

$$
\begin{array}{lcl}
\mathit{configuration} & = & \mathit{stack} \times \mathit{store} \\
\mathit{store} & = & \mathbb{N} \to \mathbb{Z} \\
\mathit{stack} & = & \mathit{frame} \cdot \mathit{frame}^* \\
\mathit{frame} & = & \mathit{point} \times \mathit{locals} \\
\mathit{locals} & = & (\mathit{Vars} \to \mathbb{Z})
\end{array}
$$

Ein *frame* (Kellerrahmen)
beschreibt den lokalen Berechnungszustand innerhalb eines Funktionsaufrufs :-)
Den Rahmen des aktuellen Aufrufs schreiben wir links.
Berechnungsschritte beziehen sich auf den aktuellen Aufruf :-)
Zusätzlich benötigte Arten von Schritten:

Aufruf $k = (u, \mathsf{ret} = f\,(b_1, \ldots, b_k);, v)$:

$$
(\boxed{(u, \rho)} \cdot \sigma, \mu) \quad \Longrightarrow \quad (\boxed{(u_f, \mathsf{enter}_k\ \rho) \cdot (v, \rho)} \cdot \sigma, \mu)
$$

u_f Anfangspunkt von f

Rückkehr:

$$
(\boxed{(r_f, \rho_2) \cdot (v, \rho_1)} \cdot \sigma, \mu) \quad \Longrightarrow \quad (\boxed{(v, \mathsf{combine}\,(\rho_1, \rho_2))} \cdot \sigma, \mu)
$$

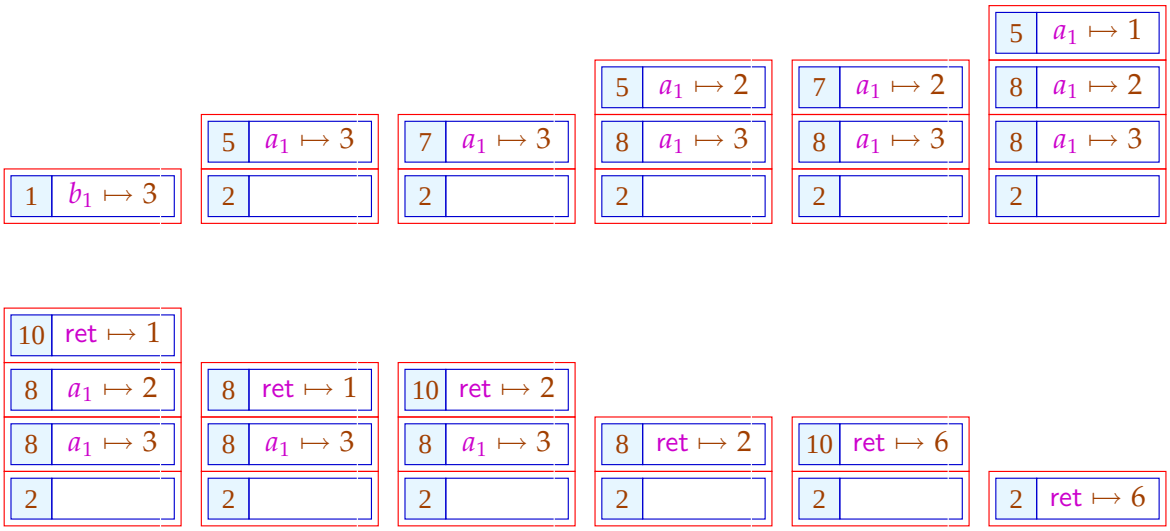
r_f Endpunkt von f

Rückgabe: $k = (u, \mathsf{return}\ e;, v)$:

$$
(\boxed{(u, \rho)} \cdot \sigma, \mu) \quad \Longrightarrow \quad (\boxed{(v, \rho \oplus \{\mathsf{ret} \mapsto [\![e]\!]\ \rho\})} \cdot \sigma, \mu)
$$

Mit dem Aufruf-Keller verwalten wir explizit den DFS-Durchlauf
über den Berechnungswald :-)

... im Beispiel:



Diese operationelle Semantik ist einigermaßen realistisch :-)

Vorteil dieser operationellen Semantik: Die Kosten eines Funktionsaufrufs sind bestimmbar. (Diese sind aber relativ teuer wegen:)

Kosten eines Prozedur-Aufrufs:

Vor Betreten des Rumpfs:
- Anlegen eines Kellerrahmens;
- Retten der Register;
- Retten der Fortsetzungsaddresse;
- Anspringen des Rumpfs.

Bei Beenden des Aufrufs:
- Aufgeben des Kellerrahmens;
- Restaurieren der Register;
- Übergeben des Ergebnisses;
- Rücksprung hinter die Aufrufstelle.

$\Longrightarrow$ ... ziemlich teuer !!!

Bemerkung: Maschinen mit einer grossen Anzahl von Registern sind im Prinzip gut, aber bei Funktionsaufrufen wegen der Rettung der Register prinzipiell teuer.Es gibt auch Hardwareunterstützung für Prozeduraufrufe in bestimmten Architekturen.(SPARC : Registerstacks!)

1. Idee: Inlining

Kopiere den Funktionsrumpf an jede Aufrufstelle !!!

Bemerkung: Der Programmierer führt Funktionen ein, um das Programm zu strukturieren, bzw. modularisieren. Auch ist die Beseitigung von Fehlern in Programmen mit Funktionen einfacher.
Der Compiler benötigt diese visuelle Struktur natürlich nicht und optimiert indem er den Funktionsrumpf an die Aufrufstelle kopiert.

Beispiel:

```
abs (a1) {                    max (a1, a2) {
    b1 = a1;                      if (a1 < a2) return a2;
    b2 = −a1;                     return a1;
    ret = max (b1, b2);       }
    return ret;
}
```

... liefert:

```
abs (a1) {
        b1 = a1;
        b2 = −a1;
        a1 = b1;
        a2 = b2;
        if (a1 < a2) { ret = a2; goto _max; }
        ret = a1; goto _max;
_max :  return ret;
}
```

$a_1 = b_1; a_2 = b_2;$
Vor der Kopie des Rumpfes, Zuweisung der aktuellen Parameter an die formalen Parameter.
Problem: Ein Konflikt ist möglich, da Variablen aus unterschiedlichen Sichtbarkeitsbereichen kommen können.

Probleme:

Variablenverschattung.

- Der einkopierte Block modifiziert evt. die a_i, b_i :-(
- Allgemeiner: Mehrfachbenutzung gleicher Variablennamen kann zu Fehlern führen;
- Mehrfach-Verwendung einer Funktion führt zu Code-Duplizierung :-((
  Es gibt auch Compiler die inverses inlining durchführen, also wenn identische Codestücke gefunden werden, werden an diesen Stellen richtige Funktionen erzeugt (function extraction).
  Dies ist natürlich nicht möglich bei rekursiven Funktionen.

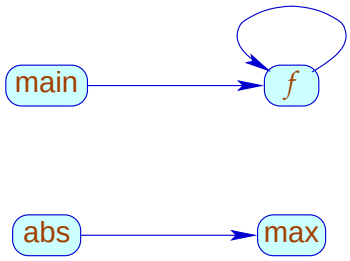
- Wie gehen wir mit Rekursion um ???

Erkennen von Rekursion:

Wir konstruieren den Aufruf-Graph des Programms.

Erkennen der Rekursion durch die starken Zusammenhangskomponenten.

In den Beispielen:



Aufruf-Graph:

Der Aufruf-Graph ist ein Problem in C: Es gibt Pointer und eben auch Funktionspointer.
Eine Analyse zur Erkennung von Funktionspointern ist nötig.

- Die Knoten sind die Funktionen.
- Eine Kante geht von g nach h, sofern der Rumpf von g einen Aufruf von h enthält.

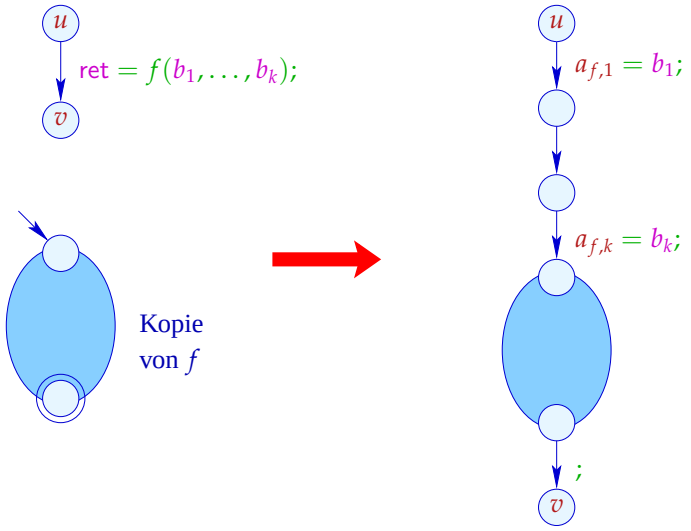
Strategien für Inlining:

- Kopiere nur Blatt-Funktionen ein, d.h. solche ohne weitere Aufrufe :-)
- Kopiere sämtliche nicht-rekursiven Funktionen ein!

... wir betrachten hier nur Blatt-Funktionen ;-)

Es ist anzumerken, dass i.A. Optimierungen dort verstärkt angewendet werden sollten die sehr rechenintensiv sind.

Transformation 10:



Beachte:

- Blatt-Funktionen benutzen keine Variablen b_i :-)
- Die Variable ret der ein-kopierten Funktion wird mit der Variablen ret der aufrufenden identifiziert.

  Das spart uns eine Umspeicherung :-)
- Die Nop-Kante können wir ebenfalls einsparen, da der *stop*-Knoten von f selbst keine ausgehenden Kanten hat ...
- Die $a_{f,i}$ sind die Kopien der aktuellen Parameter von f.
- Eigentlich müssten wir alle anderen Variablen von f noch mit 0 initialisieren :-)

2. Idee: Beseitigung von Endrekursion *Tail-rekursion*

Ein Funktionsaufruf liefert ein Ergebnis, welches direkt der Rückgabewert der aufrufenden Funktion ist. Danach darf kein weiterer Befehl folgen. Somit kann der letzte Funktionsaufruf in den gleichen Kellerrahmen gelegt werden. Es muss also kein neuer Kellerrahmen angelegt werden.

Betrachte das folgende Beispiel: *Fakultät, Aufruf z.B. f(1,20)*

$$
\begin{array}{l}
f\,(a_1, a_2)\ \{ \\
\qquad \text{if}\ \ (a_2 \leq 1)\ \ \text{return}\ \ a_1; \\
\qquad b_1 = a_1 \cdot a_2; \\
\qquad b_2 = a_2 - 1; \\
\qquad \text{ret} = f\,(b_1, b_2); \\
\qquad \text{return}\ \ \text{ret}; \\
\}
\end{array}
$$

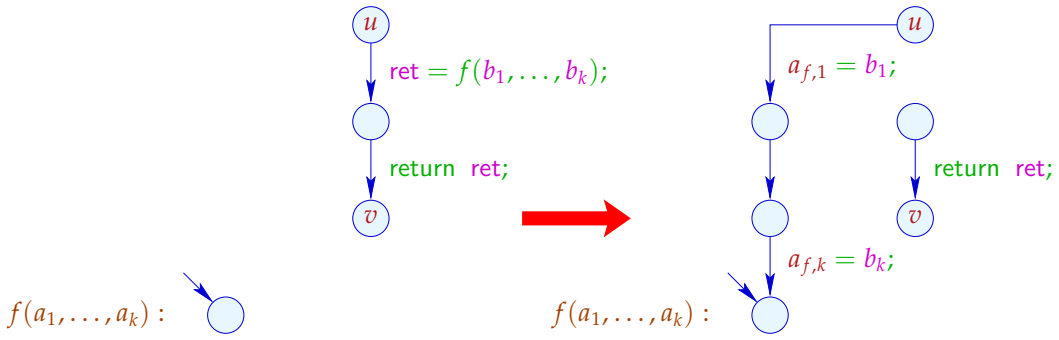
Der Funktionsaufruf im Rumpf liefert das Ergebnis.
Nach dem Funktionsaufruf gibt es im Rumpf nichts mehr zu tun.
$\Longrightarrow$ Wir könnten ihn direkt anspringen :-)
Man kann also den rekursiven Aufruf durch einen Sprung an den Anfang des Rumpfes ersetzen. (Es ergibt sich eine Schleifenstruktur) Man muss aber die Parameter vor dem Rücksprung richtig besetzen.

... das liefert im Beispiel:

$$
\begin{array}{l}
f\,(a_1, a_2)\ \{ \\
\ \_f: \ \ \text{if}\ \ (a_2 \leq 1)\ \ \text{return}\ \ a_1; \\
\qquad b_1 = a_1 \cdot a_2; \\
\qquad b_2 = a_2 - 1; \\
\qquad a_1 = b_1; \\
\qquad a_2 = b_2; \\
\qquad \text{goto}\ \ \_f; \\
\}
\end{array}
$$

// Das funktioniert, weil wir Referenzen auf Variablen
// verbieten. *In C nicht möglich*

Transformation 11:



Achtung:

- → Diese Optimierung ist besonders wichtig bei Programmiersprachen ohne Iterationskonstrukte !!!
- → Duplizieren von Code ist nicht erforderlich :-)
- → Variablen brauchen nicht umbenannt zu werden :-)
- → Die Optimierung hilft auch bei nicht-rekursiven Endaufrufen :-)
- → Der entstehende Code kann Sprünge aus dem Rumpf einer Funktion in eine andere enthalten ???

Exkurs 4: Interprozedurale Analyse

Vom Ansatz her komplizierter als **intra***prozedurale Analyse.*
Operationelle Semantik : Bei Funktionen gibt es potentiell unendlich viel Aufrufkeller.

Bisher können wir nur jede Funktion einzeln analysieren.

→ Die Kosten halten sich in Grenzen :-)

→ Die Techniken funktionieren auch bei getrennter Übersetzung :-)

→ An Funktionsaufrufen müssen wir das Schlimmste annehmen :-(
Alle Daten die von der aufgerufenen Funktion erreichbar sind, müssen auf Top gesetzt werden.

→ Konstantenpropagation funktioniert nur für lokale Konstanten :-((

Frage: Wie analysiert man rekursive Programme ???

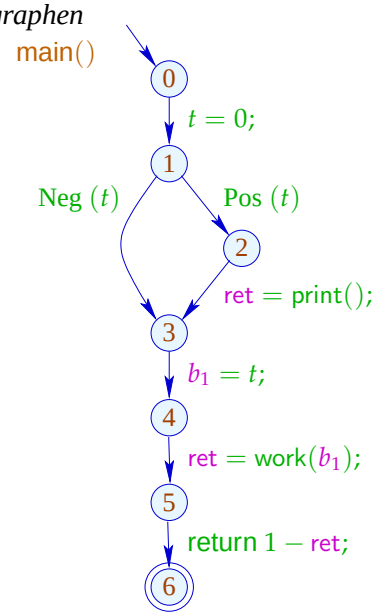
Beispiel: Konstantenpropagation *Als Beispiel für* **eine** *Optimierung!*

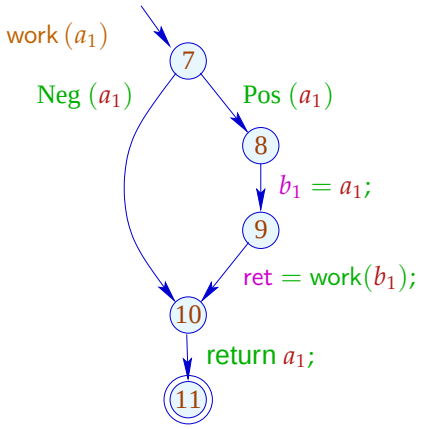
work ist rekursiv, nicht endrekursiv.(sonst ja inlining möglich) t ist so eine Art Schaltvariable für debugging

```
main() {
    t = 0;
    if (t) print();
    b1 = t;
    ret = work(b1);
    return 1 − ret;
}
```

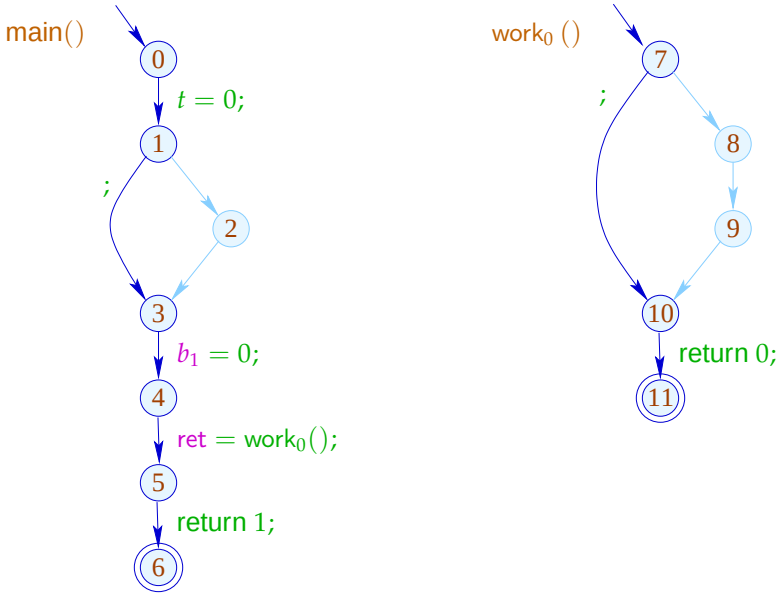
```
work(a1) {
    if (a1) {
        b1 = a1;
        work(b1);
    }
    return a1;
}
```

Kontrollflussgraphen





Es ergibt sich nach Konstantenpropagation und inlining:



Nach der Optimierung wird $work_0()$ kreiert. work sollte aufgehoben werden!

(1) Funktionaler Ansatz:

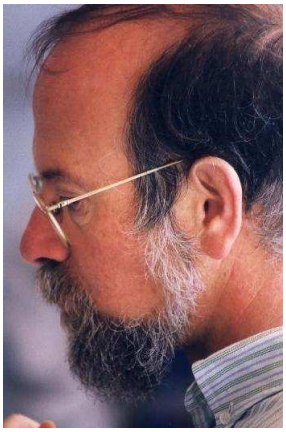
Sei $\mathbb{D}$ ein vollständiger Verband von (abstrakten) Zuständen.

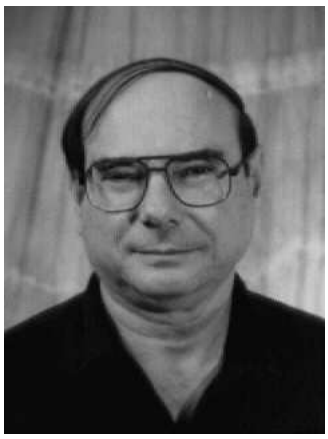
Idee:

Eine Funktion transformiert also den Zustand vor dem Aufruf in einen Zustand nach dem Aufruf der Funktion

Repräsentiere den Effekt von $f()$ durch eine Funktion:

$$[\![f]\!]^\sharp \; : \; \mathbb{D} \to \mathbb{D}$$





Micha Sharir, Tel Aviv University Amir Pnueli, Weizmann Institute

Um den Effekt einer Aufrufskante $k = (u, \mathsf{ret} = f\,(b_1, \ldots, b_k);, v)$ zu ermitteln, benötigen wir abstrakte Funktionen:

$$\begin{array}{lcl} \mathsf{enter}_f^\sharp & : & \mathbb{D} \to \mathbb{D} \\ \mathsf{combine}^\sharp & : & \mathbb{D}^2 \to \mathbb{D} \end{array}$$

enter : Aktuelle Parameter $\longrightarrow$ formale Parameter,(Parameterübergabe]
combine : Rückgabewert der Funktion wird in die Variablenbelegung vor dem Aufruf der Funktion integriert.

Damit erhalten wir:

$$[\![k]\!]^\sharp\, D \;=\; \mathsf{combine}^\sharp\,(D, [\![f]\!]^\sharp\,(\mathsf{enter}_f^\sharp\, D))$$

... für Konstantenpropagation:

$$\begin{array}{lcl} \mathbb{D} & = & (\mathit{Vars} \to \mathbb{Z}^\top)_\perp \\ \mathsf{enter}_f^\sharp\, D & = & \begin{cases} \perp & \text{falls} \quad D = \perp \\ \{a_i \mapsto (D\ b_i) \mid i = 1, \ldots, k\} & \text{sonst} \end{cases} \\ \mathsf{combine}^\sharp\,(D_1, D_2) & = & \begin{cases} \perp & \text{falls} \quad D_1 = \perp \vee D_2 = \perp \\ D_1 \oplus \{\mathsf{ret} \mapsto (D_2\ \mathsf{ret})\} & \text{sonst} \end{cases} \end{array}$$

Um die Effekte $[\![f]\!]^\sharp$ zu ermitteln, stellen wir ein Constraint-System über dem vollständigen Verband $\mathbb{D} \to \mathbb{D}$ auf:

$$\begin{array}{lcll} [\![v]\!]^\sharp & \sqsupseteq & \mathsf{Id} & v \quad \text{Eintrittspunkt} \\ [\![v]\!]^\sharp & \sqsupseteq & [\![k]\!]^\sharp \circ [\![u]\!]^\sharp & k = (u, \_, v) \quad \text{Kante} \\ [\![f]\!]^\sharp & \sqsupseteq & [\![stop_f]\!]^\sharp & stop_f \quad \text{Endpunkt von} \quad f \end{array}$$

$[\![v]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ beschreibt den Effekt aller Präfixe der Berechnungswälder w einer Funktion, die vom Eintrittspunkt nach v führen :-)

Probleme:

- Wie beschreibt man eine Funktion $f : \mathbb{D} \to \mathbb{D}$???
- Ist $\#\mathbb{D} = \infty$, hat $\mathbb{D} \to \mathbb{D}$ unendliche aufsteigende Ketten :-(

Vereinfachung: Kopier-Konstanten

→ Bedingungen interpretieren wir wie ein ; :-)

→ Wir behandeln exakt nur Zuweisungen $x = e;$ mit $e \in Vars \cup \mathbb{Z}$:-)

Variablen können also nur noch den Wert von Konstanten (bzw. Variablen) annehmen, die schon im Programm stehen. Das sind endlich viele. Damit ergeben sich endliche Wertemengen.

Beobachtung:

→ Die Effekte von Zuweisungen sind:

$$[\![x = e;]\!]^\sharp\, D \;=\; \begin{cases} D \oplus \{x \mapsto c\} & \text{falls} \quad e = c \in \mathbb{Z} \\ D \oplus \{x \mapsto (D\ y)\} & \text{falls} \quad e = y \in Vars \\ D \oplus \{x \mapsto \top\} & \text{sonst} \end{cases}$$

→ Sei $\mathbb{V}$ die (endliche !!!) Menge der konstanten rechten Seiten.
Dann haben Variablen stets Werte aus $\mathbb{V}^\top$:-))

→ Die auftretenden Effekte sind enthalten in

$$\mathbb{D}_f \to \mathbb{D}_f \qquad \text{mit} \qquad \mathbb{D}_f = (Vars \to \mathbb{V}^\top)_\perp$$

→ Dieser Verband ist riesig, aber endlich !!!

Verbesserung:

→ Nicht alle Funktionen aus $\mathbb{D}_f \to \mathbb{D}_f$ kommen wirklich vor :-)

→ Alle vorkommenden Funktionen $\lambda D.\, \bot \neq M$ sind von der Form:

$$\begin{array}{lcll} M & = & \{x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} y) \mid x \in \mathit{Vars}\} & \text{wobei:} \\ M\ D & = & \{x \mapsto (b_x \sqcup \bigsqcup_{y \in I_x} D\ y) \mid x \in \mathit{Vars}\} & \text{für} \quad D \neq \bot \end{array}$$

→ Sei $\mathbb{M}$ die Menge aller dieser Funktionen. Dann gilt für $M_1, M_2 \in \mathbb{M}$ $(M_1 \neq \lambda D.\, \bot \neq M_2)$:

$$(M_1 \sqcup M_2)\ x \quad = \quad (M_1\ x) \sqcup (M_2\ x)$$

→ Für $k = \#\mathit{Vars}$ hat $\mathbb{M}$ die Höhe $(k+1) \cdot k + 1$:-)

→ Auch die Komposition lässt sich direkt implementieren:

$$\begin{array}{rcll} (M_1 \circ M_2)\ x & = & b' \sqcup \bigsqcup_{y \in I'} y & \text{mit} \\ b' & = & b \sqcup \bigsqcup_{z \in I} b_z & \\ I' & = & \bigcup_{z \in I} I_z & \text{sofern} \\ M_1\ x & = & b \sqcup \bigsqcup_{y \in I} y & \\ M_2\ z & = & b_z \sqcup \bigsqcup_{y \in I_z} y & \end{array}$$

→ Die Effekte von Zuweisungen sehen dann so aus:

$$[\![x = e;]\!]^\sharp \quad = \quad \begin{cases} \mathsf{Id}_{\mathit{Vars}} \oplus \{x \mapsto c\} & \text{falls} \quad e = c \in \mathbb{Z} \\ \mathsf{Id}_{\mathit{Vars}} \oplus \{x \mapsto y\} & \text{falls} \quad e = y \in \mathit{Vars} \\ \mathsf{Id}_{\mathit{Vars}} \oplus \{x \mapsto \top\} & \text{sonst} \end{cases}$$

... im Beispiel:

$$\begin{array}{lcl} [\![t = 0;]\!]^\sharp & = & \{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}, \boxed{t \mapsto 0}\} \\ [\![b_1 = a_1;]\!]^\sharp & = & \{a_1 \mapsto a_1, \boxed{b_1 \mapsto a_1}, \mathsf{ret} \mapsto \mathsf{ret}, t \mapsto t\} \end{array}$$

Um die Analyse zu implementieren, müssen wir nur noch den Effekt eines Aufrufs $k = (\_, \mathsf{ret} = f\ (b_1, \ldots, b_k);, \_)$ aus dem Effekt der Funktion f ermitteln:

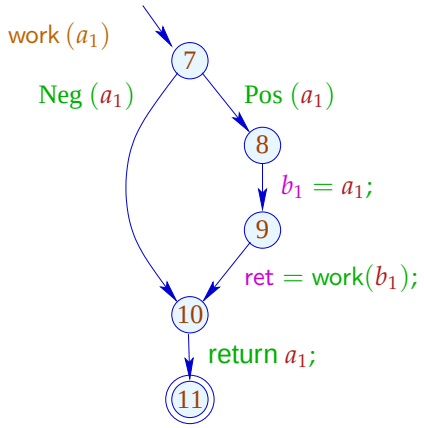
$$\begin{array}{lcl} [\![k]\!]^\sharp & = & H_f\ ([\![f]\!]^\sharp) \qquad \text{wobei:} \\ H_f\ (M) & = & \mathsf{Id} \oplus \{\mathsf{ret} \mapsto ((M \circ E_f)\ \mathsf{ret})\} \\ E_f\ x & = & \begin{cases} b_i & \text{falls} \quad x = a_i \\ 0 & \text{sonst} \end{cases} \end{array}$$

... im Beispiel:

Funktion work: Effekt : Rückgabewert ist der Wert des formalen Parameters a_1

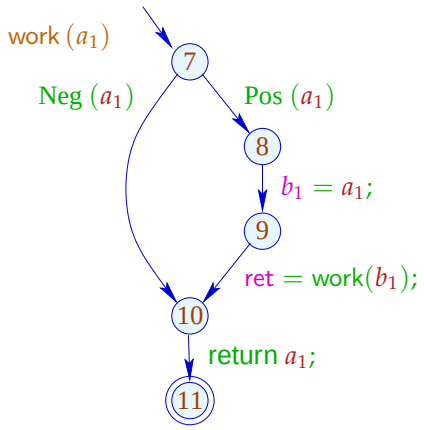
$$
\begin{array}{lrcl}
\text{Falls} & [\![\mathsf{work}]\!]^\sharp & = & \{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto a_1\} \\
\text{dann} & H_{\mathsf{work}}\,[\![\mathsf{work}]\!]^\sharp & = & \mathsf{Id} \oplus \{\mathsf{ret} \mapsto ((\![\mathsf{work}]\!]^\sharp \circ E_f)\ \mathsf{ret})\} \\
 & & = & \mathsf{Id} \oplus \{\mathsf{ret} \mapsto (\boxed{\{a_1 \mapsto b_1, b_1 \mapsto 0, \mathsf{ret} \mapsto b_1\}}\ \mathsf{ret})\} \\
 & & = & \{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto b_1\}
\end{array}
$$

Damit können wir die Fixpunkt-Iteration durchführen :-)



	1
7	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}\}$
10	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}\}$
11	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto a_1\}$
8	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}\}$
9	$\{a_1 \mapsto a_1, b_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}\}$

Nach der nächsten Iteration:



	2
7	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}\}$
10	$\{a_1 \mapsto a_1, b_1 \mapsto a_1 \sqcup b_1, \mathsf{ret} \mapsto a_1 \sqcup \mathsf{ret}\}$
11	$\{a_1 \mapsto a_1, b_1 \mapsto a_1 \sqcup b_1, \mathsf{ret} \mapsto a_1\}$
8	$\{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto \mathsf{ret}\}$
9	$\{a_1 \mapsto a_1, b_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}\}$

Komposition:

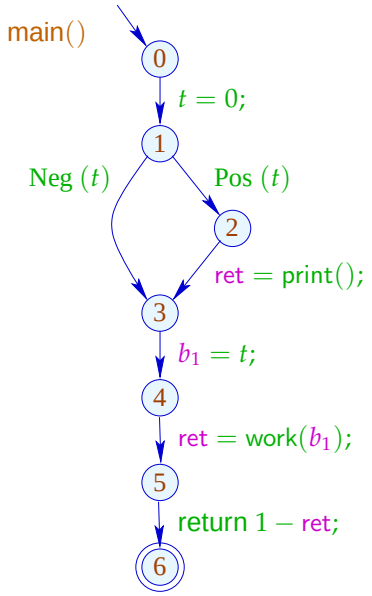
$$\begin{aligned}[\![(9,\ldots,10)]\!]^\sharp \circ [\![9]\!]^\sharp &= \{a_1 \mapsto a_1, b_1 \mapsto b_1, \mathsf{ret} \mapsto b_1\} \circ \\ &\quad \{a_1 \mapsto a_1, b_1 \mapsto a_1, \mathsf{ret} \mapsto \mathsf{ret}\} \\ &= \{a_1 \mapsto a_1, b_1 \mapsto a_1, \mathsf{ret} \mapsto a_1\}\end{aligned}$$

Wenn wir die Effekte von Funktionsaufrufen kennen,
können wir ein Constraintsystem aufstellen,
um den abstrakten Zustand bei Erreichen eines Punkts ermitteln:

$\mathcal{R}$ steht für reach

$$\begin{array}{lcll}
\mathcal{R}[\mathsf{main}] & \sqsupseteq & \mathsf{enter}_0\ d_0 & \\
\mathcal{R}[f] & \sqsupseteq & \mathsf{enter}_f^\sharp\ (\mathcal{R}[u]) & k = (u, \mathsf{ret} = f\,(a_1,\ldots,a_k);, _) \quad \text{Aufruf} \\
\mathcal{R}[v] & \sqsupseteq & \mathcal{R}[f] & v \quad \text{Anfangspunkt von} \quad f \\
\mathcal{R}[v] & \sqsupseteq & [\![k]\!]^\sharp\ (\mathcal{R}[u]) & k = (u, _, v) \quad \text{Kante}
\end{array}$$

... im Beispiel:



0	$\{b_1 \mapsto 0, \mathsf{ret} \mapsto 0, t \mapsto 0\}$
1	$\{b_1 \mapsto 0, \mathsf{ret} \mapsto 0, t \mapsto 0\}$
2	$\{b_1 \mapsto 0, \mathsf{ret} \mapsto 0, t \mapsto 0\}$
3	$\{b_1 \mapsto 0, \mathsf{ret} \mapsto \top, t \mapsto 0\}$
4	$\{b_1 \mapsto 0, \mathsf{ret} \mapsto \top, t \mapsto 0\}$
5	$\{b_1 \mapsto 0, \mathsf{ret} \mapsto 0, t \mapsto 0\}$
6	$\{b_1 \mapsto 0, \mathsf{ret} \mapsto \top, t \mapsto 0\}$

Diskussion:

- Zumindest Kopier-Konstanten lassen sich interprozedural ermitteln.
- Dazu mussten wir Bedingungen und kompliziertere Zuweisungen ignorieren :-(
- In der zweiten Phase hätten wir allerdings exakter rechnen können :-)
- Die weitere Abstrahierung war aus zwei Gründen notwendig:

  (1) Die Menge der auftretenden Transformer $\mathbb{M} \subseteq \mathbb{D} \to \mathbb{D}$ muss endlich sein;

  (2) Die Funktionen $M \in \mathbb{M}$ müssen effizient implementierbar sein :-)
- Auf die zweite Bedingung kann evt. verzichtet werden ...

Beobachtung: Sharir/Pnueli, Cousot

→ Oft werden Funktionen nur mit wenigen verschiedenen abstrakten Argumenten aufgerufen.

→ Man könnte dann doch jede Funktion für nur genau diese Aufrufe analysieren :-)

→ Stelle das folgende Constraint-System auf:

$$
\begin{array}{lcll}
[\![v,a]\!]^\sharp & \supseteq & a & v \text{ Eintrittspunkt} \\
[\![v,a]\!]^\sharp & \supseteq & \mathsf{combine}^\sharp\,([\![u,a]\!], [\![f, \mathsf{enter}^\sharp_f\,[\![u,a]\!]^\sharp]\!]^\sharp) & (u, \mathsf{ret} = f\,(a_1,\ldots,a_k);, v) \text{ Aufruf} \\
 & & & \textit{Geschachtelte Variablen} : \textit{Indirekte Adressierung} \\
 & & & \textit{Dynamische Variablenabhängigkeit} \\
[\![v,a]\!]^\sharp & \supseteq & [\![lab]\!]^\sharp\,[\![u,a]\!]^\sharp & k = (u, lab, v) \text{ Kante} \\
[\![f,a]\!]^\sharp & \supseteq & [\![stop_f, a]\!]^\sharp & stop_f \text{ Endpunkt von } f
\end{array}
$$

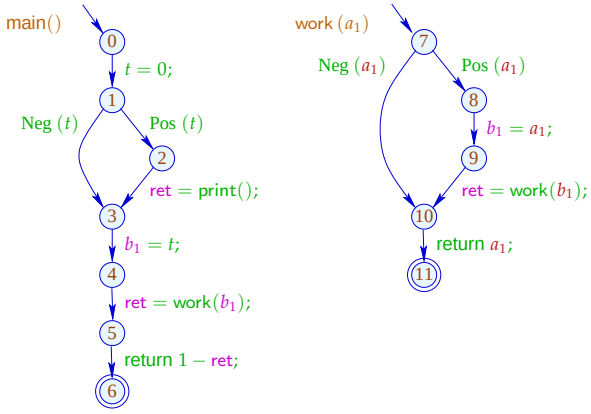
// $[\![v,a]\!]^\sharp$ == Wert des Effekts für das Argument a .

Diskussion:

- Dieses Constraint-System ist i.a. riesengroß :-(
- Wir wollen es aber gar nicht komplett lösen !!!
- Uns reicht es, die korrekten Werte für alle Aufrufe zu ermitteln, die vorkommen, d.h. für die Berechnung des Werts $[\![\mathsf{main}(), a_0]\!]^\sharp$ benötigt werden
  $\Longrightarrow$ wir verwenden unseren lokalen Fixpunkt-Algorithmus :-))
- Der Fixpunkt-Algo liefert uns sogar noch die Menge der aktuellen Parameter $a \in \mathbb{D}$, für die eine Funktion (möglicherweise) aufgerufen wird sowie die Werte an allen ihren Programm-Punkten für jeden dieser Aufrufe :-)

... im Beispiel:

Versuchen wir einfach einmal eine volle Konstanten-Propagation ...



0	$\lambda x.0$	$\lambda x.0$
1	$\lambda x.0$	$\lambda x.0$
2	$\lambda x.0$	$\bot$
3	$\lambda x.0$	$\lambda x.0$
4	$\lambda x.0$	$\lambda x.0$
7	$\lambda x.0$	$\lambda x.0$
8	$\lambda x.0$	$\bot$
9	$\lambda x.0$	$\bot$
10	$\lambda x.0$	$\lambda x.0$
11	$\lambda x.0$	$\lambda x.0$
5	$\lambda x.0$	$\lambda x.0$
6	$\lambda x.0$	$(\lambda x.0) \oplus \{\text{ret} \mapsto 1\}$
main()	$\lambda x.0$	$(\lambda x.0) \oplus \{\text{ret} \mapsto 1\}$

Diskussion:

- Im Beispiel terminiert die Analyse schnell :-)
- Falls $\mathbb{D}$ endliche Höhe hat, terminiert die Analyse, sofern nur jede Funktion während der Iteration nur mit endlich vielen verschiedenen Argumenten aufgerufen wird :-))
- Analoge Analyse-Algorithmen erwiesen sich bei der Analyse von Prolog als äußerst effizient und präzise :-)
- Zusammen mit einer Points-To-Analyse und Propagation selbst von negativer Konstanten-Information haben wir diesen Algorithmus äußerst erfolgreich zur Fehlersuche in C mit Posix-Threads eingesetzt :-)

(2) Der Call-String-Ansatz:

Idee:

→ Berechne die Menge aller erreichbaren Aufrufkeller!

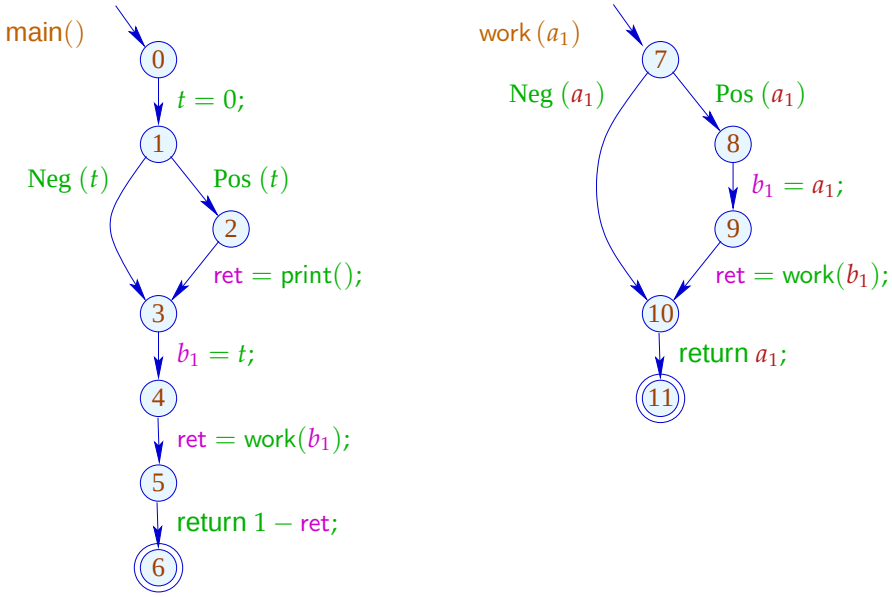
→ Diese ist i.a. unendlich :-(

→ Behandle Keller bis zu einer festen Tiefe d exakt!
Behalte von längeren Kellern nur das obere Ende der Länge d :-)

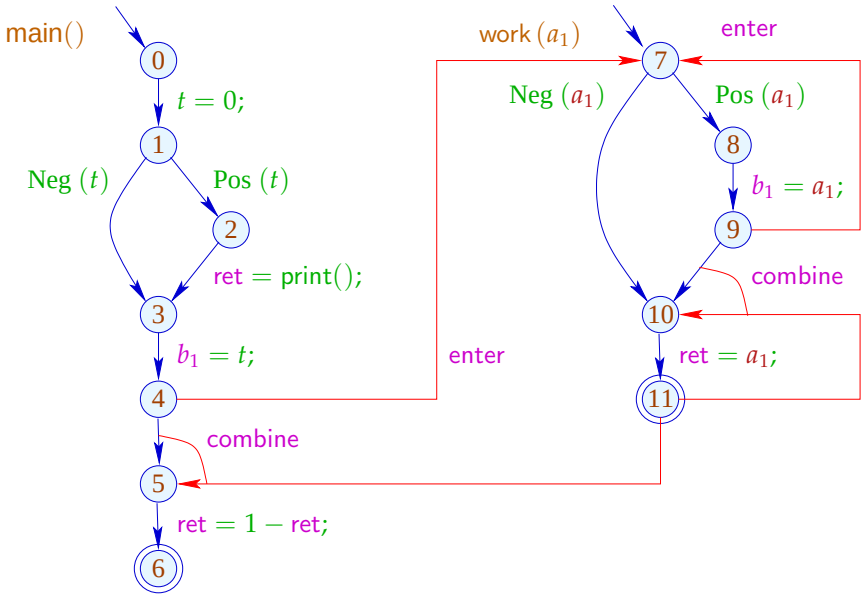
→ Wichtiger Spezialfall: $d = 0$.

$\Longrightarrow$ Betrachte nur die obersten Kellerrahmen ...

... im Beispiel:



Somit ergeben sich Rücksprünge zu allen möglichen Aufrufstellen



Die Bedingungen für $5, 7, 10$ sind dann etwa:

Der Graph könnte als Hyper-, bzw. Supergraph gesehen werden.

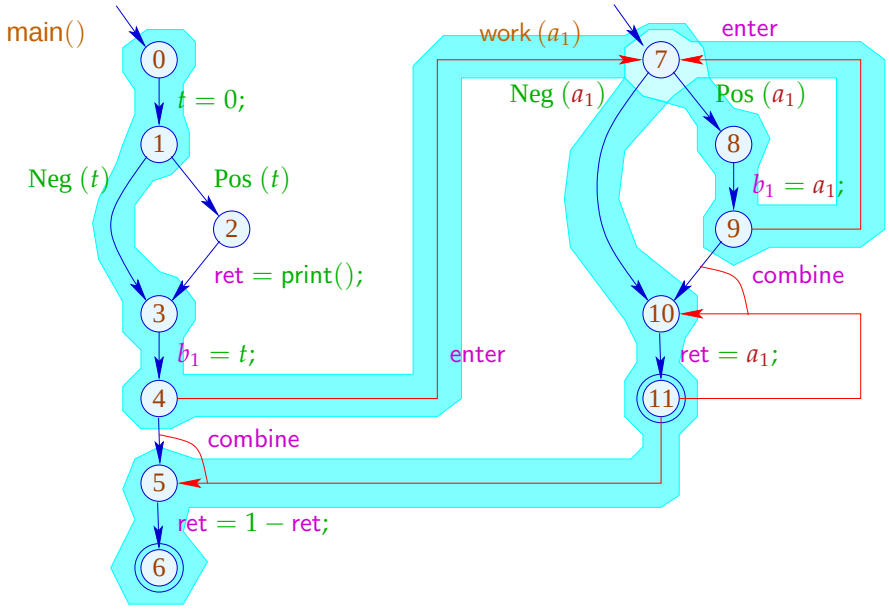
Die Werte von 10 hängen hier von 9 und 11, die von 5 von 4 und 11 ab.

Achtung : combine : hat 2 Argumente hier die Werte von 9 und 11, bzw. 4 und 11.

$$\begin{array}{rcl} \mathcal{R}[5] & \sqsupseteq & \mathsf{combine}^\sharp\,(\mathcal{R}[4], \mathcal{R}[11]) \\ \mathcal{R}[7] & \sqsupseteq & \mathsf{enter}_f^\sharp\,(\mathcal{R}[4]) \\ \mathcal{R}[7] & \sqsupseteq & \mathsf{enter}_f^\sharp\,(\mathcal{R}[9]) \\ \mathcal{R}[10] & \sqsupseteq & \mathsf{combine}^\sharp\,(\mathcal{R}[9], \mathcal{R}[11]) \end{array}$$

Achtung:

Der resultierende Supergraph enthält offensichtlich unmögliche Pfade ...



Beachte:

→ Im Beispiel finden wir zwar die gleichen Ergebnisse:
Mehr Pfade machen die Ergebnisse evt. weniger präzise.

Insbesondere analysieren wir jede Funktion nur für ein (evt. sehr nichtssagendes) Argument-Tupel :-(

→ Die Analyse terminiert — sofern nur $\mathbb{D}$ keine unendlichen echt aufsteigenden Ketten besitzt :-)

→ Die Korrektheit zeigt man relativ zur operationellen Semantik mit den Stacks.

→ Für die Korrektheit des funktionalen Ansatzes ist die Semantik über Berechnungswälder besser geeignet :-)

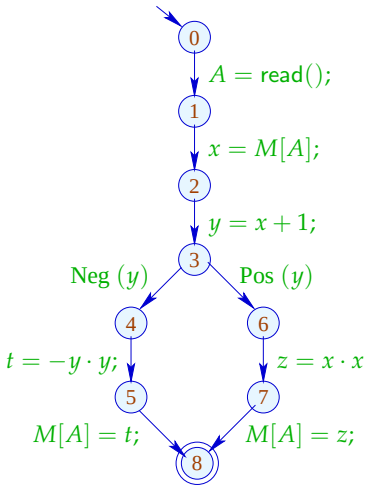
3 Ausnutzung von Hardware-Einrichtungen

Bemerkung : Alle Ansätze für die optimale Ausnutzung von Hardware-Einrichtungen sollten parameterisiert werden, da sich die Hardware ja ständig ändert.
z.B. Anzahl der Register als Parameter angeben..... usw.

3.1 Register

Beispiel:

$A = \mathsf{read}();$
$x = M[A];$
$y = x + 1;$
$\mathsf{if}\ \ (y)\ \ \{$
$\quad z = x \cdot x;$
$\quad M[A] = z;$
$\}\ \ \mathsf{else}\ \ \{$
$\quad t = -y \cdot y;$
$\quad M[A] = t;$
$\}$



Das Programm benötigt 5 Variablen ...

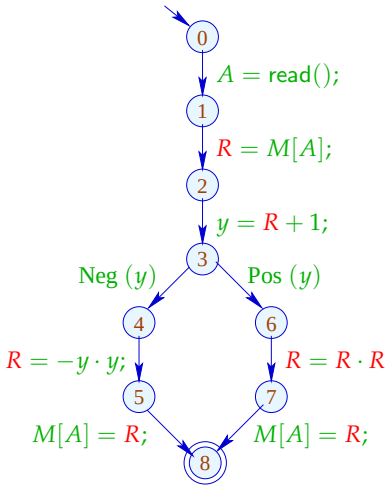
Problem:

Was tun, wenn das Programm mehr Variablen benutzt, als Register vorhanden sind :-(

Idee:

Benutze ein Register für mehrere Variablen :-)
Im Beispiel etwa eines für x, t, z ...

```
A = read();
R = M[A];
y = R + 1;
if (y) {
        R = R · R;
        M[A] = R;
} else {
        R = −y · y;
        M[A] = R;
}
```



Achtung:

Dies funktioniert nur, wenn man weiß, an welchen Programmpunkten eine Variable benötigt wird. Dies kann durch Feststellen des Lebendigkeitsbereichs herausgefunden werden.

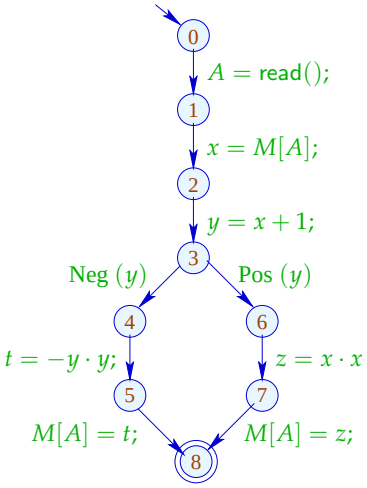
Das geht nur, wenn sich die Lebendigkeitsbereiche nicht überschneiden :-)

Der (wahre) Lebendigkeitsbereich von x ist:

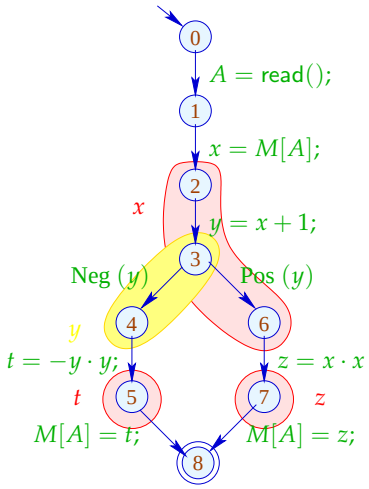
$$\mathcal{L}[x] = \{u \mid x \in \mathcal{L}[u]\}$$

... im Beispiel:

Lebendigkeitsanayse (rückwärts)



	$\mathcal{L}$
8	$\emptyset$
7	$\{A, z\}$
6	$\{A, x\}$
5	$\{A, t\}$
4	$\{A, y\}$
3	$\{A, x, y\}$
2	$\{A, x\}$
1	$\{A\}$
0	$\emptyset$



Lebendigkeitsbereiche:

A	$\{1,\ldots,7\}$
x	$\{2,3,6\}$
y	$\{2,4\}$
t	$\{5\}$
z	$\{7\}$

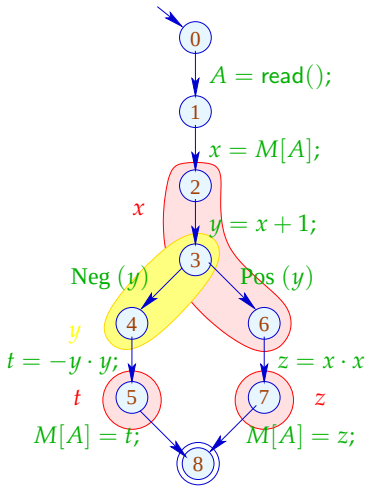
Die Lebendigkeitsbereiche von x,t,z überlagern sich nicht. Deshalb kann hier das gleiche Register benutzt werden.

Um Mengen kompatibler Variablen zu finden, konstruieren wir den Interferenz-Graphen $I = (Vars, E_I)$, wobei:

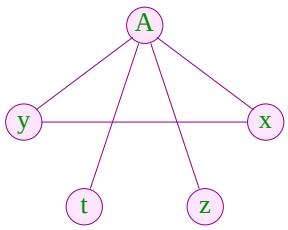
$$E_I = \{\{x, y\} \mid x \neq y, \mathcal{L}[x] \cap \mathcal{L}[y] \neq \emptyset\}$$

E_I enthält eine Kante für $x \neq y$ genau dann wenn x, y an einem gemeinsamen Punkt lebendig sind :-)

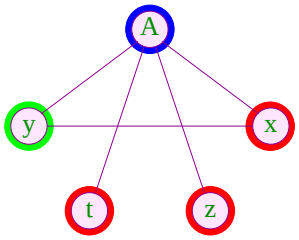
... im Beispiel:



Interferenz-Graph:



Variablen, die nicht mit einer Kante verbunden sind, dürfen dem gleichen Register zugeordnet werden :-)



Farbe $=$ Register

Graphentheoretisches Problem : Färben von Graphen

Sviatoslav Sergeevich Lavrov,
Russische Akademie der Wissenschaften (1962)

Gregory J. Chaitin, University of Maine (1981)

Abstraktes Problem:

Gegeben: Ungerichteter Graph (V, E) .

Gesucht: Minimale Färbung, d.h. Abbildung $c : V \to \mathbb{N}$ mit

(1) $c(u) \neq c(v)$ für $\{u, v\} \in E$;

(2) $\bigsqcup\{c(u) \mid u \in V\}$ minimal!

- Im Beispiel reichen 3 Farben :-) Aber Achtung:
- Die minimale Färbung ist i.a. nicht eindeutig :-(
- Es ist NP-vollständig herauszufinden, ob eine Färbung mit maximal k Farben möglich ist :-((

$\Longrightarrow$

Wir sind auf Heuristiken angewiesen oder Spezialfälle :-)

Greedy-Heuristik:

- Beginne irgendwo mit der Farbe 1;
- Wähle als jeweils neue Farbe die kleinste Farbe, die verschieden ist von allen bereits gefärbten Nachbarn;
- Ist ein Knoten gefärbt, färbe alle noch nicht gefärbten Nachbarn;
- Behandle eine Zusammenhangskomponente nach der andern ...

... etwas konkreter:

```
forall (v ∈ V) c[v] = 0;
forall (v ∈ V) color (v);

void color (v) {
    if (c[v] ≠ 0) return;
    neighbors = {u ∈ V | {u, v} ∈ E};
    c[v] = ⨅{k > 0 | ∀ u ∈ neighbors : k ≠ c(u)};
    forall (u ∈ neighbors)
        if (c(u) == 0) color (u);
}
```

Die neue Farbe lässt sich leicht berechnen, nachdem die Nachbarn nach ihrer Farbe geordnet wurden :-)

Diskussion:

- → Im wesentlichen ist das Prä-order DFS :-)
- → In der Theorie kann das Ergebnis beliebig weit vom Optimum entfernt sein :-(
- → ... ist aber in der Praxis ganz gut :-)
- → ... Achtung: verschiedene Varianten sind patentiert !!!

Der Algorithmus funktioniert umso besser, je kleiner die Lebendigkeitsbereiche sind ...

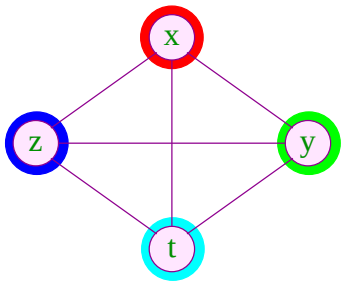
Idee: Life range splitting

Durch Verkleinerung der Anzahl der Kanten (Überschneidung der Lebendigkeitsbereiche) werden auch die Lebendigkeitsbereiche verkleinert und somit wird die Färbung erleichtert.

Beispiel:

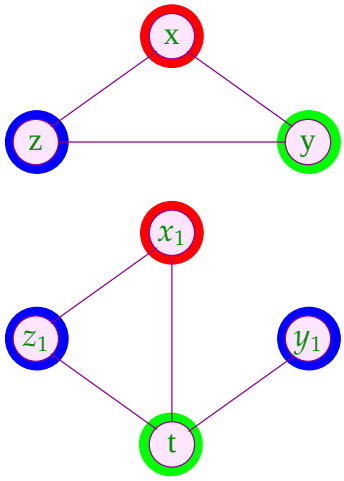
Hier benötigt man 4 Register

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x = x + 1;$	x
$z = M[A_1];$	x, z
$t = M[x];$	x, z, t
$A_2 = x + t;$	x, z, t
$M[A_2] = z;$	x, t
$y = M[x];$	y, t
$M[y] = t;$	



Die Lebendigkeitsbereiche von x und z können wir aufteilen:

	$\mathcal{L}$
	x, y, z
$A_1 = x + y;$	x, z
$M[A_1] = z;$	x
$x_1 = x + 1;$	x_1
$z_1 = M[A_1];$	x_1, z_1
$t = M[x_1];$	x_1, z_1, t
$A_2 = x_1 + t;$	x_1, z_1, t
$M[A_2] = z_1;$	x_1, t
$y_1 = M[x_1];$	y_1, t
$M[y_1] = t;$	



Hier benötigt man 3 Register, hat aber mehr Variablen!

Technisch:

Um die Lebendigkeitsbereiche zu verkleinern, wird der Endpunkt eines Lebendigkeitsbereiches bei Definition gesetzt

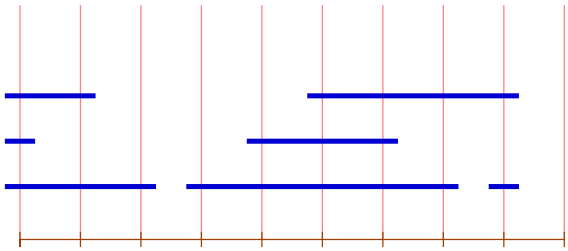
Eine Kante (u, lab, v) heißt x-transparent, falls lab keine Definition von x ist.

u, v gehören zum selben minimalen x-Lebendigkeitsbereich , falls $x \in \mathcal{L}[u] \cap \mathcal{L}[v]$ und u, v durch einen ungerichteten Pfad x-transparenter Kanten verbunden sind ...

Für jeden der minimalen x-Lebendigkeitsbereiche $L_1, \ldots, L_k$ für wir eine Variante von x ein :-)

Spezialfall: Basis-Blöcke

Die Interferenzgraphen für minimale Lebendigkeitsbereiche auf Folgen von Zuweisungen sind Intervall-Graphen:



Knoten $=$ Intervall
Kante $=$ gemeinsamer Punkt

Zu jedem Punkt können wir die Überdeckungszahl der inzidenten Intervalle angeben.

Satz:

maximale Überdeckungszahl
$=$ Größe der maximalen Klique
$=$ maximal nötige Anzahl Farben :-)

Graphen mit dieser Eigenschaft heißen perfekt ...
Eine minimale Färbung kann in polynomieller Zeit berechnet werden :-))

Idee:

- → Iteriere (konzeptuell) über die Punkte $0, \ldots, m-1$!
- → Verwalte eine Liste der aktuell freien Farben.
- → Beginnt ein neues Intervall, vergib die nächste freie Farbe.
- → Endet ein Intervall, gib seine Farbe frei.

Damit ergibt sich folgender Algorithmus:

```
free = [1,...,k];
for (i = 0; i < m; i++) {
    init[i] = []; exit[i] = [];
}
forall (I = [u,v] ∈ Intervals) {
    init[u] = (I :: init[u]); exit[i] = (I :: exit[v+1]);
}
for (i = 0; i < m; i++) {
    forall (I ∈ exit[i]) free = color[I] :: free;
    forall (I ∈ init[i]) {
        color[I] = hd free; free = tl free;
    }
}
```

Diskussion:

- → Für Basis-Blöcke können wir eine optimale Aufteilung der Variablen auf eine Register ermitteln :-)
- → Das gleiche Problem ist bereits für einfache Schleifen (circular arc graphs) NP-schwierig :-(
- → Für beliebige Programme wird man deshalb eine Heuristik zum Graph-Färben einsetzen ...
- → Dieses Verfahren funktioniert besser, wenn wir die Lebendigkeitsbereiche maximal unterteilen :-)
- → Reicht die Anzahl der realen Register nicht aus, lagert man die überzähligen in einen festen Speicherbereich aus.
- → Man bemüht sich dabei, zumindest die in innersten Schleifen benutzten Variablen in Registern zu halten.

Hier wäre wieder eine Unterscheidung der Variablen nach der sog. Temperatur (s. frühere Lektion) optimal. Damit kann man Variable in Schleifenrümpfen erkennen und diese in Registern halten.

Interprozedurale Registerverteilung:

→ Für jede lokale Variable ist ein Eintrag im Kellerrahmen reserviert.

→ Vor dem Aufruf einer Funktion müssen die Register in den Kellerrahmen gerettet und danach restauriert werden. *Bei Prozessoren mit einer hohen Anzahl von Registern ist dies u.U. sehr aufwendig.*

→ Gelegentlich gibt es dafür Hardware-Unterstützung :-) *SPARC* Dann ist ein Aufruf für alle Register transparent.

→ Verwalten wir Retten / Restaurieren selbst, können wir ...

- nur Register retten, deren Inhalte nach dem Aufruf noch benötigt werden :-)
- Register erst bei Bedarf restaurieren — und dann evt. in andere Register $\Longrightarrow$ Verkleinerung der Lebendigkeitsbereiche :-)

3.2 Instruktionen

Es gibt Prozessoren mit sehr grossen Instruktionssätzen, die i.A. sehr unstrukturiert aufgebaut sind.

Problem:

- unregelmäßige Instruktionssätze ...
- mehrere Adressierungsarten, die evt. mit arithmetischen Operationen kombiniert werden können;
  Auch verschiedene Datenbreiten
- Register für unterschiedliche Verwendungen ...
  Spezialregister: In- /Dekrementregister, Adressregister, Datenregister usw.

Beispiel: Motorola MC68000

Dieser einfachste Prozessor der 680x0-Reihe besitzt

- 8 Daten- und 8 Adressregister;
- eine Vielzahl von Adressierungsarten ...

Notation	Beschreibung	Semantik
D_n	Datenregister direkt	D_n
A_n	Adressregister direkt	A_n
(A_n)	Adressregister indirekt	$M[A_n]$
$d(A_n)$	Adressregister indirekt mit Displacement	$M[A_n + d]$
$d(A_n, D_m)$	Adressregister indirekt mit Index und Displacement	$M[A_n + D_m + d]$
x	Absolut kurz	$M[x]$
x	Absolut lang	$M[x]$
$\#x$	Unmittelbar	x

- Der MC68000 ist eine 2-Adress-Maschine, d.h. ein Befehl darf maximal 2 Adressierungen enthalten. Die Instruktion:

  $$\texttt{add}\ \ D_1\ \ D_2$$

  addiert die Inhalte von D_1 und D_2 und speichert das Ergebnis nach und D_2 :-)

- Die meisten Befehle lassen sich auf Bytes, Wörter (2 Bytes) oder Doppelwörter (4 Bytes) anwenden.

  Das unterscheiden wir durch Anhängen von .B, .W, .D (Default: .W)
- Die Ausführungszeit eines Befehls ergibt sich (i.a.) aus den Kosten der Operation plus den Kosten für die Adressierung der Operanden ...

Die Ausführungszeit eines Befehls ist bei diesem Prozessor (Motorola MC68000) genau angegeben. Das ist bei moderneren Prozessoren oft nicht der Fall.
Alternativ bietet sich statt der Addition der Ausführungszeiten der einzelnen Befehlen in einer Befehlsfolge als Kosten, einfach die Anzahl der Befehle in einer Befehlsfolge als Kriterium an.

	Adressierungsart	Byte / Wort	Doppelwort
D_n	Datenregister direkt	0	0
A_n	Adressregister direkt	0	0
(A_n)	Adressregister indirekt	4	8
$d(A_n)$	Adressregister indirekt mit Displacement	8	12
$d(A_n, D_m)$	Adressregister indirekt mit Index und Displacement	10	14
x	Absolut kurz	8	12
x	Absolut lang	12	16
$\#x$	Unmittelbar	4	8

Beispiel:

Die Instruktion: move.B $8(A_1, D_1.W), D_5$
benötigt: 4 + 10 + 0 = 14 Zyklen

Alternativ könnten wir erzeugen:

adda	$\#8, A_1$	Kosten: $8+8+0=16$
adda	$D_1.W, A_1$	Kosten: $8+0+0=8$
move.B	$(A_1), D_5$	Kosten: $4+4+0=8$

mit Gesamtkosten 32 oder:

adda	$D_1.W, A_1$	Kosten: $8+0+0=8$
move.B	$8(A_1), D_5$	Kosten: $4+8+0=12$

mit Gesamtkosten 20 :-)

Achtung:

- Die verschieden Code-Sequenzen sind im Hinblick auf den Speicher und das Ergebnis äquivalent !
- Sie unterscheiden sich im Hinblick auf den Wert des Registers A_1 sowie die gesetzten Bedingungs-Codes !!
- Ein schlauer Instruktions-Selektor muss solche Randbedingungen berücksichtigen :-)

Etwas größeres Beispiel:

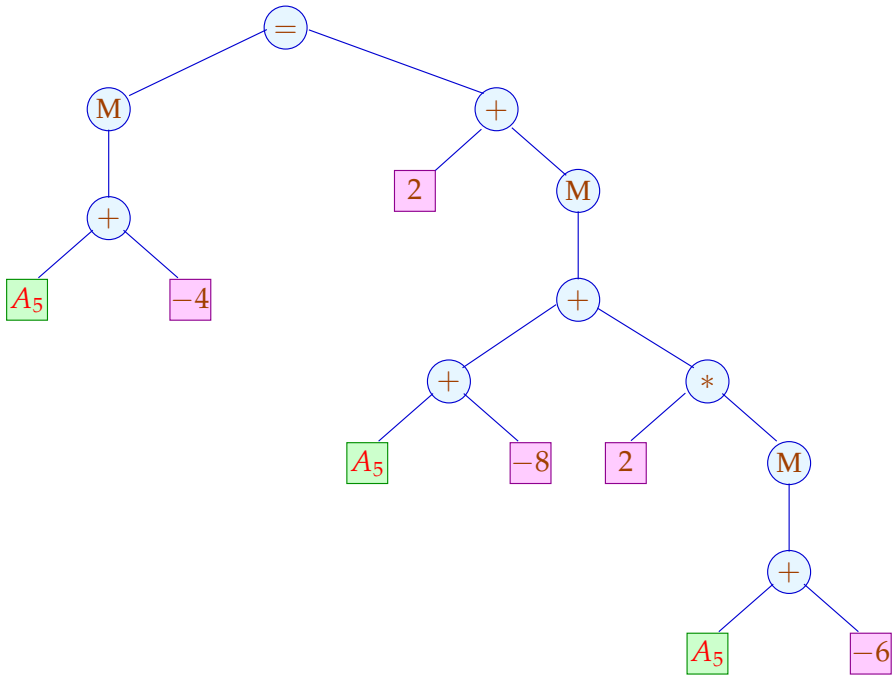
$$\begin{aligned}&\text{int } b,\ i,\ a[100];\\ &b = 2 + a[i];\end{aligned}$$

Nehmen wir an, die Variablen werden relativ zu einem Framepointer A_5 mit den Adressen $-4, -6, -8$ adressiert. Dann entspricht der Zuweisung das Stück Zwischen-Code:

$$M[A_5 - 4] = 2 + M[A_5 - 8 + 2 \cdot M[A_5 - 6]];$$

Multiplikation mit 2, wegen Byte-Adressierung

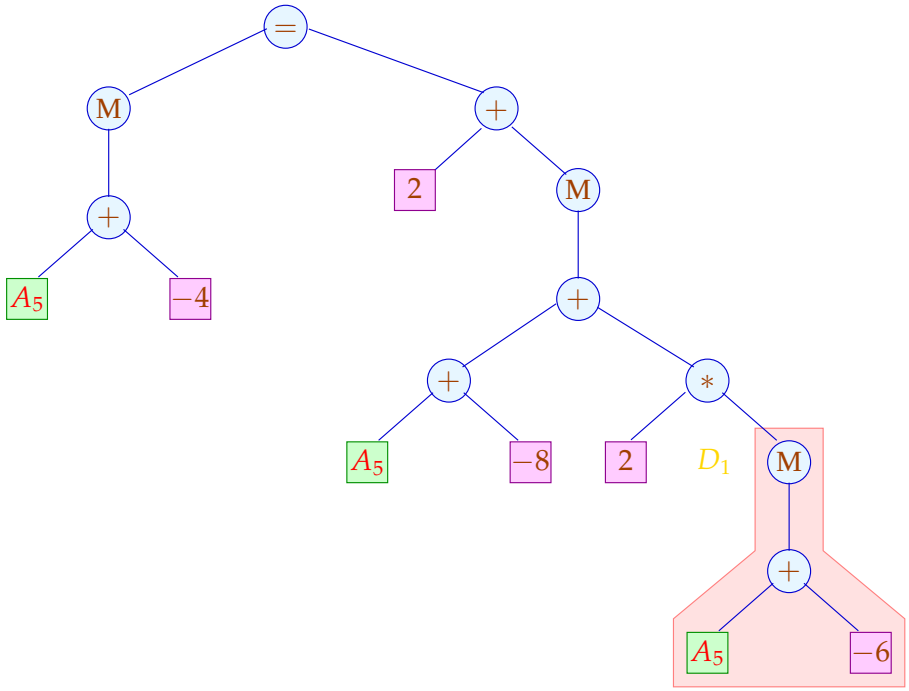
Das entspricht dem Syntaxbaum:



Eine mögliche Code-Sequenz:

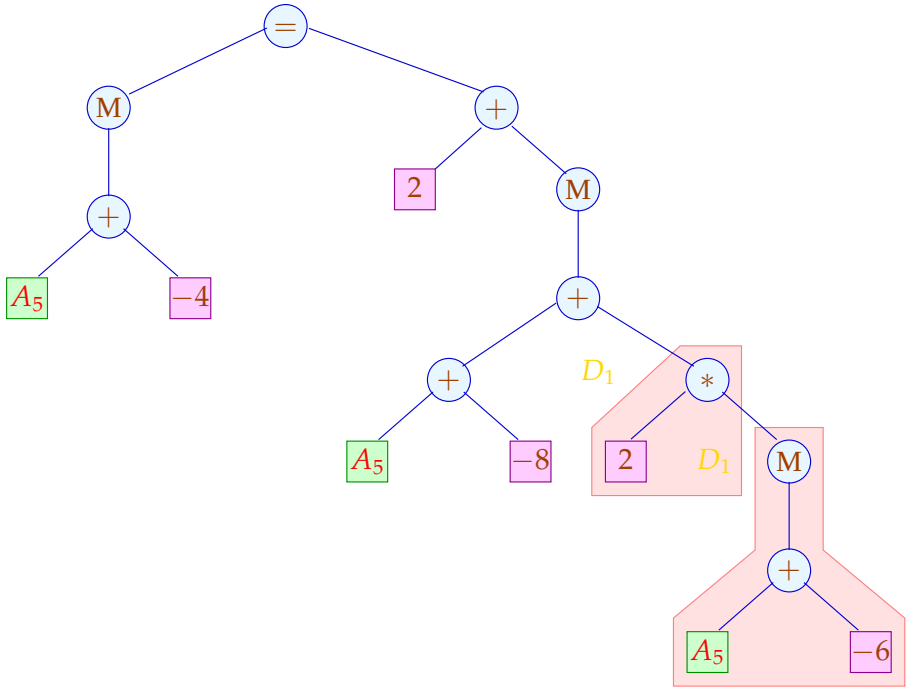
move	$-6(A_5),\ D_1$	Kosten:	12
add	$D_1,\ D_1$	Kosten:	4
move	$-8(A_5, D_1),\ D_2$	Kosten:	14
addq	$\#2,\ D_2$	Kosten:	4
move	$D_2,\ -4(A_5)$	Kosten:	12
	Gesamtkosten :		46

Graphische Repräsention
move $-6(A_5),\ D_1$



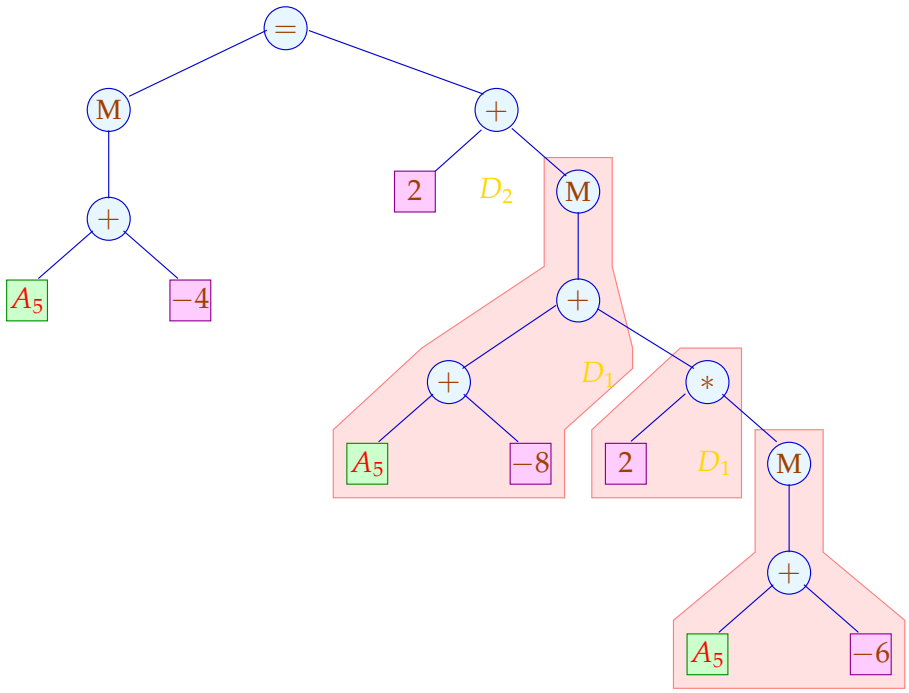
Graphische Repräsentation

add D_1, D_1



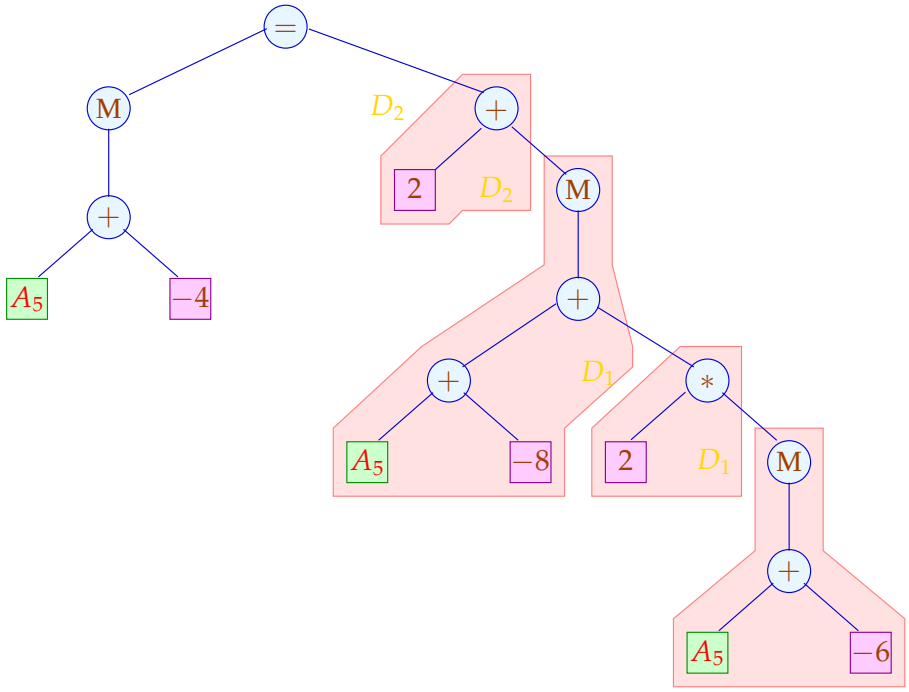
Graphische Repräsentation

move $-8(A_5, D_1)$, D_2



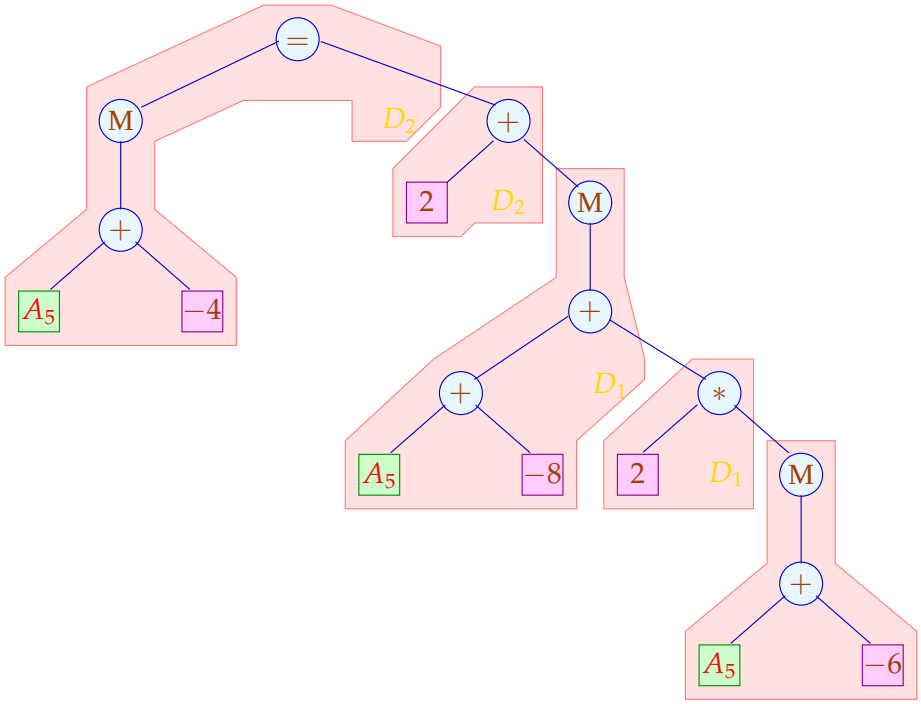
Graphische Repräsentation

addq #2, D_2



Graphische Repräsentation

move D_2, $-4(A_5)$



Eine alternative Code-Sequenz:

move.L	$A_5,\ A_1$	Kosten:	4
adda.L	$\#-6,\ A_1$	Kosten:	12
move	$(A_1),\ D_1$	Kosten:	8
mulu	$\#2,\ D_1$	Kosten:	44
move.L	$A_5,\ A_2$	Kosten:	4
adda.L	$\#-8,\ A_2$	Kosten:	12
adda.L	$D_1,\ A_2$	Kosten:	8
move	$(A_2),\ D_2$	Kosten:	8
addq	$\#2,\ D_2$	Kosten:	4
move.L	$A_5,\ A_3$	Kosten:	4
adda.L	$\#-4,\ A_3$	Kosten:	12
move	$D_2,\ (A_3)$	Kosten:	8
	Gesamtkosten :		124

Diskussion:

→ Die Folge ohne komplexe Adressierungsarten ist erheblich teurer :-(

→ Sie benötigt auch mehr Hilfsregister :-(

→ Die beiden Folgen sind nur äquivalent im Hinblick auf den Speicher — die Register haben anschließend verschiedene Inhalte ...

→ Eine korrekte Folge von Instruktionen kann als eine Pflasterung des Syntaxbaums aufgefasst werden !!!

Genereller Ansatz:

Suche nach der geeigneten Instruktionsselektion

- Wir betrachten Basis-Blöcke vor der Registerverteilung:

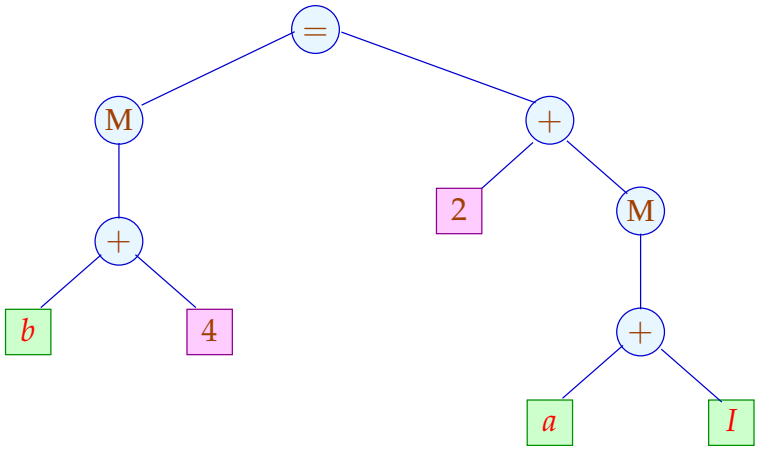
$$
\begin{aligned}
A &= a + I;\\
D_1 &= M[A];\\
D_2 &= D_1 + 2;\\
B &= b + 4;\\
M[B] &= D_2
\end{aligned}
$$

Versuch aus diesen einfachen Zuweisungen komplexere Instruktionen zu erzeugen.

- Wir fassen diese als Folge von Bäumen auf. Wurzeln:

- Werte, die mehrmals verwendet werden;
- Variablen, die am Ende des Blocks lebendig sind;
- Stores.

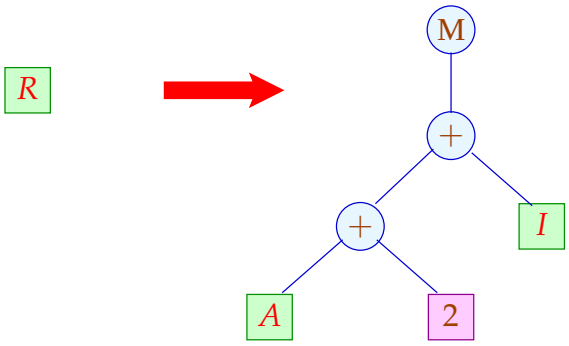
... im Beispiel:



Die Hilfsvariablen A, B, D_1, D_2 sind vorerst verschwunden :-)

Idee:

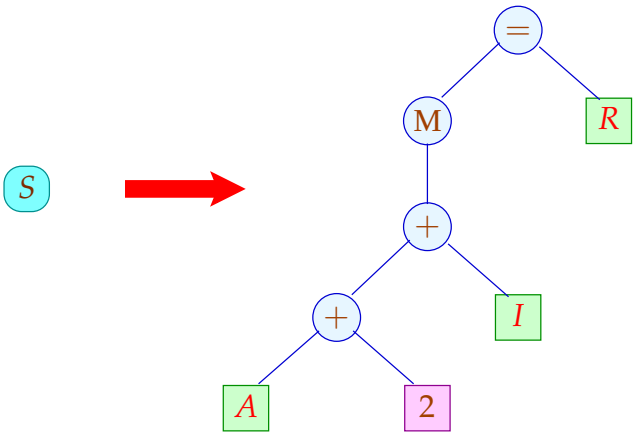
Beschreibe den Effekt einer Instruktion als Ersetzungsregel auf Bäumen:
Die Instruktion: $R = M[A + 2 + D];$
entspricht zum Beispiel:



linke Seite	Ergebnisregister(klasse)
rechte Seite	berechneter Wert für Ergebnisregister
innere Knoten	• Load M • Arithmetik
Blätter	• Argumentregister(klassen) • Konstanten(klasse)

Die Grundidee erweitern wir (evt.) um eine Store-Operation.

Für die Instruktion: $M[A+2+D] = R;$
erlauben wir uns:



Die linke Seite S kommt nicht in rechten Seiten vor :-)

Spezifikation des Instruktionssatzes:

(1)	verfügbare Registerklassen	//	Nichtterminale
(2)	Operatoren und Konstantenklassen	//	Terminale
(3)	Instruktionen	//	Regeln

Auch S ist ein Nichtterminal. Register können auch Terminale sein.

$\Longrightarrow$ reguläre Baumgrammatik

Triviales Beispiel:

Mit einem kleinen Instruktionssatz

Loads :	Comps :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A+A]$	$D \rightarrow D+D$	$A \rightarrow D$

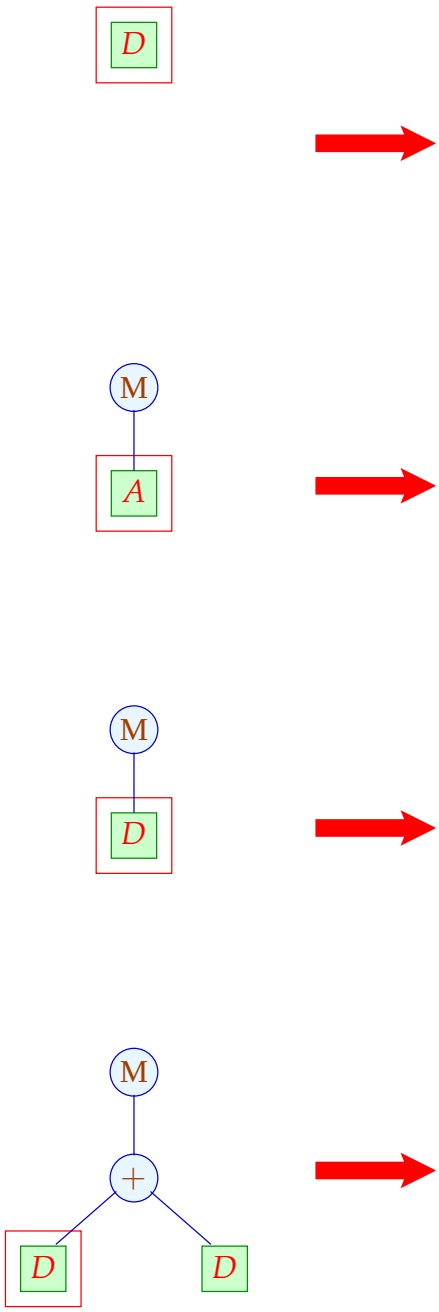
- Registerklassen D (Data) und A (Address).
- Arithmetik wird nur für Daten unterstützt ...
- Laden nur für Adressen :-)

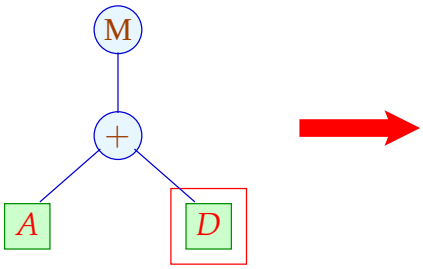
- Zwischen Daten- und Adressregistern gibt es Moves.

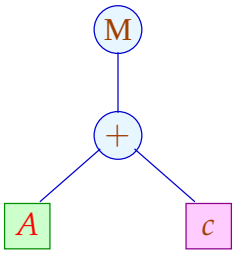
Target: $M[A + c]$

Aufgabe:

Finde Folge von Regelanwendungen, die das Target aus einem Nichtterminal erzeugt ...







Die umgekehrte Folge der Regelanwendungen liefert eine geeignete Instruktionsfolge :-)

Verschiedene Ableitungen liefern verschiedene Folgen ...

Problem:

→ Wie durchsuchen wir systematisch die Menge aller Ableitungen ?

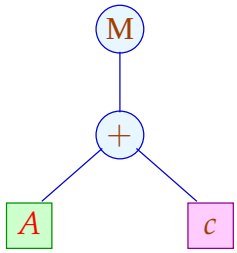
→ Wie finden wir die beste ??

Beobachtung:

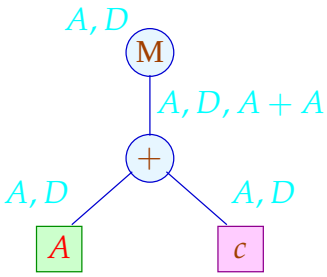
- Nichtterminale stehen stets an den Blättern.
- Statt eine Ableitung für das Target topdown zu raten, sammeln wir sämtliche Möglichkeiten bottom-up auf

  $\Longrightarrow$ Tree parsing
- Dazu lesen wir die Regeln von rechts nach links ...

Zu jedem Knoten wird notiert, aus welchen Nonterminalen er ableitbar ist.



A ist aus A und D ableitbar.
c ist aus A und D ableitbar.
+ (Additionsoperator) : A, D und auch A+A falls man den Ausdruck nicht auswertet.



Für jeden Teilbaum t des Targets sammeln wir die Menge

$$Q(t) \subseteq \{S\} \cup \mathsf{Reg} \cup \mathsf{Term}$$

Reg die Menge der Registerklassen,
Term die Menge der Teilbäume rechter Seiten — auf mit:

$$Q(t) = \{s \mid s \Rightarrow^* t\}$$

Diese ergeben sich zu:

$$\begin{array}{lcl} Q(R) & = & \mathsf{Move}\,\{R\} \\ Q(c) & = & \mathsf{Move}\,\{c\} \\ Q(a(t_1,\ldots,t_k)) & = & \mathsf{Move}\,\{s = a(s_1,\ldots,s_k) \in \mathsf{Term} \mid s_i \in Q(t_i)\} \\ & & \quad // \quad \text{normalerweise} \quad k \leq 2 \quad \text{:-)} \end{array}$$

Die Hilfsfunktion Move bildet den Abschluss unter Regelanwendungen:
Ungleichungssystem weil Rekursion!

$$\begin{array}{lcl} \mathsf{Move}\,(L) & \supseteq & L \\ \mathsf{Move}\,(L) & \supseteq & \{R \in \mathsf{Reg} \mid \exists\, s \in L : \quad R \rightarrow s\} \end{array}$$

Die kleinste Lösung dieses Constraint-Systems lässt sich aus der Grammatik in linearer Zeit berechnen :-)

// Im Beispiel haben wir in $Q(t)$ auf s verzichtet,
// falls s kein echter Teilterm einer rechten Seite ist :-)

Auswahlkriterien:

- Länge des Codes;
- Laufzeit der Ausführung; *Falls die Takte der Befehle bekannt sind*
- Parallelisierbarkeit;
- ...

Achtung:

Die Laufzeit von Instruktionen kann vom Kontext abhängen !!?

Vereinfachung:

Jede Instruktion r habe Kosten $c[r]$.
Die Kosten einer Instruktionsfolge sind additiv: *Eben beim MC 68000*

$$c[r_1 \ldots r_k] \;=\; c[r_1] + \ldots + c[r_k]$$

	c	Instruktion
0	3	$D \rightarrow M[A + A]$
1	2	$D \rightarrow M[A]$
2	1	$D \rightarrow D + D$
3	1	$D \rightarrow c$
4	1	$D \rightarrow A$
5	1	$A \rightarrow D$

Aufgabe:

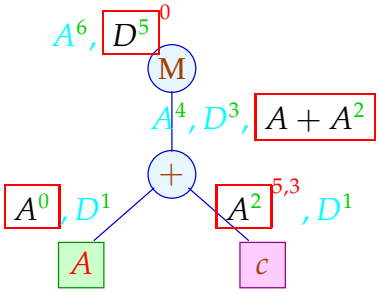
Wähle eine Instruktionsfolge mit minimalen Kosten !

Idee:

Sammle Ableitungen bottom-up auf unter

* Kostenkalkulation und
* Auswahl.

... im Beispiel:



Kostenkalkulation:

$$\begin{array}{lcll} c_t[s] & = & c_{t_1}[s_1] + \ldots + c_{t_k}[s_k] & \text{falls} \quad s = a(s_1,\ldots,s_k), t = a(t_1,\ldots,t_k) \\ c_t[R] & = & \bigsqcap\{c[R,s] + c_t[s] \mid s \in Q(t)\} & \text{wobei} \end{array}$$

$$\begin{array}{lcll} c[R,s] & \leq & c[r] & \text{falls} \quad r : R \to s \\ c[R,s] & \leq & c[r] + c[R',s] & \text{falls} \quad r : R \to R' \end{array}$$

Das Constraint-System für $c[R,s]$ kann in Zeit $\mathcal{O}(n \cdot \log\, n)$ gelöst werden — falls n die Anzahl der Paare R,s ist :-)

Für jedes R,s liefert die Fixpunkt-Berechnung eine Folge:

$$\pi[R,s] \;:\; R \Rightarrow R_1 \Rightarrow \ldots \Rightarrow R_k \Rightarrow s$$

deren Kosten gerade $c[R,s]$ ist :-)
Mithilfe der $\pi[R,s]$ lässt sich eine billigste Ableitung topdown rekonstruieren :-)

Im Beispiel:

$$\begin{array}{lcl} D_2 & = & c; \\ A_2 & = & D_2; \\ D_1 & = & M[A_1 + A_2]; \end{array}$$

mit Kosten 5 . Die Alternative:

$$\begin{array}{lcl} D_2 & = & c; \\ D_3 & = & A_1; \\ D_4 & = & D_3 + D_2; \\ A_2 & = & D_4; \\ D_1 & = & M[A_2]; \end{array}$$

hätte Kosten 7 :-)

Diskussion:

- Die Code-Erzeugung muss schnell gehn :-)
- Anstelle für jeden Knoten neu zu überprüfen, wie die Regeln zusammen passen, kann die Berechnung auch in einen endlichen Automaten kompiliert werden :-))

Endliche deterministische Automaten finden auch speziell im Compilerbau Anwendung. (z.B. Scanner)

Ein deterministischer endlicher Baumautomat (DTA) A besteht aus:

$$\begin{array}{lll} Q & = & \text{endliche Menge von Zuständen} \\ \Sigma & = & \text{Operatoren und Konstanten} \\ \delta_a & = & \text{Übergangsfunktion für } a \in \Sigma \\ F \subseteq Q & = & \text{akzeptierende Zustände} \end{array}$$

Akzeptierende Zustände (Theorie) : In der Praxis werden die Zustände interpretiert.

Dabei ist:

$$\begin{array}{lll} \delta_c : & Q & \text{falls } c \text{ Konstante} \\ \delta_k : & Q^k \to Q & \text{falls } a \text{ } k\text{-stellig} \end{array}$$

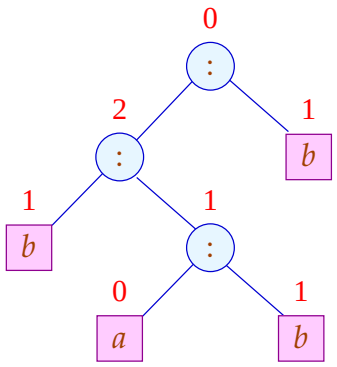
DFS-Traversierung in Postorderdurchlauf. (Zuerst die Söhne, dann die Knoten)
Stelligkeit bei uns: M : einstellig, + : zweistellig

Beispiel:

Zustandsmenge : 0,1,2 mit Endzustand 0;
Der Automat zählt die Anzahl der Blätter mit Beschriftung b modulo 3

$$\begin{array}{lllll} Q & = & \{0,1,2\} & F & = & \{0\} \\ \Sigma & = & \{a,b,:\} & & & \\ \delta_a & = & 0 & \delta_b & = & 1 \\ & \delta_: (s_1,s_2) & = & (s_1+s_2)\%3 & & \end{array}$$

// akzeptiert alle Bäume mit $3 \cdot k$ b-Blättern



Der Zustand an einem Knoten a ergibt sich
aus den Zuständen der Kinder mittels δ_a (-:

$$\begin{aligned} Q(c) &= \delta_c \\ Q(a(t_1,\ldots,t_k)) &= \delta_a(Q(t_1),\ldots,Q(t_k)) \end{aligned}$$

Die von A definierte Sprache (oder: Menge von Bäumen) ist:

$$\mathcal{L}(A) = \{t \mid Q(t) \in F\}$$

... in unserer Anwendung:

$$\begin{array}{lcl} Q & == & \text{Teilmengen von } \mathsf{Reg} \cup \mathsf{Term} \cup \{S\} \\ \quad // & & \text{I.a. werden nicht sämtliche Teilmengen benötigt :-)} \\ F & == & \text{gewünschter Effekt} \\ \delta_R & == & \mathsf{Move}\,\{R\} \\ \delta_c & == & \mathsf{Move}\,\{c\} \\ \delta_a(Q_1,\ldots,Q_k) & == & \mathsf{Move}\,\{s = a(s_1,\ldots,s_k) \in \mathsf{Term} \mid s_i \in Q_i\} \end{array}$$

Die Zustände des endlichen Automaten sind Teilmengen von Mustern und Registern.
Diesen wird mithilfe von δ eine neue Teilmenge zugeordnet.

... im Beispiel:

$$\begin{aligned} \delta_c &= \{A, D\} = q_0 \\ &= \delta_A \\ &= \delta_D \\ \delta_+(q_0,q_0) &= \{A, D, A+A\} = q_1 \\ &= \delta_+(q_0,\_) \\ &= \delta_+(\_,q_0) \end{aligned}$$

$$\begin{aligned} \delta_M(q_0) &= \{A, D\} = q_0 \\ &= \delta_M(q_1) \end{aligned}$$

Um die Anzahl der Zustände zu reduzieren, haben wir die vollständigen rechten Seiten, die keine echten Teilmuster sind, in den Zuständen weggelassen :-)

Integration der Kostenberechnung:

Man möchte nun die billigste Codererzeugung herausfinden, somit müssen für jeden Teilbaum die Kosten mitprotokolliert werden. Diese Kosten können bei grossen Bäumen im Prinzip beliebig gross werden.

Problem:

Kosten können (im Prinzip) beliebig groß werden ;-(
Unser FTA besitzt aber nur endlich viele Zustände :-((

Idee:

Pelegri-Lopart 1988

Betrachte nicht absolute Kosten — sondern relative !!!

Eduardo Pelegri-Llopart,
Sun Microsystems, Inc.

Beobachtung:

- In gängigen Prozessoren kann man Werte von jedem Register in jedes andere schieben $\Longrightarrow$

  Die Kosten zwischen Registern differieren nur um eine Konstante :-) *Eben die Kosten eines MOVE-Befehls.*
- Komplexe rechte Seiten lassen sich i.a. mittels elementarerer Instruktionen simulieren $\Longrightarrow$

  Die Kosten zwischen Teilausdrücken und Registern differieren nur um eine Konstante :-))
- Die Kostenberechnung ist additiv $\Longrightarrow$

Wir können statt mit absoluten Kosten-Angaben auch mit Kosten-Differenzen rechnen !!!

Von diesen gibt es nur endlich viele :-)

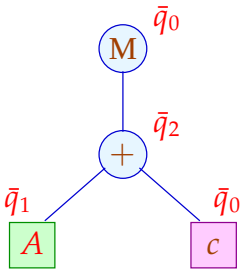
... im Beispiel:

$$
\begin{array}{rcl}
\delta_c & = & \{A \mapsto 1, D \mapsto 0\} \quad = \quad \bar{q}_0 \\
 & = & \delta_D \\
\delta_A & = & \{A \mapsto 0, D \mapsto 1\} \quad = \quad \bar{q}_1
\end{array}
$$

$$
\begin{array}{rcl}
\delta_+(\bar{q}_1, \bar{q}_0) & = & \{A \mapsto 2, D \mapsto 1, A+A \mapsto 0\} \quad = \quad \bar{q}_2 \\
\delta_+(\bar{q}_0, \bar{q}_0) & = & \{A \mapsto 1, D \mapsto 0, A+A \mapsto 1\} \quad = \quad \bar{q}_3 \\
\delta_+(\bar{q}_1, \bar{q}_1) & = & \{A \mapsto 4, D \mapsto 3, A+A \mapsto 0\} \quad = \quad \bar{q}_4 \\
 & \ldots & \\
\delta_M(\bar{q}_2) & = & \{A \mapsto 1, D \mapsto 0\} \quad = \quad \bar{q}_0 \\
 & = & \delta_M(\bar{q}_i) \qquad\qquad , \quad i = 0, \ldots, 4
\end{array}
$$

Durch das Anwenden von δ_M auf $\bar{q}_2$ bekommt man wieder $\bar{q}_0$:
Man sieht, dass viele Zustände zusammenfallen.

... das liefert die folgende Berechnung:



Für jede Konstanten-Klasse c und jedes Register R in δ_c
tabellieren wir die zu wählende billigste Berechnung:

$$c \; : \quad \{A \mapsto 5, 3, D \mapsto 3\}$$

Analog tabellieren wir für jeden Operator a ,
jedes $\tau \in \bar{Q}^k$ und jedes R in $\delta_a(\tau)$:

M	select_M
$\bar{q}_0$	$\{A \mapsto 5,1, D \mapsto 1\}$
$\bar{q}_1$	$\{A \mapsto 5,1, D \mapsto 1\}$
$\bar{q}_2$	$\{A \mapsto 5,0, D \mapsto 0\}$
$\bar{q}_3$	$\{A \mapsto 5,1, D \mapsto 1\}$
$\bar{q}_4$	$\{A \mapsto 5,0, D \mapsto 0\}$

Für "+" ist die Tabelle besonders einfach:

$+$	$\bar{q}_j$
$\bar{q}_i$	$\{A \mapsto 5,3, D \mapsto 3\}$

Problem:

- Für reale Instruktionssätze benötigt man leicht um die 1000 Zustände.
- Die Tabellen für mehrstellige Operatoren werden riesig :-(

$\Longrightarrow$ Wir benötigen Verfahren der Tabellen-Komprimierung ...

Tabellen-Kompression:

Die Tabelle für "+" sieht im Beispiel so aus:

$+$	$\bar{q}_0$	$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_4$
$\bar{q}_0$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$
$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_4$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$
$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$
$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$
$\bar{q}_4$	$\bar{q}_3$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_3$	$\bar{q}_3$

Die meisten Zeilen / Spalten sind offenbar ganz ähnlich ;-)

Idee 1: Äquivalenzklassen

Zwei Zustände sind modulo eines Operators a äquivalent, wenn die Zeile und Spalte gleich sind.

Wir setzen $q \equiv_a q'$, genau dann wenn

$$\forall\, p: \quad \delta_a(q,p) = \delta_a(q',p) \quad \wedge \quad \delta_a(p,q) = \delta_a(p,q')$$
$$\wedge \quad \mathsf{select}_a(q,p) = \mathsf{select}_a(q',p) \quad \wedge \quad \mathsf{select}_a(p,q) = \mathsf{select}_a(p,q')$$

Im Beispiel:

$$Q_1 = \{\bar{q}_0, \bar{q}_2, \bar{q}_3, \bar{q}_4\}$$
$$Q_2 = \{\bar{q}_1\}$$

mit:

+	Q_1	Q_2
Q_1	$\bar{q}_3$	$\bar{q}_2$
Q_2	$\bar{q}_2$	$\bar{q}_4$

In der Praxis werden die Zeilen und Spalten nicht sehr oft identisch sein. Das Verfahren ist also etwas zu sensibel.

Idee 2: Zeilenverschiebung

Man geht von dichten zu spärlichen Matrizen über. In der Numerik werden i.A. die Null (0) nicht repräsentiert. Bei unserem Verfahren werden die häufigsten Einträge nicht repräsentiert.

Sind viele Einträge gleich (im Beispiel etwa default $= \bar{q}_3$), genügt es, die übrigen Einträge zu speichern ;-)

Im Beispiel:

+	$\bar{q}_0$	$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_4$
$\bar{q}_0$		$\bar{q}_2$			
$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_4$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$
$\bar{q}_2$		$\bar{q}_2$			
$\bar{q}_3$		$\bar{q}_2$			
$\bar{q}_4$		$\bar{q}_2$			

Dann legen wir:

(1) gleiche Zeilen übereinander;

(2) verschiedene (Klassen von) Zeilen auf Lücke verschoben übereinander:

	$\bar{q}_0$	$\bar{q}_1$	$\bar{q}_2$	$\bar{q}_3$	$\bar{q}_4$
class	0	1	0	0	0

	0	1
disp	0	2

	0	1	2	3	4	5	6
A	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_4$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$	$\bar{q}_2$
valid	0	0	1	1	1	1	1

Für jeden Eintrag im ein-dimensionalen Feld A vermerken wir in valid , zu welcher Zeile der Eintrag gehört ...

Ein Feld-Zugriff $\delta_+(\bar{q}_i, \bar{q}_j)$ wird dann so realisiert:

$$
\begin{array}{rcl}
\delta_+(\bar{q}_i, \bar{q}_j) & = & \mathsf{let}\ \ c \ = \ \mathsf{class}[\bar{q}_i]; \\
& & \qquad d \ = \ \mathsf{disp}[c]; \\
& & \mathsf{in}\ \ \mathsf{if}\ \ (\mathsf{valid}[d+j] \equiv c) \\
& & \qquad \mathsf{then}\ \ \mathsf{A}[d+j] \\
& & \qquad \mathsf{else}\ \ \mathsf{default} \\
& & \mathsf{end}
\end{array}
$$

Bemerkung:
Die Automaten werden unter Einbeziehung der Kostenfunktion konstruiert. Es entstehen grosse Tabellen, die mit verschiedenen Standardverfahren komprimiert werden können. Diese Verfahren sollen auch die Zugriffe auf diese komprimierten Tabellen realisieren.

Diskussion:

- Die Tabellen werden i.a. erheblich kleiner.
- Dafür werden Tabellenzugriffe etwas teurer.
- Das Verfahren versagt in einigen (theoretischen) Fällen.
- Dann bleibt immer noch das dynamische Verfahren ...

  möglicherweise mit Caching der einmal berechneten Werte, um unnötige Mehrfachberechnungen zu vermeiden :-)

Dieses Caching-Verfahren wird u.A. von grep und fgrep verwandt.

Reinhard Wilhelm, Saarbrücken

Bemerkung:
Für jeden Operator (z.B. +) wird eine Tabelle angelegt. Die Anzahl der Dimensionen dieser Tabelle entspricht der Stelligkeit des Operators. (+ ist zweistellig, also 2-dimensionale Matrix)
Da diese Matrizen sehr gross werden können, werden sie durch spezielle Verfahren komprimiert.

Diskussion:

- Die Tabellen werden i.a. erheblich kleiner.
- Dafür werden Tabellenzugriffe etwas teurer.
- Das Verfahren versagt in einigen (theoretischen) Fällen.
- Dann bleibt immer noch das dynamische Verfahren ...

  möglicherweise mit Caching der einmal berechneten Werte, um unnötige Mehrfachberechnungen zu vermeiden :-)

Matrizenkomprimierung ist allgemein auch ein Problem beim Compilerbau. (z.B Parsertabellen)

3.3 Instruction Level Parallelität

Optimierungen sollten die speziellen Architekturen eines Prozessors zur Verbesserung der Effizienz der auszuführenden Programmen nützen.

Moderne Prozessoren führen nicht eine Instruktion nach der anderen aus.

Wir betrachten hier zwei Ansätze:

(1) VLIW (Very Large Instruction Words)
(2) Pipelining

VLIW:

Eine Instruktion führt simultan bis zu k (etwa 4:-) elementare Instruktionen aus.

Pipelining:

Instruktionsausführungen können zeitlich überlappen.

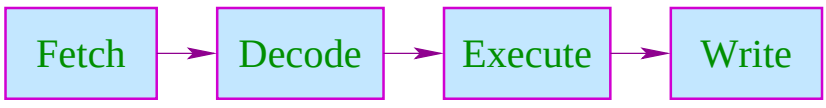
VLIW und Pipelining werden natürlich auf modernen Prozessoren verknüpft angewandt.

Beispiel: *VLIW*

$$w = (R_1 = R_2 + R_3 \mid D = D_1 * D_2 \mid R_3 = M[R_4])$$

Achtung:

- Instruktionen belegen Hardware-Einrichtungen. *Register, Datenpfade, Alus*
- Instruktionen greifen auf die gleichen Register zu $\Longrightarrow$ Hazards
- Ergebnisse einer Instruktion liegen erst nach einiger Zeit vor.
- Während dieser Zeit wechselt i.a. die benutzte Hardware:



Holen der Instruktion aus dem Programm-Cache oder aus dem Hauptspeicher, Dekodierung der Instruktion, Ausführen, Schreiben in den Speicher oder in ein Register.

- Während Execute bzw. Write werden evt. unterschiedliche interne Register/Busse/Alus benutzt.

Wir schließen:

Aufteilung der Instruktionsfolge in Wörter und ihre Aufeinanderfolge
ist Restriktionen unterworfen ...

Im folgenden ignorieren wir die Phasen Fetch und Decode :-)

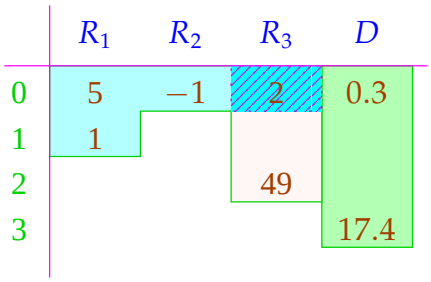
Beispiele für Restriktionen:

(1) maximal ein Load/Store pro Wort;
(2) maximal ein Jump;
(3) maximal ein Write in das selbe Register.

Timing:

Gleitkomma-Operation	3
Laden/Speichern	2
Integer-Arithmetik	1

Timing-Diagramm:



R_3 wird überschrieben, nachdem die Addition 2 abgeholte :-)

Wird auf ein Register mehrfach zugegriffen (hier: R_3),
wird eine Strategie zur Konfliktlösung benötigt ...

Konflikte:

Read-Read: Ein Register wird mehrfach ausgelesen.

$\Longrightarrow$ i.a. unproblematisch :-)

Read-Write: Ein Register wird in einer Instruktion sowohl gelesen wie geschrieben.

Lösungsmöglichkeiten:

- ... verbieten!
- Lesen wird verzögert (stalls), bis Schreiben beendet ist!
- Lesen zeitlich vor dem Schreiben liefert den alten Wert!
  Gleichzeitiges Lesen wird verzögert/verboten/bevorzugt.

Write-Write: Ein Register wird mehrfach beschrieben.

$\Longrightarrow$ i.a. problematisch :-)

Lösungsmöglichkeiten:

- ... verbieten!
- ...

In unseren Beispielen ...

- erlauben wir gleichzeitiges Lesen;
- verbieten wir gleichzeitiges Schreiben bzw. Schreiben und Lesen;
- fügen wir keine Stalls ein.

Wir betrachten erst mal nur Basis-Blöcke, d.h. Folgen von Zuweisungen ...

Man muss also die Datenabhängigkeit der einzelnen Instruktionen herausfinden, repräsentieren und benutzen.

Idee: Datenabhängigkeitsgraph

Knoten	Instruktionen
Kanten	Abhängigkeiten

Beispiel:

$$
\begin{array}{ll}
(1) & x = x + 1; \\
(2) & y = M[A]; \\
(3) & t = z; \\
(4) & z = M[A + x]; \\
(5) & t = y + z;
\end{array}
$$

Mögliche Abhängigkeiten:

Definition	$\rightarrow$	Use	//	Reaching Definitions
Use	$\rightarrow$	Definition	//	???
Definition	$\rightarrow$	Definition	//	Reaching Definitions

Um festzustellen welche Definitionen vor einer Benutzung stehen müssen, benötigt man eine Programmanalyse.

Reaching Definitions: Ankommende Definitionen

Ermittle für jedes u , welche Variablen-Definitionen ankommen
$\Longrightarrow$ mithilfe Ungleichungssystem berechenbar :-)

Der abstrakte Bereich:

$\mathbb{R} = 2^{Nodes}$ // Man hätte auch Kanten nehmen können :-)

Da es sich hier um Zuweisungskanten handelt, sind diese durch ihre Anfangspunkte (Knoten) eindeutig definiert: Somit kann dieser Knoten einer solchen Kante als deren Definition benutzt werden. Es gibt ja nur bei den Bedingungen Verzweigungen, bei den Definitionen nicht.

Die Transfer-Funktionen:

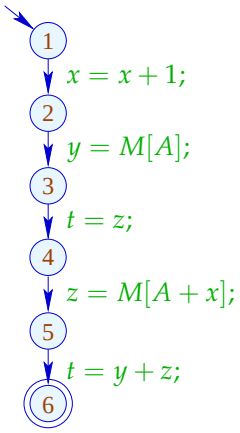
$$
\begin{array}{lcl}
[\![(\_, ;, \_)]\!]^\sharp\, R & = & R \\
[\![(\_, Pos(e), \_)]\!]^\sharp\, R & = & [\![(\_, Neg(e), \_)]\!]^\sharp\, R \;=\; R \\
[\![(u, x = e;, \_)]\!]^\sharp\, R & = & (R \backslash Defs_x) \cup \{u\} \qquad \text{wobei} \\
 & & Defs_x \text{ die Menge der Definitionen von } x \text{ ist} \\
[\![(u, x = M[A];, \_)]\!]^\sharp\, R & = & (R \backslash Defs_x) \cup \{u\} \\
[\![(\_, M[A] = x;, \_)]\!]^\sharp\, R & = & R
\end{array}
$$

Die Information wird offenbar vorwärts propagiert, wobei die Ordnung auf dem vollständigen Verband $\mathbb{R}$ "$\subseteq$" ist :-)

Vor Programm-Ausführung ist die Menge der ankommenden Definitionen
$d_0 = \{\bullet_x \mid x \in Vars\}$.

Hier nicht die leere Menge! Es werden Dummy-Knoten eingeführt, da die Variablen vor dem Betreten des Basic-Blocks einen wohldefinierten Wert haben.

... im Beispiel:



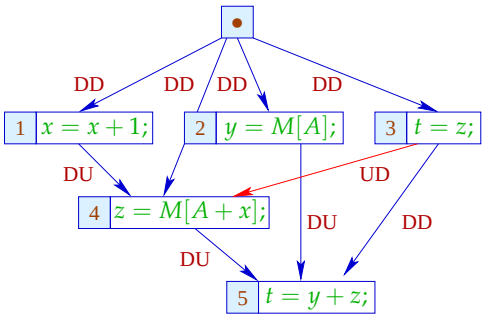
	$\mathcal{R}$
1	$\{\bullet_x, \bullet_y, \bullet_z, \bullet_t\}$
2	$\{1, \bullet_y, \bullet_z, \bullet_t\}$
3	$\{1,2, \bullet_z, \bullet_t\}$
4	$\{1,2,3, \bullet_z\}$
5	$\{1,2,3,4\}$
6	$\{1,2,4,5\}$

Seien U_i, D_i die Mengen der an einer von u_i ausgehenden Kante benutzten bzw. definierten Variablen. Dann gilt:

$$(u_1, u_2) \in DD \quad \text{falls} \quad u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap D_2 \neq \emptyset$$
$$(u_1, u_2) \in DU \quad \text{falls} \quad u_1 \in \mathcal{R}[u_2] \wedge D_1 \cap U_2 \neq \emptyset$$

... im Beispiel:

		Def	*Use*
1	$x = x + 1;$	$\{x\}$	$\{x\}$
2	$y = M[A];$	$\{y\}$	$\{A\}$
3	$t = z;$	$\{t\}$	$\{z\}$
4	$z = M[A + x];$	$\{z\}$	$\{A, x\}$
5	$t = y + z;$	$\{t\}$	$\{y, z\}$



Am Programmpunkt 3 und 5 haben wir eine Definition von t. Wir haben dabei eine DD-Abhängigkeit. Am Programmpunkt 3 wird das z benutzt, an 4 überschrieben, UD-Abhängigkeit.

Die UD-Kante $(3, 4)$ haben wir eingefügt, um zu verhindern, dass z vor der Benutzung überschrieben wird :-)

Im nächsten Schritt versehen wir jede Instruktion mit (ihren benötigten Resourcen, insbesondere) ihrer Zeit.
Man müsste natürlich auch noch die verwendeten Datenpfade, databusses, Register und Alus mit vermerken.

Man muss also vermerken, dass die Zuweisung 1 einen Takt, die Zuweisung 2 zwei Takte,die Zuweisung 3 einen Takt, die Zuweisung 4 zwei Takte und die Zuweisung 5 einen Takt benötigt. usw.

Wir wollen eine möglichst parallele korrekte Wortfolge bestimmen.
Dazu verwalten wir den aktuellen System-Zustand:

$$\Sigma : \mathit{Vars} \to \mathbb{N}$$

$$\Sigma(x) \;\hat{=}\; \text{zu wartende Zeit, bis } x \text{ vorliegt}$$

Am Anfang:

$$\Sigma(x) = 0$$

Wir müssen als Invariante garantieren, dass alle Operationen bei Betreten des Basisblocks abgeschlossen sind :-)

Dann füllen wir sukzessive die Slots der Wort-Folge:

- Wir beginnen bei den minimalen Knoten des Abhängigkeitsgraphen.
- Können wir nicht alle Slots eines Worts füllen, fügen wir ; ein :-)
- Nach jeder eingefügten Instruktion berechnen wir Σ neu.

Achtung:

→ Die Ausführung zweier VLIWs kann überlappen !!!

→ Die Berechnung einer optimalen Folge ist NP-hart ...

Beispiel: Wortbreite $k = 2$

Wort		Zustand			
1	2	x	y	z	t
		0	0	0	0
$x = x + 1$	$y = M[A]$	0	1	0	0
$t = z$	$z = M[A + x]$	0	0	1	0
		0	0	0	0
$t = y + z$		0	0	0	0

In jedem Takt beginnt die Ausführung eines neuen Worts.
Im Zustand brauchen wir uns nur merken, wieviele Takte auf das Ergebnis noch gewartet werden muss :-)

Beachte:

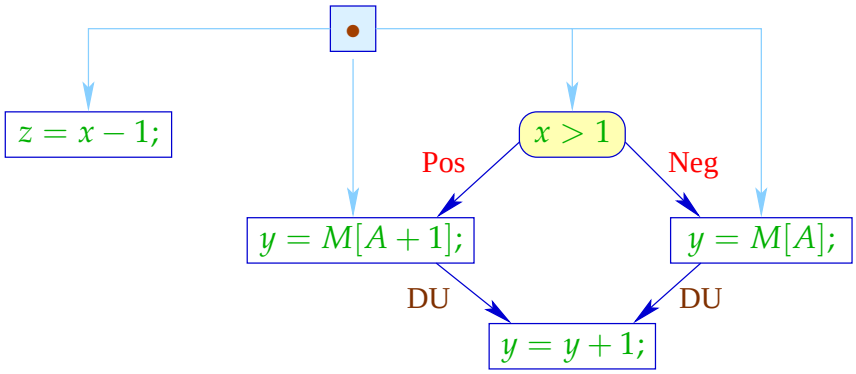
- Wenn Instruktionen zukünftiger Wortwahl weitere Restriktionen auferlegen, vermerken wir diese ebenfalls in Σ .
- Trotzdem unterscheiden wir nur endlich viele System-Zustände :-)
- Die Berechnung des Effekts eines VLIW auf Σ lässt sich in einen endlichen Automaten compilieren !!!
- Dieser Automat könnte allerdings sehr groß sein :-(
- Die Qual der billigsten Auswahl erspart er uns nicht :-(
- Basis-Blöcke sind leider i.a. nicht sehr groß

  $\Longrightarrow$ die Möglichkeiten zur Parallelisierung sind beschränkt :-((

Erweiterung 1: Azyklischer Code

$$
\begin{array}{l}
\text{if } (x > 1) \ \{ \\
\qquad y = M[A]; \\
\qquad z = x - 1; \\
\} \ \text{else} \ \{ \\
\qquad y = M[A + 1]; \\
\qquad z = x - 1; \\
\} \\
y = y + 1;
\end{array}
$$

Im Abhängigkeitsgraph müssen wir zusätzlich die Kontroll-Abhängigkeiten vermerken ...



Das Statement $z = x - 1;$ wird mit immer den gleichen Argumenten in beiden Zweigen ausgeführt und modifiziert keine der sonst benutzten Variablen :-)

Wir hätten es ohnehin vor das if schieben können :-))

Als Code können wir deshalb erzeugen:

	$z = x - 1$	if $!(x > 0)$ goto A
	$y = M[A]$	
	goto B	
$A:$	$y = M[A + 1]$	
$B:$	$y = y + 1$	

Die Zuweisung z = x-1; kommt in beiden Zweigen der Bedingung unabhängig vor. Optimierend kann diese vor oder nach der Bedingung gestellt werden.

Bei jedem Einsprung garantieren wir die Invariante :-(

Erlauben wir mehrere (bekannte) Zustände beim Betreten eines Teil-Basisblocks, können wir für diesen Code erzeugen, der allen diesen Bedingungen entspricht.

... im Beispiel:

	$z = x - 1$	if $!(x > 0)$ goto A
	$y = M[A]$	goto B
$A:$	$y = M[A + 1]$	
$B:$		
	$y = y + 1$	

Es findet quasi ein delayed load statt. Das Ergebnis ist erst nach dem Sprung vorhanden.
Deshalb Sprung an eine NOP-Stelle.

Reicht uns diese Parallelität immer noch nicht, könnten wir versuchen, spekulativ Arbeit vorziehen ...

Dazu erforderlich:

- eine Idee, welche Alternative häufiger gewählt wird;
- die falsche Ausführung darf zu keiner Katastrophe d.h. Laufzeitfehlern führen (z.B. wegen Division durch 0);
- die falsch Ausführung muss rückgängig gemacht werden können (evt. durch verzögertes Commit) oder darf keinen beobachtbaren Effekt haben ...

Erlauben wir mehrere (bekannte) Zustände beim Betreten eines Teil-Basisblocks, können wir für diesen Code erzeugen, der allen diesen Bedingungen entspricht.

Breitere Instruktionsworte kann man nun spekulativ füllen.
Bei Korrektheit wird das Ergebnis z.B. durch ein Commit aktiviert.

... im Beispiel:

Die Zuweisung $y = M[A]$ *wird spekulativ vor die Abfrage gestellt.*
Bei Erfüllung der Bedingung ist das Ergebnis schon vorhanden.
Bei Nichterfüllung wird die Zuweisung $y = M[A+1]$ *ausgeführt.*
Es entsteht also kein Problem.

	$z = x - 1$	$y = M[A]$	if $(x > 0)$ goto B
	$y = M[A+1]$		
$B:$			
	$y = y + 1$		

Im Fall $x \leq 0$ haben wir $y = M[A]$ zuviel ausgeführt.
Dieser Wert wird aber im nächsten Schritt direkt überschrieben :-)

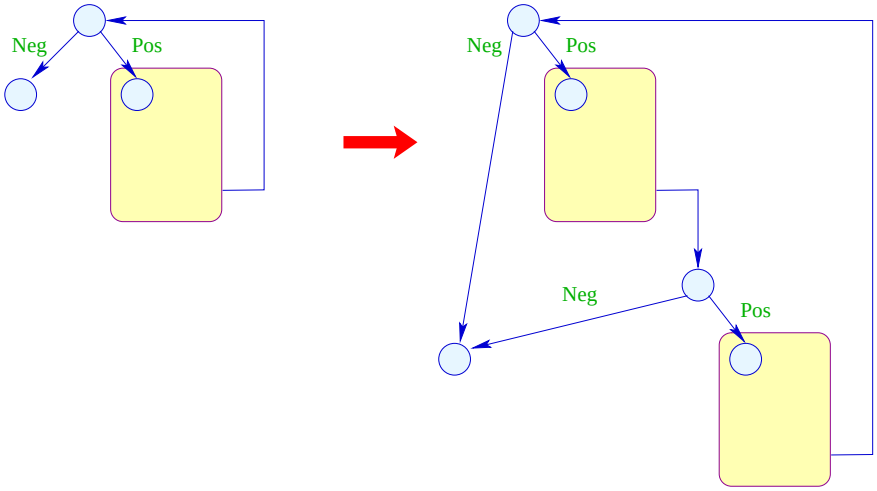
Allgemein:

$x = e;$ hat keinen beobachtbaren Effekt in einem Zweig,
falls x in diesem Zweig tot ist :-)

Trace-scheduling: Spekulativ wird ein bevorzugter Programmierpfad gewählt.
Man setzt spekulativ auf diesen Pfad. Wird dieser Pfad nicht gewählt , muss der Korrekturcode ausgeführt werden.
Spekulativ sollte natürlich der Weg gewählt werden, welcher der Normalfall ist.
Dies ist im Allgemeinen schwierig.

Erweiterung 2: Abwickeln von Schleifen

Wir wickeln wichtige, d.h. innere Schleifen mehrmals ab:



Nun ist auch klar, welche Seite bei Tests zu begünstigen ist:
diejenige, die innerhalb des abgerollten Rumpfs der Schleife bleibt :-)

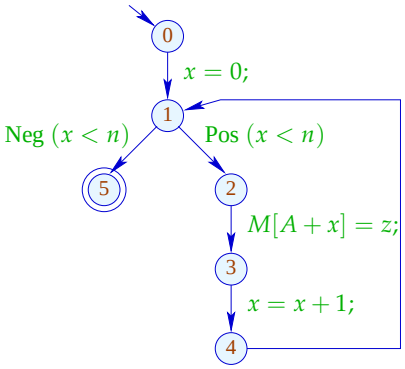
Achtung:

- Die verschiedenen Instanzen des Rumpfs werden relativ zu möglicherweise unterschiedlichen Anfangszuständen übersetzt :-)
- Der Code hinter der Schleife muss gegenüber dem Endzustand jedes Sprungs aus der Schleife korrekt sein!

Die basic-blocks stehen nun übereinander und man kennt die Einsprungbedingungen.
Durch Berücksichtugung dieser Einsprungbedingungen ist die Codeerzeugung einfacher, im Gegensatz zur Invariante, bei der alle Daten vorhanden sein müssen.

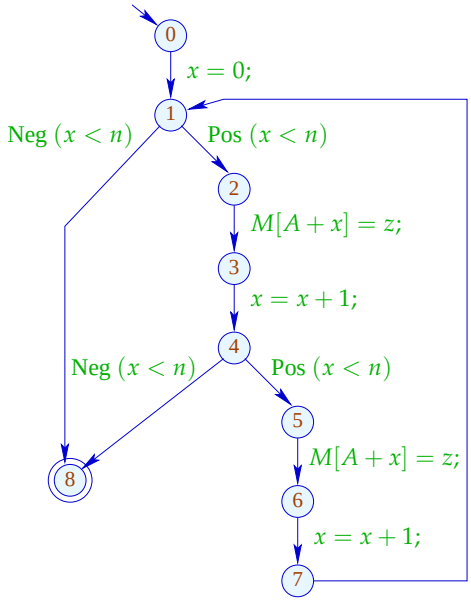
Beispiel: *for-Schleife. Hier mit numerischem Code, dieser ist optimal für eine Optimierung*

$$\text{for } (x = 0; x < n; x{+}{+})$$
$$M[A + x] = z;$$



Verdoppelung des Rumpfs liefert:

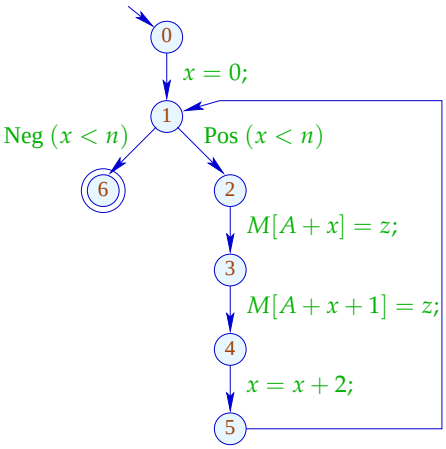
```
for (x = 0; x < n; x++) {
        M[A + x] = z;
        x = x + 1;
        if (!(x < n)) break;
        M[A + x] = z;
}
```



Störend : Zwischenabfrage; Zweifache Modifizierung von x.
Abhilfe : Propagierung des x an die Stellen seiner Benutzung.

Besser wäre es, wenn wir auf den Test in der Mitte verzichten könnten.
Das ist möglich, wenn wir wissen, dass n stets gerade ist :-)
Dann haben wir:

```
for (x = 0; x < n; x = x + 2) {
        M[A + x] = z;
        M[A + x + 1] = z;
}
```



Diskussion:

- Beseitigung der Zwischenabfrage zusammen mit Verschieben des Zwischen-Inkrements ans Ende zeigt, dass die verschiedenen Rumpf-Iterationen in Wahrheit unabhängig sind :-)

- Wir gewinnen trotzdem nicht viel, da wir nur maximal ein Store pro Wort gestatten :-(
- Sind die rechten Seiten allerdings komplizierter, könnten wir deren Auswertung mit je einem Store pro Takt verschränken :-)

Lässt die Architektur die Möglichkeit des parallelen Store zu, ergibt sich Effizienz.

Erweiterung 3: *loop fusion*

Möglicherweise bietet eine Schleife allein nicht genug Möglichkeiten zur Parallelisierung :-(
... möglicherweise aber zwei aufeinander folgende :-)

Beispiel:

$$
\begin{array}{ll}
\textsf{for}\ (x = 0; x < n; x{+}{+})\ \{ & \textsf{for}\ (x = 0; x < n; x{+}{+})\ \{ \\
\quad R = M[B + x]; & \quad R = M[B + x]; \\
\quad S = M[C + x]; & \quad S = M[C + x]; \\
\quad T_1 = R + S; & \quad T_2 = R - S; \\
\quad M[A + x] = T_1; & \quad M[C + x] = T_2; \\
\} & \}
\end{array}
$$

Um beide Schleifen zu einer zusammen zu fassen, muss:

- das Iterations-Schema übereinstimmen;
- die beiden Schleifen greifen auf unterschiedliche Daten zu.

Im Falle von einzelnen Variablen lässt sich das leicht verfizieren.
Schwieriger ist das in Anwesenheit von Pointern oder Feldern. *Alias-analyse?*
Unter Rückgriff auf das Source-Programm kann man Zugriffe auf statisch allokierte disjunkte Felder erkennen.
Problem ergibt sich bei Programmiersprachen bei denen Felder überlappen können
Analyse von Zugriffen auf das gleiche Feld ist erheblich schwieriger ...

Nehmen wir für das Beispiel an, die Bereiche
$[A, A + n - 1], [B, B + n - 1], [C, C + n - 1]$ überlappen nicht.
Offenbar können wir dann die beiden Schleifen kombinieren zu:

$$
\begin{array}{ll}
\text{for } (x = 0; x < n; x{+}{+}) \ \{ & \\
\quad R = M[B + x]; & \quad R = M[B + x]; \\
\quad S = M[C + x]; & \quad S = M[C + x]; \\
\quad T_1 = R + S; & \quad T_2 = R - S; \\
\quad M[A + x] = T_1; & \quad M[C + x] = T_2; \\
\} &
\end{array}
$$

Die erste Schleife darf in Iteration x auf keine Daten zugreifen,
die die zweite Schleife in Iterationen $< x$ modifiziert.

Die zweite Schleife darf in Iteration x auf keine Daten zugreifen,
die die erste Schleife in Iterationen $> x$ überschreibt.

I.a. muss man dazu die Indexausdrücke analysieren.

Die Indexausdrücke des Lesens und des Schreibens in den Schleifen müssen auf Konflikte untersucht werden.

Sind diese linear, führt das auf Probleme des integer linear programming:

Diese ILP-Algorithmen werden zur Optimierung von betriebswirtschaftlichen Problemen (z.B. Produktionsabläufe] eingesetzt.

$$
\begin{array}{rcl}
x_{\mathsf{write}} & \geq & C \\
x_{\mathsf{write}} & \leq & C + x - 1 \\
x_{\mathsf{read}} & = & C + x \\
x_{\mathsf{read}} & = & x_{\mathsf{write}}
\end{array}
$$

... hat offenbar keine Lösung :-)
Keine Lösung: Kein Konflikt!

Allgemeine Form:

$$
\begin{array}{rcl}
x & \geq & t_1 \\
t_2 & \geq & x \\
y & = & s \\
x & = & y
\end{array}
$$

für lineare Ausdrücke s, t_1, t_2 über den Iterations-Variablen.
Das lässt sich vereinfachen zu:

$$0 \leq s - t_1 \qquad 0 \leq t_2 - s$$

Was macht man damit ???

Einfacher Fall:

Die beiden Ungleichungen haben über $\mathbb{Q}$ eine leere Lösungsmenge.
Dann ist die Lösungsmenge auch über $\mathbb{Z}$ leer :-)

In unserem Beispiel:

$$\begin{array}{rcll} 0 & \leq & C + x - C & = & x \\ 0 & \leq & C + x - 1 - (C + x) & = & -1 \end{array}$$

Die zweite Ungleichung hat überhaupt keine Lösung :-)

Gleiche Vorzeichen:

Kommt eine Variable x in allen Ungleichungen mit gleichem Vorzeichen vor, gibt es immer eine Lösung :-(

Es gibt also immer eine rationale, sowie auch eine Integer-Lösung!

Beispiel:

$$\begin{array}{rcl} 0 & \leq & 13 + 7 \cdot x \\ 0 & \leq & -1 + 5 \cdot x \end{array}$$

Man muss x nur wählen als:

$$x \geq \max(-\frac{13}{7}, \frac{1}{5}) = \frac{1}{5}$$

Ungleiche Vorzeichen:

Eine Variable x kommt in einer Ungleichung negativ, in allen anderen höchstens positiv vor. Dann kann man ein Ungleichungssystem ohne x konstruieren ...

Beispiel:

$$\begin{array}{rcl} 0 & \leq & 13 - 7 \cdot x \\ 0 & \leq & -1 + 5 \cdot x \end{array} \iff \begin{array}{rcl} x & \leq & \frac{13}{7} \\ 0 & \leq & -1 + 5 \cdot x \end{array}$$

Man findet die obere Grenze für x, nämlich $(\frac{13}{7})$*, diese kann man in die untere Ungleichung einsetzen. Diese Ungleichung wird dadurch nur abgeschwächt (Da Obere Schranke).*
Es gibt eine rationale Lösung.

Da $0 \leq -1 + 5 \cdot \frac{13}{7}$ hat das System eine rationale Lösung ...

Eine Variable:

Die Ungleichungen, in denen x positiv vorkommt, liefern untere Schranken.
Die Ungleichungen, in denen x negativ vorkommt, liefern obere Schranken.
Seien G, L die grösste untere bzw. kleinste obere Schranke.
Dann liegen alle (ganzzahligen) Lösungen im Intervall $[G, L]$:-)

Beispiel:

$$\begin{array}{rcl} 0 & \leq & 13 - 7 \cdot x \\ 0 & \leq & -1 + 5 \cdot x \end{array} \iff \begin{array}{rcl} x & \leq & \frac{13}{7} \\ x & \geq & \frac{1}{5} \end{array}$$

Die einzige ganzzahlige Lösung des Systems ist $x = 1$:-)

Diskussion:

- Lösungen sind natürlich immer nur innerhalb der Grenzen der Iterationsvariablen interessant.
- Jede ganzzahlige Lösung dort liefert einen Konflikt.
- Verschränkte Berechnung der Schleifen ist möglich, sofern es keinerlei Konflikte gibt :-)
- Die angegebenen Spezialfälle reichen, um den Fall von zwei Ungleichungen über $\mathbb{Q}$ bzw. einer Variable über $\mathbb{Z}$ zu behandeln.
- Die Anzahl der Variablen in den Ungleichungen entspricht der Anzahl der geschachtelten for-Schleifen $\Longrightarrow$ sie ist i.a. klein :-)

Diskussion:

Eine Reihe von Schedulingproblemen lassen sich mit ILP (Integer Linear Programming) lösen. z.B. Einführung von Variablen für den Zeitpunkt des Starts von tasks. Dauer und Anzahl der slots ist bekannt. Dann ist das kombinatorische System durch eine Reihe von Ungleichungen darstellbar. (Aber : NP-Schwierig!)

- Integer Linear Programming (ILP) kann die Erfüllbarkeit herausfinden einer endlichen Menge von Gleichungen/Ungleichungen über $\mathbb{Z}$ der Form:

$$\sum_{i=1}^{n} a_i \cdot x_i = b \quad \text{bzw.} \quad \sum_{i=1}^{n} a_i \cdot x_i \geq b\ , \quad a_i \in \mathbb{Z}$$

- Darüber hinaus kann eine (lineare) Zielfunktion optimiert werden :-)
- Achtung: Bereits das Entscheidungsproblem ist i.a. NP-schwierig !!!
- Trotzdem gibt es erstaunlich effiziente Implementierungen.
- Nicht nur Schleifen-Verschmelzung, auch andere Umstrukturierungen von Schleifen führen auf ILP-Probleme ...

ILP-Implementierungen werden auch im Compilerbau angewandt

Exkurs 5: Presburger Arithmetik

ILP ist eine Ausprägung von Presburger Arithmetik.

Viele Probleme der Informatik lassen sich ohne Multiplikation formulieren :-)

Wir betrachten hier erst einmal zwei einfache Spezialfälle ...

Spezialfälle von ILP.
Dieses Gleichungssystem soll nun nicht über $\mathbb{Q}$ gelöst werden, sondern über $\mathbb{Z}$.
Man darf also nicht dividieren. Gauss-Elimination kann nicht angewandt werden.

1. Lineare Gleichungen

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

Fragen:

- Gibt es eine Lösung über $\mathbb{Q}$? *Kann mit Gauss-Elimination gelöst werden.*
  Problem: Die Koeffizienten können exponentiell gross werden.
- Gibt es eine Lösung über $\mathbb{Z}$? *Kann mit modifizierter Gauss-Elimination gelöst werden.*
- Gibt es eine Lösung über $\mathbb{N}$? *Rucksackproblem!*

Schauen wir uns dazu nochmal die Gleichungen an:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

Antworten:

- Gibt es eine Lösung über $\mathbb{Q}$? Ja
- Gibt es eine Lösung über $\mathbb{Z}$? Nein
- Gibt es eine Lösung über $\mathbb{N}$? Nein

Komplexität:

- Gibt es eine Lösung über $\mathbb{Q}$? polynomiell
- Gibt es eine Lösung über $\mathbb{Z}$? polynomiell
- Gibt es eine Lösung über $\mathbb{N}$? NP-schwierig

Lösungsverfahren für Integers

Beobachtung 1:

$$a_1x_1 + \ldots + a_kx_k = b \qquad (\forall\, i:\ a_i \neq 0)$$

hat eine Lösung genau dann wenn

$$ggT\{a_1, \ldots, a_k\} \quad | \quad b$$

$$\text{Also wenn } ggT(a_i) = d \quad \rightsquigarrow \quad \exists z_i; \quad \sum a_i z_i = d$$

Erweiterter Euklid Algorithmus. Hierbei sollten die gewählten Zahlen möglichst klein sein.

Beispiel:

$$5y - 10z = 18$$

hat keine Lösung über $\mathbb{Z}$:-)

Beobachtung 2:

Eine Variable mit Koeffizient ± 1 kann beseitigt werden.

Beispiel:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & - & y & + & 5z & = & 3 \end{array}$$

Dann ergibt sich:

$$\begin{array}{rcrcrcr} 2x & + & 3y & & & = & 24 \\ x & = & y & - & 5z & + & 3 \end{array}$$

Durch Einsetzen von x in die erste Gleichung erhält man schliesslich:

$$5y \quad - \quad 10z \quad = \quad 18$$

(Die Gleichung hat keine Lösung, da es keinen ggt gibt.)

Beobachtung 3:

Wenn dasselbe Feld in verschiedenen Schleifen benutzt wird und die Abhängigkeiten der Zugriffe untersucht werden sollen, muss man Indexausdrücke vergleichen. Ein Konflikt tritt dann auf, wenn diese Gleichungs- und Ungleichungssysteme über diese Abhängigkeiten der Indexausdrücke ganzahlige Lösungen haben.

Jede (lösbare) Gleichung kann so massiert werden, dass sie eine Variable mit Koeffizient ± 1 besitzt :-)

... mithilfe von uni-modularen Variablentransformationen :-))

Nehmen wir an, die Gleichung enthalte $a_1x_1 + a_2x_2$ mit

$$\text{ggT}\{a_1, a_2\} = p$$

Idee:

Ersetze x_1, x_2 durch zwei neue Variablen t_1, t_2 so dass zum Einen gilt:

$$\begin{array}{rcrcr} pt_1 & = & a_1x_1 & + & a_2x_2 \\ t_2 & = & b_1x_1 & + & b_2x_2 \end{array}$$

für geeignete b_1, b_2 ... und zum Anderen,

alle Lösungen für t_1, t_2 auch Lösungen für x_1, x_2 ergeben :-)

$\Longrightarrow$ Die inverse Matrix der Transformation:

$$\begin{pmatrix} \frac{a_1}{p} & \frac{a_2}{p} \\ b_1 & b_2 \end{pmatrix}$$

sollte ganzzahlige Koeffizienten haben.

Dies ist der Fall, wenn *Also die Determinante der Matrix* ± 1 *ist.*

$$\frac{a_1}{p}b_2 - \frac{a_2}{p}b_1 \quad = \quad \pm 1$$

also:

$$a_1 b_2 - a_2 b_1 \quad = \quad \pm p$$

Da a_1, a_2 den ggT p haben,
findet Euclid's Algo λ_1, λ_2 mit:

$$a_1\lambda_1 + a_2\lambda_2 \quad = \quad p$$

$\Longrightarrow$

Wähle: $b_1 = -\lambda_2 \qquad b_2 = \lambda_1.$

Dann:

$$\begin{array}{rcl} x_1 & = & \lambda_1 t_1 - \frac{a_2}{p}t_2 \\ x_2 & = & \lambda_2 t_1 + \frac{a_1}{p}t_2 \end{array}$$

Beispiel:

$$\begin{array}{rcl} -2x_1 + 5x_2 + 3x_3 & = & 2 \\ -4x_1 + 3x_2 - 2x_3 & = & -1 \end{array}$$

Euclid: $\lambda_1 = -1 \quad \lambda_2 = -1$

$\Longrightarrow$

$$\begin{array}{rcl} x_1 & = & -t_1 - 3t_2 \\ x_2 & = & -t_1 - 4t_2 \end{array}$$

Ersetzen vom x_1, x_2 mit t_1, t_2 liefert:

$$\begin{array}{rcl} -7t_1 - 26t_2 + 3x_3 & = & 2 \\ t_1 - 2x_3 & = & -1 \end{array}$$

... und wir haben eine Variable beseitigt :-)

Lösen über $\mathbb{N}$

- ... ist von großer praktischer Bedeutung;
- ... hat zur Entwicklung vieler neuer Techniken geführt;
- ... erlaubt leicht die Kodierung NP-schwieriger Probleme;
- ... bleibt schwierig, sogar wenn nur drei Variablen pro Gleichung erlaubt sind.

Rückblick:
Thema : Schleifenumstrukturierung.
In einem Programm stehen 2 Schleifen hintereinander.
Idee : Zusammenlegung der beiden Schleifen.
Bei der Komposition der Rümpfe muss sichergestellt werden, dass z.B. bei einem Zugriff auf das gleiche Feld in den beiden Rümpfen keine Konflikte entstehen. Insbesondere müssen die Indexausdrücke verglichen werden. Man halt also Gleichungen und Ungleichungen zwischen Indexausdrücken. Hat dieses Gleichungs-, Ungleichungssystem eine ganzzahlige Lösung, ist eine Zusammenlegung nicht möglich.
z.B. Hat die Gleichung:
$5x - 17y + 21z = 101$
eine ganzzahlige Lösung?
Diese Gleichung ist in $\mathbb{Z}$ lösbar.

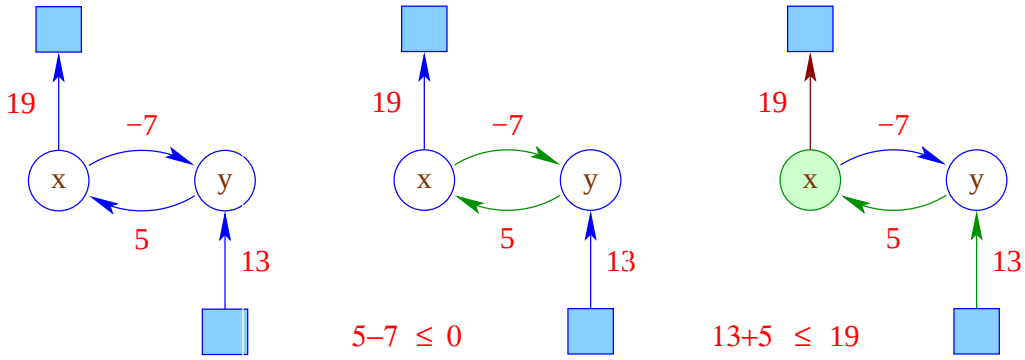
2. Ein polynomieller Spezialfall: *zwei Variablen. Lösung in $\mathbb{Z}$*

$$\begin{array}{rcl} & x & \geq & y+5 \\ 19 & \geq & x & \\ & y & \geq & 13 \\ & y & \geq & x-7 \end{array}$$

- Es gibt maximal zwei Variablen pro Un-Gleichung;
- keine Skalierungsfaktoren.

Man hat Ungleichungen zweierlei Sorten:
Man hat eine obere und untere Schranke der Variablen und den Mindestabstand zwischen Variablen.

Idee: Repräsentiere das System als Graph:



Die Ungleichungen sind erfüllbar genau dann wenn

- die Gewichte jedes Kreises maximal ≤ 0 sind;
  Hier: $(5 - 7 \leq 0)$
- die Gewichte, die x erreichen, maximal $\leq$ der Gewichte sind, die x verlassen.
  Hier: $(13 + 5 \leq 19)$

Lösung z.B.wäre x=18 und y=13

Berechne die reflexive und transitive Hülle der Kanten-Gewichte!

3. Ein allgemeines Lösungsverfahren:

Idee: Fourier-Motzkin-Elimination

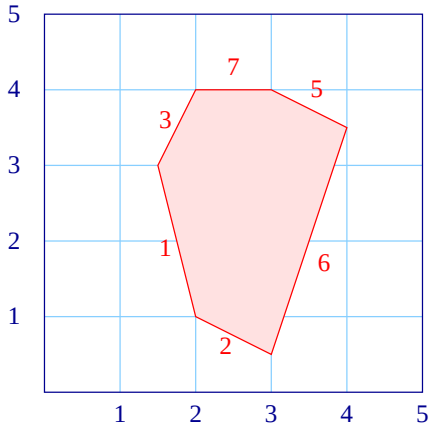
- Beseitige sukzessive einzelne Variablen x !
- Alle Ungleichungen mit positiven Vorkommen von x liefern untere Schranken.
- Alle Ungleichungen mit negativen Vorkommen von x liefern obere Schranken.
- Alle unteren Schranken müssen kleiner oder gleich allen oberen Schranken sein ;-))

Jean Baptiste Joseph Fourier, 1768–1830

Beispiel:

Jede Ungleichung definiert einen Halbraum. Der Schnitt aller Lösungen liefert ein konvexes Polytop.

$$
\begin{array}{rcll}
9 & \leq & 4x_1 + x_2 & (1) \\
4 & \leq & x_1 + 2x_2 & (2) \\
0 & \leq & 2x_1 - x_2 & (3) \\
6 & \leq & x_1 + 6x_2 & (4) \\
-11 & \leq & -x_1 - 2x_2 & (5) \\
-17 & \leq & -6x_1 + 2x_2 & (6) \\
-4 & \leq & -x_2 & (7)
\end{array}
$$



Für x_1 finden wir:

$$
\begin{array}{rcll}
\frac{9}{4} - \frac{1}{4}x_2 & \leq & x_1 & (1) \\
4 - 2x_2 & \leq & x_1 & (2) \\
\frac{1}{2}x_2 & \leq & x_1 & (3) \\
6 - 6x_2 & \leq & x_1 & (4) \\
x_1 & \leq & 11 - 2x_2 & (5) \\
x_1 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (6) \\
-4 & \leq & -x_2 & (7)
\end{array}
$$

Man hat nun jeweils 4 Ungleichungen für die untere Schranke und 2 für die obere Schranke des x_1
Wenn es ein solches x_1 gibt, müssen alle unteren Schranken kleiner gleich allen oberen sein, d.h.:

$$
\begin{array}{rcll}
\frac{9}{4} - \frac{1}{4}x_2 & \leq & 11 - 2x_2 & (1,5) \\
\frac{9}{4} - \frac{1}{4}x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (1,6) \\
4 - 2x_2 & \leq & 11 - 2x_2 & (2,5) \\
4 - 2x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (2,6) \\
\frac{1}{2}x_2 & \leq & 11 - 2x_2 & (3,5) \\
\frac{1}{2}x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (3,6) \\
6 - 6x_2 & \leq & 11 - 2x_2 & (4,5) \\
6 - 6x_2 & \leq & \frac{17}{6} + \frac{1}{3}x_2 & (4,6) \\
-4 & \leq & -x_2 & (7)
\end{array}
$$

$$
\begin{array}{rcll}
-35 & \leq & -7x_2 & (1,5) \\
-\frac{7}{12} & \leq & \frac{7}{12}x_2 & (1,6) \\
-7 & \leq & 0 & (2,5) \\
\frac{7}{6} & \leq & \frac{7}{3}x_2 & (2,6) \\
-22 & \leq & -5x_2 & (3,5) \\
-\frac{17}{6} & \leq & -\frac{1}{6}x_2 & (3,6) \\
-5 & \leq & 4x_2 & (4,5) \\
\frac{19}{6} & \leq & \frac{19}{3}x_2 & (4,6) \\
-4 & \leq & -x_2 & (7)
\end{array}
\qquad \text{oder:} \qquad
\begin{array}{rcll}
-5 & \leq & -x_2 & (1,5) \\
-1 & \leq & x_2 & (1,6) \\
-7 & \leq & 0 & (2,5) \\
\frac{1}{2} & \leq & x_2 & (2,6) \\
-\frac{22}{5} & \leq & -x_2 & (3,5) \\
-17 & \leq & -x_2 & (3,6) \\
-\frac{5}{4} & \leq & x_2 & (4,5) \\
\frac{1}{2} & \leq & x_2 & (4,6) \\
-4 & \leq & -x_2 & (7)
\end{array}
$$

Das ist der Ein-Variablen-Fall, den wir exakt lösen können:

$$\max\{-1, \boxed{\tfrac{1}{2}}, -\tfrac{5}{4}, \tfrac{1}{2}\} \leq x_2 \leq \min\{5, \tfrac{22}{5}, 17, \boxed{4}\}$$

Daraus können wir folgern: $x_2 \in [\frac{1}{2}, 4]$:-)

Im Allgemeinen:

- Das ursprüngliche System hat eine Lösung in $\mathbb{Q}$, gdw. das System nach Eliminierung einer Variable eine Lösung in $\mathbb{Q}$ besitzt :-)
- In jedem Eliminierungsschritt kann sich die Anzahl der Ungleichungen quadrieren $\Longrightarrow$ exponentielle Laufzeit :-((
- Es lässt sich so modifizieren, dass es Erfüllbarkeit über $\mathbb{Z}$ entscheidet $\Longrightarrow$ Omega-Test *Anwendung : z.B. Automatische Parallelisierung von numerischem Code.*

William Worthington Pugh, Jr.
University of Maryland, College Park

Idee:

- Wir beseitigen sukzessive die Variablen. Dabei müssen wir allerdings Divisionen vermeiden ...
  Divisionen entstehen durch die Koeffizienten bei der zu beseitigenden Variablen.
- Hat x überall Koeffizienten ± 1, machen wir Fourier-Motzkin-Elimination :-)
- Andernfalls stellen wir x auf einer Seite mit positivem Koeffizienten frei ...

Betrachten wir etwa (1) und (6) :

$$\begin{aligned} 6 \cdot x_1 &\leq 17 + 2x_2 \\ 9 - x_2 &\leq 4 \cdot x_1 \end{aligned}$$

E.O. können wir echte Ungleichungen betrachten:

$$\begin{aligned} 6 \cdot x_1 &< 18 + 2x_2 \\ 8 - x_2 &< 4 \cdot x_1 \end{aligned}$$

... und jeweils durch den ggT teilen:

$$\begin{aligned} 3 \cdot x_1 &< 9 + x_2 \\ 8 - x_2 &< 4 \cdot x_1 \end{aligned}$$

Das impliziert:

$$3 \cdot (8 - x_2) < 4 \cdot (9 + x_2)$$

Offenbar gilt:

- Ist die abgeleitete Ungleichung unerfüllbar, dann das ganze System :-)
- Sind alle so abgeleiteten Ungleichungen erfüllbar, gibt es eine Lösung, die aber möglicherweise nicht ganzzahlig ist :-(
- Es gibt aber eine ganzzahlige Lösung, sofern zwischen unterer und oberer Schranke stets genug Platz ist, so dass ein Integer dazwischen passt.
- Sei $\alpha < a \cdot x$ $b \cdot x < \beta$.
  Dann muss nicht nur gelten:
  $$b \cdot \alpha < a \cdot \beta$$
  sondern sogar
  $$\boxed{a \cdot b} < a \cdot \beta - b \cdot \alpha$$

... im Beispiel:

$$12 \;<\; 4 \cdot (9 + x_2) - 3 \cdot (8 - x_2)$$

oder:

$$12 \;<\; 12 + 7x_2$$

bzw:

$$0 \;<\; x_2$$

Im Beispiel lassen sich auch diese verschärften Ungleichungen erfüllen
$\Longrightarrow$ das System hat über $\mathbb{Z}$ eine Lösung :-)

Überlegung:

- Sind die verschärften Ungleichungen erfüllbar, dann auch das ursprüngliche System. Die Umkehrung gilt i.A. nicht :-(
- In dem Fall ist bei einem Paar Ungleichungen weniger Platz:

  $$a \cdot \beta \leq b \cdot \alpha + \boxed{a \cdot b}$$

  oder:

  $$b \cdot \alpha < ab \cdot x < b \cdot \alpha + \boxed{a \cdot b}$$

  Kürzen durch b liefert:

  $$\alpha < a \cdot x < \alpha + \boxed{a}$$

  $\Longrightarrow$ $\boxed{\alpha + i = a \cdot x}$ für ein $i \in \{1, \ldots, a-1\}$!!!

Diskussion:

→ Fourier-Motzkin-Elimination ist nicht das beste Verfahren für rationale Ungleichungssysteme.

→ Der Omega-Test ist notwendig exponentiell :-)

Wenn das System lösbar ist, terminiert der Test i.a. schnell.

Mit unlösbaren Systemen tut er sich schwerer :-(

→ Auch für ILP gibt es andere/intelligentere Verfahren ...

→ Für Probleme bei Programmiersprachen funktioniert er wohl ganz gut :-)

4. Verallgemeinerung zu einer Logik

Man versucht ein System von Gleichungen und Ungleichungen zu lösen.
Man sucht eine Lösung von Konjunktionen und Disjunktionen dieser Gleichungen unter Hinzunahme von Quantoren.

Disjunktion:

$$\begin{array}{lcll} (x-2y=15 & \wedge & x+y=7) & \vee \\ (x+y=6 & \wedge & 3x+z=-8) & \end{array}$$

Quantoren:

$$\exists\, x: \quad z-2x=42 \quad \wedge \quad z+x=19$$

Mojzesz Presburger, 1904–1943 (?)

Presburger Arithmetik $=$ normale Arithmetik ohne Multiplikation

Arithmetik : hochgradig unentscheidbar :-(
sogar sogar unvollständig :-((
Es gibt Aussagen der Arithmetik die wahr , aber die nicht beweisbar sind.
$\Longrightarrow$ Hilberts 10tes Problem
$\Longrightarrow$ Gödels Theorem

Presburger Formeln:

$$\begin{array}{lll} \phi & ::= & x + y = z \quad | \quad x = n \quad | \\ & & \phi_1 \wedge \phi_2 \quad | \quad \neg\phi \quad | \\ & & \exists\, x : \quad \phi \end{array}$$

Ziel: PSAT

Man kann Fourier-Motzkin-Elimination mit Quantoren betreiben,
indem man versucht, diese Quantoren zu eliminieren.
Oder, die Lösung durch Konstruktion eines Wort-Automaten für die Presburger Arithmetik.

Finde Werte in $\mathbb{N}$ für die freien Variablen, so dass ϕ gilt ...

Idee: Codiere die Werte der Variablen als Worte :-)

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	**0**	0	0

Für jede Variable gibt es eine Spur in der die Bitfolge der Variable steht.

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	**0**	0	0

Man kann nun jede Spalte als Zeichen auffassen.

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	**0**	0	0

Beobachtung:

Die Menge der erfüllenden Variablenbelegungen ist regulär :-))

$$
\begin{array}{lcll}
\phi_1 \wedge \phi_2 & \Longrightarrow & \mathcal{L}(\phi_1) \cap \mathcal{L}(\phi_2) & \text{(Durchschnitt)} \\
\neg\phi & \Longrightarrow & \overline{\mathcal{L}(\phi)} & \text{(Komplement)} \\
\exists\, x: \quad \phi & \Longrightarrow & \pi_x(\mathcal{L}(\phi)) & \text{(Projektion)}
\end{array}
$$

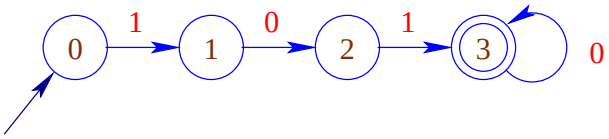
Weg-Projizierung der x-Komponente:

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0
17	x	1	0	0	0	1	0	0	0

213	t	1	0	1	0	1	0	1	1
42	z	0	1	0	1	0	1	0	0
89	y	1	0	0	1	1	0	1	0

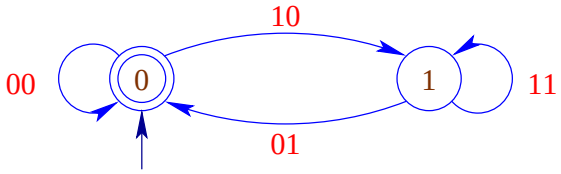
Automaten für Basis-Prädikate:

$x = 5$

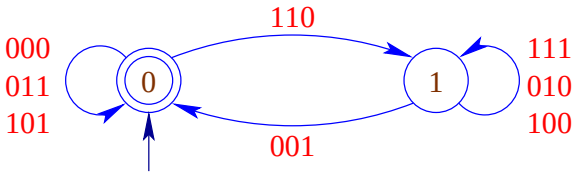


Die Zustände repräsentieren quasi den Übertrag.

$x+x = y$



$x+y = z$



In der Praxis sind die Automaten i.A. sehr klein

Ergebnisse:

Ferrante, Rackoff,1973 : $\text{PSAT} \leq \text{DSPACE}(2^{2^{c \cdot n}})$

Fischer, Rabin,1974 : $\text{PSAT} \geq \text{NTIME}(2^{2^{c \cdot n}})$

3.4 Verbesserung der Speicher-Organisation

Die Speicherorganisiation ist sehr stark von den jeweiligen features des Prozessors abhängig, bzw. vom design des Prozessors.

Ziel:

- Ausnutzung von Caches

  $\Longrightarrow$ Verringerung der Anzahl der Cache-Misses

- Verringerung der Allokations / Deallokations-Kosten

  Nicht nur das Anlegen von Speicherplatz ist kostenintensiv, sondern auch das Halten nichtbenötigter Daten im Speicher.

  $\Longrightarrow$ Ersetzung von Heap-Allokation durch Stack-Allokation

  Insbesondere bei JAVA werden Objekte im Heap angelegt!

  $\Longrightarrow$ Unterstützung der Freigabe überflüssiger Heap-Objekte

- Verringerung der Zugriffskosten

  $\Longrightarrow$ Verkürzung der Indirektionsketten (Unboxing)

  Hier insbesondere bei funktionalen Programmiersprachen. Funktionswerte, sowie auch einfache Integerwerte liegen auf dem heap und müssen dereferenziert werden. Man versucht Integers auf dem Stack zu halten.

1. Cache-Optimierung:

Idee: lokale Speicherzugriffe

Bei einem Zugriff auf den Speicher sollte der nächste Zugriff in der Nähe liegen.
Es wird also ein grösserer Speicherabschnitt in den Cache geladen, dann sind Zugriffe die in der Nähe des vorhergehenden Zugriffs liegen, schnell.

- Laden aus dem Speicher lädt nicht nur ein Byte, sondern füllt eine ganze Cache-Zeile.
- Zugriff auf benachbarte Zellen werden billiger.
- Passen alle Daten einer inneren Schleife in den Cache,
  wird die Iteration extrem speicher-effizient ...

Mögliche Lösungen:

→ Organisiere Zugriffe auf die vorhanden Daten um !

→ Organisiere die Daten um !

Solche Optimierungen funktionieren i.a. automatisch nur für Felder :-(

Diese Optimierung ist auch bei Datenbankanfragen möglich, aber schwieriger zu realisieren.

Beispiel: *Hier wird in der inneren Schleife zuerst über die Spalten iteriert, dies ist ineffizient*

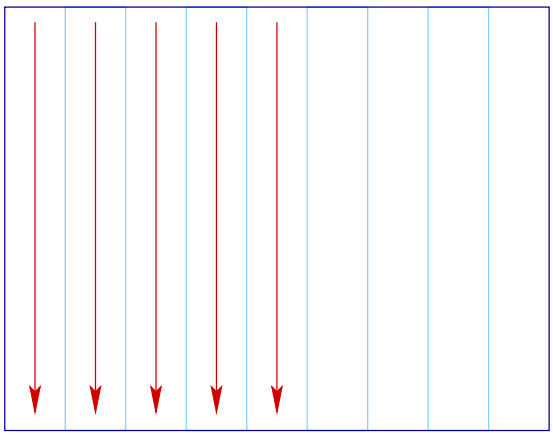
$$\begin{array}{l} \text{for } (j = 1; j < n; j{+}{+}) \\ \quad \text{for } (i = 1; i < m; i{+}{+}) \\ \quad\quad a[i][j] = a[i-1][j-1] + a[i][j]; \end{array}$$

$\Longrightarrow$ Iteriere stets erst über die Zeilen!
$\Longrightarrow$ Vertausche die Reihenfolge der Iterationen:

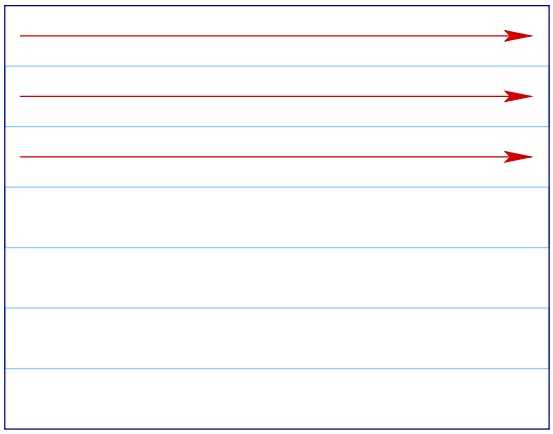
$$\begin{array}{l} \text{for } (i = 1; i < m; i{+}{+}) \\ \quad \text{for } (j = 1; j < n; j{+}{+}) \\ \quad\quad a[i][j] = a[i-1][j-1] + a[i][j]; \end{array}$$

Wann ist das erlaubt ???

Iterations-Schema: vorher: *Iteration über Spalten*

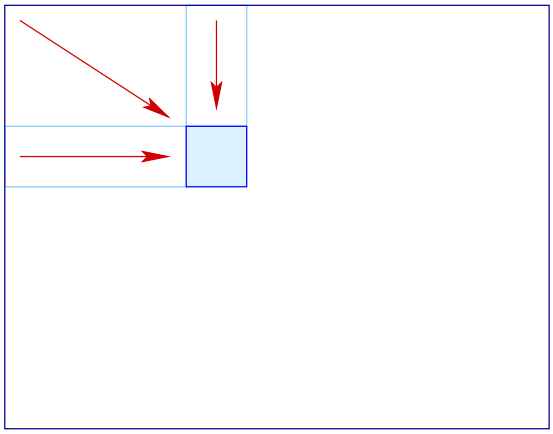


Iterations-Schema: nachher: *Iteration über Zeilen*



Bei der Iteration über die Spalten sind bis zum Erreichen des Elementes mit den Indizies $[i, j]$ *vorher andere Elemente berechnet worden , als würde über die Zeilen iteriert .*
Problem: Falls das aktuelle Element $[i, j]$ *von anderen Elementen der Matrix abhängt, muss sichergestellt sein, dass diese bei beiden Varianten der Iteration richtig berechnet sind.*

Iterations-Schema: erlaubte Abhängigkeiten:



In unserem Fall müssen wir überprüfen, dass die folgenden Gleichungs-Systeme keine Lösung haben:

Schreiben		Lesen
(i_1, j_1)	$=$	$(i_2 - 1, j_2 - 1)$
i_1	$\leq$	i_2
j_2	$\leq$	j_1
(i_1, j_1)	$=$	$(i_2 - 1, j_2 - 1)$
i_2	$\leq$	i_1
j_1	$\leq$	j_2

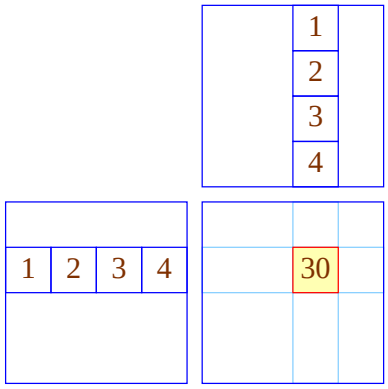
Das erste impliziert: $j_2 \leq j_2 - 1$ Hurra!
Das zweite impliziert: $i_2 \leq i_2 - 1$ Hurra!

Beispiel: Matrix-Matrix-Multiplikation

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        for (k = 0; k < K; k++)
            c[i][j] = c[i][j] + a[i][k] · b[k][j];
```

Über $b[][]$ iterieren wir spaltenweise :-(

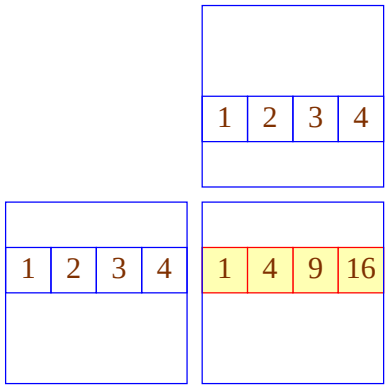
Es wird also die i-te Zeile mit der j-ten Spalte multipliziert.
Der Zugriff über die Spalten ist ineffizient.



Vertausche die beiden inneren Schleifen:

```
for (i = 0; i < N; i++)
    for (k = 0; k < K; k++)
        for (j = 0; j < M; j++)
            c[i][j] = c[i][j] + a[i][k] · b[k][j];
```

Ist das erlaubt ???



Diskussion:

- Die Korrektheit folgt genauso wie eben :-)
- Eine ähnliche Idee lässt sich auch zur Implementierung von Matrix-Multiplikation zeilen-komprimierter Matrizen benutzen :-))
- Möglicherweise muss das Programm erst konditioniert werden,
  damit die Anwendbarkeit der Transformation erkannt wird :-(
- Matrix-Multiplikation benötigt evt. erst eine Initialisierung der Ergebnis-Matrix ...

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++) {
        c[i][j] = 0;
        for (k = 0; k < K; k++)
            c[i][j] = c[i][j] + a[i][k] · b[k][j];
    }
```

Nun können die inneren Schleifen wieder vertauscht werden.

- Jetzt können wir die beiden Iterationen nicht einfach vertauschen :-(
- Wir können aber die Iteration über j duplizieren ...

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) c[i][j] = 0;
    for (j = 0; j < M; j++)
        for (k = 0; k < K; k++)
            c[i][j] = c[i][j] + a[i][k] · b[k][j];
}
```

Zur Korrektheit:

$\Longrightarrow$ Die gelesenen Einträge (hier: keine) dürfen im Rest des Rumpfs nicht modifiziert werden !!!

$\Longrightarrow$ Die Reihenfolge der Schreibzugriffe einer Zelle darf sich nicht ändern :-)

Man erhält:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) c[i][j] = 0;
    for (k = 0; k < K; k++)
        for (j = 0; j < M; j++)
            c[i][j] = c[i][j] + a[i][k] · b[k][j];
}
```

Diskussion:

- Statt mehrere Schleifen zusammen zu fassen, haben wir Schleifen distribuiert :-)
- Desgleichen zieht man Abfragen vor die Schleife $\Longrightarrow$ if-Distribution ...

Achtung:

Statt dieser Transformation könnte man die innere Schleife auch anders optimieren:

$$
\begin{array}{l}
\textsf{for}\ \ (i = 0; i < N; i{+}{+}) \\
\quad \textsf{for}\ \ (j = 0; j < M; j{+}{+})\ \ \{ \\
\qquad t = 0; \\
\qquad \textsf{for}\ \ (k = 0; k < K; k{+}{+}) \\
\qquad\quad t = t + a[i][k] \cdot b[k][j]; \\
\qquad c[i][j] = t; \\
\quad \}
\end{array}
$$

Durch Einführung einer Hilfsvariable t (dies sollte ein Hilfregister sein) erspart man sich einige Speicherzugriffe. Die optimierende Schleifenvertauschung kann nun aber nicht mehr durchgeführt werden. Hier könnte eine Anwendung eines sog. Tempreal erfolgen. Tempreal ist ein spezielles Register für Gleitkomma-Arithmetik, das einige Prozessoren aufweisen. Dieses Register ist aber i.A. für den Programmierer nicht ansprechbar, bzw. nicht zugängig.

Idee:

Finden wir ein heftig benutztes Feld-Element $a[e_1]\ldots[e_r]$, dessen Index-Ausdrücke e_l innerhalb der inneren Schleife konstant sind, können wir stattdessen ein Hilfsregister spendieren :-)

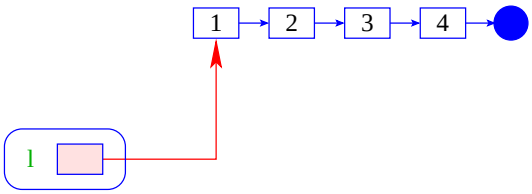
Achtung:

Diese Optimierung verhindert die vorherige und umgekehrt ...

Diskussion:

- Die bisherigen Optimierungen beziehen sich auf Iterationen über Feldern.
- Cache-sensible Organisation anderer Datenstrukturen ist möglich,
  aber i.a. nicht vollautomatisch möglich ...

Beispiel: Keller



Vorteil:

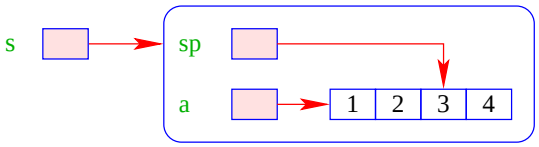
+ Die Implementierung ist einfach :-)
+ Die Operationen push / pop erfordern konstante Zeit :-)
+ Die Datenstruktur ist potentiell beliebig groß :-)

Nachteil:

− Die einzelnen Listenknoten können beliebig über den Speicher verteilt sein :-(

Alternative:

Man nimmt statt der Liste ein Feld. Der Stack besteht nun aus dem Feld und dem Stackpointer



Vorteil:

+ Die Implementierung ist auch einfach :-)
+ Die Operationen push / pop erfordern konstante Zeit :-)
+ Die Daten liegen konsequtiv; Stack-Schwankungen sind im Mittel gering
  $\Longrightarrow$ gutes Cache-Verhalten !!!

Nachteil:

– Die Datenstruktur ist beschränkt :-(

Verbesserung:

- Ist das Feld voll, ersetze es durch ein doppelt so großes !!!
- Wird das Feld leer bis auf ein Viertel, halbiere es wieder !!!

$\Longrightarrow$ Die Extra-Kosten sind amortisiert konstant :-)
$\Longrightarrow$ Die Implementierung ist nicht mehr ganz so trivial :-}

Diskussion:

→ Die gleiche Idee klappt auch für Schlangen :-)

→ Andere Datenstrukturen bemüht man sich, blockweise aufzuteilen.

Problem: wie organisiert man die Zugriffe, dass sie möglichst lange auf dem selben Block arbeiten ???

$\Longrightarrow$ Algorithmen auf externen Daten

Historische Sicht:
Damaliges Problem: Sehr kleiner Hauptspeicher, grosse Datenmengen auf externen Datenträgern mit sehr langsamen Zugriff (Bandgeräte o.Ä)
Heute :
Aktuelles Problem: Cache und Hauptspeicher. Die historischen Praktiken zur Optimierung können auch hier angewandt werden.

2. Stack-Allokation statt Heap-Allokation

Problem:

- Programmiersprachen wie Java legen alle Datenstrukturen im Heap an — selbst wenn sie nur innerhalb der aktuellen Methode benötigt werden :-(
- Überlebt kein Verweis auf diese Daten den Aufruf, wollen wir sie auf dem Stack allokieren :-)

Andernfalls:
Es wird über einen Pointer auf einen nicht mehr vorhandenen Stackframe zugegriffen.

$\Longrightarrow$ Escape-Analyse

Idee: *Da es sich um Pointerzugriffe handelt, wird Alias-Analyse angewandt.*

Berechne Alias-Information.
Bestimme, ob ein erzeugtes Objekt möglicherweise von außen erreichbar ist ...

Beispiel: unsere Pointer-Sprache

$$
\begin{aligned}
&x = \mathsf{new}();\\
&y = \mathsf{new}();\\
&x \rightarrow a = y;\\
&z = y;\\
&\mathsf{return}\ \ z;
\end{aligned}
$$

... könnte ein möglicher Methoden-Rumpf sein ;-)

Von außen zugänglich sind Objekte, die: *Entkommen!*

- von return zurück geliefert werden;
- einer globalen Variablen zugewiesen werden;
- von solchen Objekten erreichbar sind.

... im Beispiel:

$$
\begin{aligned}
&x = \mathsf{new}();\\
&y = \boxed{\mathsf{new}()};\\
&\boxed{x \rightarrow a} = y;\\
&z = \boxed{y};\\
&\mathsf{return}\ \ \boxed{z};
\end{aligned}
$$

Wir schließen:

- Die Objekte, die das erste new() anlegt, können nicht entkommen.
- Wir können sie darum auf dem Stack allokieren :-)

Achtung:

Das ist natürlich nur sinnvoll, wenn von dieser Sorte nur wenige pro Methoden-Aufruf angelegt werden :-(

Liegt deshalb ein solches lokales `new()` in einer Schleife, sollten wir die Objekte vorsichtshalber doch im Heap anlegen ;-)

Erweiterung: Formale Parameter

- Wir benötigen eine interprozedurale Alias-Analyse :-)
  Bisher nur intraprozedurale Alias-Analyse behandelt.
- Kennen wir das gesamte Programm, können wir z.B. die Kontrollflussgraphen der einzelnen Funktionen zu einem einzigen zusammen fassen (durch Hinzufügen geeigneter Kanten) und für diesen Alias-Information berechnen ...
- Achtung: benutzen wir die selben Namen $y_1, y_2, \ldots$ für die formalen Parameter, wird die Information dort notwendig ungenau :-(
- Kennen wir das Gesamtprogramm nicht, müssen wir annehmen, dass jede Referenz, die einer anderen Funktion bekannt ist, entkommt :-((

Escape-Analysen spielen auch bei funktionalen Sprachen eine Rolle (z.B: Unboxing)

3.5 Zusammenfassung

Wir haben jetzt diverse Optimierungen kennen gelernt zur besseren Ausnutzung der Hardware-Gegebenheiten.
Hier teilweise auch konträre Optimierungen (z.b. Schleifenzerlegung und Schleifenvereinigung (Schleifenvereinigung ist erst dann durchzuführen,wenn die Iteration nicht mehr geändert wird))

Reihenfolge ihrer Anwendung:

- Erst globale Restrukturierungen der Prozeduren/Funktionen sowie der Schleifen für besseres Speicherverhalten ;-)
  Mögliche globale Strukturierungen:
  Es wäre auch eine Organisation des gesamten Programms möglich. z.B: Zerlegung in einen funktionalen Teil zur Startup-Zeit und Teile welche bei Bedarf nachgeladen werden. Der Compiler könnte dies durch eine Abhängigkeitsanalyse feststellen.
  Oder auch Aufteilen der Prozeduren in einen Hauptteil und z.B. einen Fehlerbehandlungsteil, der dann nur bei Bedarf geladen wird.
- Dann lokale Umstrukturierung für optimale Nutzung des Instruktionssatzes und der Prozessor-Parallelität :-)
- Dann Registerverteilung und schließlich
- Peephole-Optimierung für den letzten Schliff ...

Funktionen:	Endrekursion + Inlining Stack-Allokation
Schleifen:	Iterationsverbesserung → if-Distribution → for-Distribution Werte-Caching
Rümpfe:	Life-Range-Splitting Instruktions-Auswahl Instruktions-Anordung mit → Schleifen-Abwicklung → Schleifen-Verschmelzung
Instruktionen:	Register-Verteilung Peephole-Optimierung

4 Optimierung funktionaler Programme

SML (Funktoren, Strukuren), OCAML (objektorientiere Features), HASKELL, GOPHER

Beispiel:

$$\begin{array}{lll} \textbf{fun}\ \mathsf{fac}\ x & = & \textbf{if}\ x \leq 1\ \textbf{then}\ 1 \\ & & \textbf{else}\ x \cdot \mathsf{fac}\ (x-1) \end{array}$$

- Es gibt keine Basis-Blöcke :-(
- Es gibt keine Schleifen :-(
- Viele Funktionen sind rekursiv :-((

Strategien zur Optimierung:

$\Longrightarrow$ Verbessere spezielle Ineffizienzen wie:

- Pattern Matching
- Lazy Evaluation (falls vorhanden ;-)
- Indirektionen — Unboxing / Escape-Analyse
- Zwischendatenstrukturen — Deforestation

patternmatching. *Patternmatching ist i.A. ineffizient. Bei der Prüfung auf ein bestimmtes Muster, müssen immer wieder die selben Knoten getestet werden.*
lazy evaluation *z.B. bei HASKELL. Man fängt an den Rumpf einer Funktion auszuwerten, bevor man die Werte für die Argumente kennt. D.f. Striktheitsanalyse wird notwendig zur Beseitigung von Lazy Evaluation.*
unboxing: *JAVA : (integers, etc)*
deforestation = Entwaldung: *Datenstrukturen sind Bäume. Das Verarbeiten von Datenstrukturen erzeut wiederum neue Datenstrukturen. d.h. Eine Folge von Transformationen eines Baums in einen anderen. Diese Bäume werden auf- und abgebaut. Ziel: Eingabebaum wird konsumiert und* **nur** *der Ausgabebaum wird erzeugt. Alle Zwischenbäume fallen weg.*
$f_1(f_2(f_3 list))$

$\Longrightarrow$ Entdecke bzw. erzeuge Schleifen mit Basis-Blöcken :-)

- Endrekursion
- *Schleifen sind vorhanden, wenn auch versteckt*
- Inlining
- **let**-Floating

Wende dann allgemeine Optimierungs-Techniken an!

... etwa durch Übersetzung nach C ;-)

Compiler für funktionale Sprachen arbeiten i.A. ohne Optimierung, sondern übersetzen in eine imperative Programmiersprache. Optimierungen werden dann durch den Compiler für diese imperative Programmiersprache durchgeführt.
Man folgt also der Strategie Schleifen zu erzeugen und Basisblöcke zu verlängern und dann zu optimieren.

Achtung:

Wir benötigen neue Programmanalyse-Techniken, um Informationen über funktionale Programme zu sammeln.

Man möchte nun wissen, welche Werte Variable haben können.

Beispiel: Inlining

$$\begin{array}{lcl} \textbf{fun}\ \mathsf{max}\,(x,y) & = & \textbf{if}\ x > y\ \textbf{then}\ x \\ & & \textbf{else}\ y \\ \textbf{fun}\ \mathsf{abs}\ z & = & \mathsf{max}\,(z,-z) \end{array}$$

Klassischer Fall von Inlining : Man möchte nun max in abs einsetzen.

Als Ergebnis der Optimierung erwarten wir ...

$$\begin{array}{lcll} \textbf{fun}\ \mathsf{max}\,(x,y) & = & \textbf{if}\ x > y\ \textbf{then}\ x & \\ & & \textbf{else}\ y & \\ \textbf{fun}\ \mathsf{abs}\ z & = & \textbf{let} & \textbf{val}\ x = z \\ & & & \textbf{val}\ y = -z \\ & & \textbf{in} & \boxed{\begin{array}{l}\textbf{if}\ x > y\ \textbf{then}\ x \\ \textbf{else}\ y\end{array}} \\ & & \textbf{end} & \end{array}$$

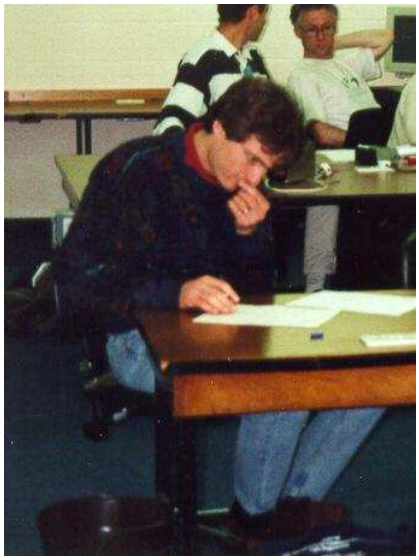
Diskussion:

max ist zuerstmal nur ein Name.

Wir müssen nun überprüfen welche Funktion ist nun gemeint

Wir müssen herausfinden, welchen Wert er zur Laufzeit haben kann

$\Longrightarrow$ Wert-Analyse erforderlich !!



Nevin Heintze im australischen Team
des Prolog-Programmier-Wettbewerbs, 1998

Das ganze Bild:

Um nun die Programmanalyse von Nevin Heintze beschreiben zu können, nehmen wir :

4.1 Eine einfache Zwischensprache

Zur Vereinfachung betrachten wir:

$$
\begin{array}{lcl}
v & ::= & b \mid (x_1, \ldots, x_k) \mid c\ x \mid \mathbf{fn}\ x \Rightarrow e \\
e & ::= & v \mid (x_1\ x_2) \mid (\square_1\ x) \mid (x_1\ \square_2\ x_2) \mid \\
 & & \mathbf{let}\ x_1 = e_1\ \ldots\ x_k = e_k\ \mathbf{in}\ e_0\ \mathbf{end} \mid \\
 & & \mathbf{letrec}\ x_1 = e_1\ \ldots\ x_k = e_k\ \mathbf{in}\ e_0\ \mathbf{end} \mid \\
 & & \mathbf{case}\ x\ \mathbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k\ \mathbf{end} \\
p & ::= & b \mid x \mid c\ x \mid (x_1, \ldots, x_k)
\end{array}
$$

letrec dient zur Definition von rekursiven Funktionen.
case führt das patternmatching auf das x durch.
Es gibt keine if-Anweisung, da es den Datentyp boolean gibt .(Zwei-Konstruktor Datentyp)
Die Auswertung geschieht mit case.

wobei b eine Konstante ist, x eine Variable, c ein (Daten-)Konstruktor und $\square_i$ i-stellige Operatoren sind.

Diskussion:

- Konstruktoren und Funktionen sind stets ein-stellig.
  Dafür gibt es explizite Tupel :-)
- **if**-Ausdrücke und Fall-Unterscheidung in Funktions- Definitionen wird auf **case**-Ausdrücke zurückgeführt.
- In Fall-Unterscheidungen sind nur einfache Muster erlaubt.
  $\Longrightarrow$ Komplizierte Muster müssen zerlegt werden ...
- **let**-Definitionen entsprechen Basis-Blöcken :-)
- Typ-Annotationen an Variablen, Mustern oder Ausdrücken könnten weitere nützliche Informationen enthalten
  — wir verzichten aber drauf :-)

... im Beispiel:

Die Definition von max sieht dann so aus:

$$\begin{array}{lll} \mathsf{max} = \mathbf{fn}\ x \Rightarrow & \mathbf{case}\ x\ \mathbf{of}\ (x_1, x_2)\ : & \\ & \quad \mathbf{let}\ z = x_1 < x_2 & \\ & \quad \mathbf{in}\ \ \mathbf{case}\ z & \\ & \qquad \mathbf{of}\ \ \mathsf{True}\ :\ \ x_2 & \\ & \qquad \mid\ \ \mathsf{False}\ :\ \ x_1 & \\ & \quad \mathbf{end} & \\ & \mathbf{end} & \\ \mathbf{end} & & \end{array}$$

Entsprechend haben wir für abs :

$$\begin{array}{lll} \mathsf{abs} = \mathbf{fn}\ x \Rightarrow & \mathbf{let}\ z_1 = -x & \\ & \quad z_2 = (x, z_1) & \\ & \mathbf{in}\ (\mathsf{max}\ z_2) & \\ & \mathbf{end} & \end{array}$$

4.2 Eine einfache Wert-Analyse

Idee:

Für jeden Teilausdruck e sammeln wir die Menge $[\![e]\!]^\sharp$ der möglichen Werte von e ...

Sei V die Menge der vorkommenden Konstanten (-Klassen), Konstruktor-Anwendungen und Funktionen. Dann wählen wir als vollständigen Verband natürlich:

$$\mathbb{V} = 2^V$$

Wir stellen wir ein Ungleichungs-System auf:

- Ist e ein Wert d.h. von der Form: $b, c\,x, (x_1, \ldots, x_k)$ oder $\mathbf{fn}\,x \Rightarrow e$ erzeugen wir:
$$[\![e]\!]^\sharp \supseteq \{e\}$$
- Ist $e \equiv (x_1\ x_2)$ und $f \equiv \mathbf{fn}\ x \Rightarrow e_1$, dann
$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (f \in [\![x_1]\!]^\sharp)\,?\,[\![e_1]\!]^\sharp\ :\ \emptyset \\ [\![x]\!]^\sharp & \supseteq & (f \in [\![x_1]\!]^\sharp)\,?\,[\![x_2]\!]^\sharp\ :\ \emptyset \end{array}$$

...

- int-Werte, die Operatoren zurück liefern, approximieren wir z.B. durch eine Konstante int.

  Operatoren, die Boolesche Werte liefern, liefern z.B. $\{\mathsf{True}, \mathsf{False}\}$:-)
- Ist $e \equiv \textbf{let } x_1 = e_1 \ldots x_k = e_k \textbf{ in } e_0 \textbf{ end}$. Dann erzeugen wir:

$$\begin{array}{lcl} [\![x_i]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

- Analog für $e \equiv \textbf{letrec } x_1 = e_1 \ldots x_k = e_k \textbf{ in } e_0 \textbf{ end}$:

$$\begin{array}{lcl} [\![x_i]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

- Sei $e \equiv \textbf{case } x \textbf{ of } p_1 \,:\, e_1 \mid \ldots \mid p_k \,:\, e_k \textbf{ end}$.
  Dann erzeugen wir für $p_i \equiv b$,

$$[\![e]\!]^\sharp \supseteq (b \in [\![x]\!]^\sharp)\,?\,[\![e_i]\!]^\sharp \,:\, \emptyset$$

  Ist $p_i \equiv c\,y$ und $v \equiv c\,z$ ein Wert, dann

$$\begin{array}{lcl} [\![e]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp)\,?\,[\![e_i]\!]^\sharp \,:\, \emptyset \\ [\![y]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp)\,?\,[\![z]\!]^\sharp \,:\, \emptyset \end{array}$$

  Ist $p_i \equiv (y_1, \ldots, y_k)$ und $v \equiv (z_1, \ldots, z_k)$ ein Wert, dann

$$\begin{array}{lcl} [\![e]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp)\,?\,[\![e_i]\!]^\sharp \,:\, \emptyset \\ [\![y_j]\!]^\sharp & \supseteq & (v \in [\![x]\!]^\sharp)\,?\,[\![z_j]\!]^\sharp \,:\, \emptyset \end{array}$$

  Es fliesst also Information in zwei Richtungen:
  In die Variablen und in die Rückgabewerte.

  Ist $p_i \equiv y$, dann

$$\begin{array}{lcl} [\![e]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![y]\!]^\sharp & \supseteq & [\![x]\!]^\sharp \end{array}$$

Man erhält also eine Menge von Ungleichungen, die man lösen kann.
Dies sind monotone Funktionen über einen vollständigen Verband die mit Fixpunktbestimmung berechnet werden können.
Kosten dieser Analyse:
Ein quadratisches Ungleichungssystem. Höhe des Verbandes über die Menge aller vorkommenden Werte ist linear. Quadratisches Gleichungssystem über einen Verband linearer Höhe. Dies ergibt kubisch viele Auswertung rechter Seiten. dadurch erhält man einen Algorithmus n^4.
Algorithmus von Heintze : Dieser läuft in kubischer Zeit. Bei normalen Programmen läuft er fast linear.

4.3 Eine operationelle Semantik

Frage nach der Korrektheit:
Betrachtung der operationellen Semantik, mithilfe dieser wird die Korrektheit gezeigt.

Idee:

Wir konstruieren eine Big-Step operationelle Semantik, die Ausdrücke auswertet :-)

Big-Step : Die einzelnen Teilprogramme liefern einen Wert, daraus schliesst man :
Das gesamte Programm liefert einen Wert.
Big-Step : Der Nachweis, dass das Programm einen Wert liefert geschieht mit einem Baum.
Small-Step : Ausführungspfad. Schrittweise von einem Zustand zum nächsten.

Konfigurationen:

$$\begin{aligned} c &::= (e, env) \\ vc &::= (v, env) \end{aligned}$$

Wobei nun v sein könnte : $(::z, \{\, z \mapsto ((x_1, x_2), \{x_1 \mapsto 1,\ x_2 \mapsto [\,]\,\})\,\})$

$$env ::= \{x_1 \mapsto vc_1, \ldots\}$$

Werte sind Konfigurationen, in denen der Ausdruck von der Form: $b, c\,x, (x_1, \ldots, x_k)$ oder **fn** $x \Rightarrow e$ ist :-)

Umgebungen enthalten nur Werte :-))

Dies geschieht unter der Annahme einer Call-by-Value Programmiersprache. (OCAML, ML)
d.h. In den Umgebungen welche die Bindungen der Variablen enthalten, stehen immer Werte.
Werte sind also Konstanten, Konstruktoranwendungen, Tupel oder Funktionen

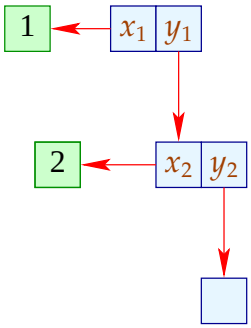
Beispiele für Werte:

$$\begin{aligned} 1 &: (1, \emptyset) \\ c\,1 &: (c\ x, \{x \mapsto (1, \emptyset)\}) \\ (1, (2, ())) \equiv [1,2] &: ((x_1, y_1), \{x_1 \mapsto 1, \\ &\qquad y_1 \mapsto ((x_2, y_2), \{x_2 \mapsto 2, \\ &\qquad\quad y_2 \mapsto ((), \emptyset)\})\}) \end{aligned}$$

Werte sehen etwas merkwürdig aus :-)

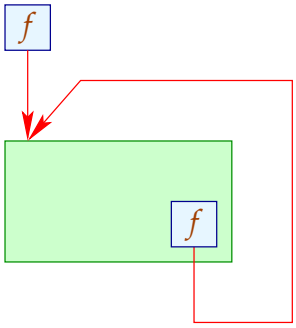
Der Grund ist, dass wir Substitutionen nie ausführen :-)
Alternativ können wir uns die Variablen in den Umgebungen als Speicherzellen vorstellen ...

$(1,\ (2,())) \equiv$



Achtung:

Rekursive Funktionen führen zu zyklischen Verweis-Strukturen ;-)



Auswege:

- Rekursive Funktionen werden auf dem Toplevel definiert :-)
  Auf deren globalen Variablen (Top-Level-Variablen) kann man somit **immer** *zugreifen.*
  Es gibt somit keine zyklischen Strukturen
- Lokale Rekursive Funktionen sind stets nur selbst rekursiv.
  Für diese führen wir einen neuen Operator fix ein ... *Zweistellig*

Aus:
letrec $x_1 = e_1$ **in** e_0 **end**
wird:
(In e_1 wird x_1 rückwärts an sich selbst gebunden)
let $x_1 = \mathsf{fix}\,(x_1, e_1)$ **in** e_0 **end**

Beispiel: Die append-Funktion

Betrachten wir die Konkatenation von zwei Listen. In ML schreiben wir einfach:

$$\begin{array}{lllll} \textbf{fun} & \mathsf{app} & [\,] & = & \textbf{fn}\ y\ \Rightarrow\ y \\ \mid & \mathsf{app} & (x :: xs) & = & \textbf{fn}\ y\ \Rightarrow\ x :: \mathsf{app}\ xs\ y \end{array}$$

In unserer eingeschränkten Zwischensprache sieht das etwas detaillierter aus :-)

$$\begin{array}{l} \mathsf{app} \;=\; \mathsf{fix}\,(\mathsf{app},\ \textbf{fn}\ x \Rightarrow \textbf{case}\ x \\ \qquad\qquad \textbf{of}\ \ [\,] : \ \ \textbf{fn}\ y \Rightarrow y \\ \qquad\qquad \mid\ \ :: z : \ \ \textbf{case}\ z\ \textbf{of}\ (x_1, x_2)\ :\ \ \textbf{fn}\ y \Rightarrow \\ \qquad\qquad\qquad\qquad \textbf{let}\ \ a_1 = \mathsf{app}\ x_2 \\ \qquad\qquad\qquad\qquad\quad\ \ a_2 = a_1\ y \\ \qquad\qquad\qquad\qquad\quad\ \ z_1 = (x_1, a_2) \\ \qquad\qquad\qquad\qquad \textbf{in}\ :: z_1 \\ \qquad\qquad\qquad\qquad \textbf{end} \\ \qquad\qquad\qquad\ \ \textbf{end} \\ \qquad\qquad\ \ \textbf{end}\ ) \end{array}$$

Die Big-Step Semantik gibt Regeln an, zu welchem Wert sich eine Konfiguration ausrechnen lässt ...

Funktionsanwendung:

$$\frac{\eta\, x_1 \;=\; (\mathbf{fn}\; x \;\Rightarrow\; e, \eta_1) \quad \eta\, x_2 \;=\; (v_2, \eta_2) \quad (e, \eta_1 \oplus \{x \mapsto (v_2, \eta_2)\}) \Longrightarrow (v_3, \eta_3)}{(x_1\; x_2, \eta) \Longrightarrow (v_3, \eta_3)}$$

Lokal rekursive Funktionsanwendung:

$$\frac{\eta\, x_1 \;=\; (\mathsf{fix}\,(f, \mathbf{fn}\; x \;\Rightarrow\; e), \eta_1) \quad \eta\, x_2 \;=\; (v_2, \eta_2) \quad (e, \eta_1 \oplus \{f \mapsto (\mathsf{fix}\,(f, \mathbf{fn}\; x \;\Rightarrow\; e), \eta_1),\; x \mapsto (v_2, \eta_2)\}) \Longrightarrow (v_3, \eta_3)}{(x_1\; x_2, \eta) \Longrightarrow (v_3, \eta_3)}$$

Fall-Unterscheidung 1:

$$\frac{\eta\, x \;=\; (b, \eta_1) \quad (e_i, \eta) \Longrightarrow (v_i, \eta_i)}{(\mathbf{case}\; x\; \mathbf{of}\; p_1 \;:\; e_1 \mid \ldots \mid p_k \;:\; e_k\; \mathbf{end}, \eta) \Longrightarrow (v_i, \eta_i)}$$

sofern $p_i \equiv b$ das erste auf b passende Muster ist :-)

Fall-Unterscheidung 2:

$$\frac{\eta\, x \;=\; (c\, z, \eta_1) \quad (e_i, \eta \oplus \{x_i \mapsto (\eta\, z)\}) \Longrightarrow (v_i, \eta_i)}{(\mathbf{case}\; x\; \mathbf{of}\; p_1 \;:\; e_1 \mid \ldots \mid p_k \;:\; e_k\; \mathbf{end}, \eta) \Longrightarrow (v_i, \eta_i)}$$

sofern $p_i \equiv c\; x_i$ das erste auf $c\, z$ passende Muster ist :-)

Fall-Unterscheidung 3:

$$\frac{\eta\, x \;=\; ((z_1, \ldots, z_m), \eta_1) \quad (e_i, \eta \oplus \{y_j \mapsto (\eta\, z_j) \mid j = 1, \ldots, m\}) \Longrightarrow (v_i, \eta_i)}{(\mathbf{case}\; x\; \mathbf{of}\; p_1 \;:\; e_1 \mid \ldots \mid p_k \;:\; e_k\; \mathbf{end}, \eta) \Longrightarrow (v_i, \eta_i)}$$

für das erste passende Muster $p_i \equiv (y_1, \ldots, y_m)$:-)

Fall-Unterscheidung 4:

$$\frac{}{(\textbf{case } x \textbf{ of } p_1 \,:\, e_1 \mid \ldots \mid p_k \,:\, e_k \textbf{ end}, \eta) \Longrightarrow (v_i, \eta_i)}$$

sofern $p_i \equiv x_i$ und alle Muster davor fehl schlugen :-)

Lokale Definitionen:

$$\frac{\begin{array}{l}(e_1, \eta) \Longrightarrow (v_1, \eta_1) \\ (e_2, \eta \oplus \{x_1 \mapsto (v_1, \eta_1)\}) \Longrightarrow (v_2, \eta_2) \\ \ldots \\ (e_k, \eta \oplus \{x_1 \mapsto (v_1, \eta_1), \ldots, x_{k-1} \mapsto (v_{k-1}, \eta_{k-1})\}) \Longrightarrow (v_k, \eta_k) \\ (e_0, \eta \oplus \{x_1 \mapsto (v_1, \eta_1), \ldots, x_k \mapsto (v_k, \eta_k)\}) \Longrightarrow (v_0, \eta_0)\end{array}}{(\textbf{let } x_1 = e_1 \ldots x_k = e_k \textbf{ in } e_0 \textbf{ end}, \eta) \Longrightarrow (v_0, \eta_0)}$$

Variablen:

Bemerkung:
Fixpunkte einstelliger Funktionen genügen. Bei mehreren Funktionen können Tupel gebildet werden und davon der Fixpunkt berechnet werden.

$$\frac{\eta\,(x) \;=\; (v_1, \eta_1)}{(x, \eta) \Longrightarrow (v_1, \eta_1)}$$

Konstanten:

$$(cv, \eta) \Longrightarrow (cv, \eta)$$

Korrektheit der Analyse:

Man zeigt für jedes (e, η) , das in einer Ableitung für das Programm vorkommt:

- Falls $\eta\,(x) = (v, \eta_1)$, dann ist $v \in [\![x]\!]^\sharp$.
- Falls $(e, \eta) \Longrightarrow (v, \eta_1)$, dann ist $v \in [\![e]\!]^\sharp$.

Fazit:

$[\![e]\!]^\sharp$ liefert eine Obermenge der Werte, zu denen sich e möglicherweise ausrechnet :-)
Technisch zeigt man die Korrektheit, indem man eine Induktion über die Länge der Ableitungen durch-

führt.

4.4 Anwendung: Inlining

Man versucht durch inlining möglichst grosse basic-blocks (eben lange Folgen von Anweisungen) zu erzeugen.

Probleme:

- globale Variablen.
  Durch Verschiebung von Code kann es mit globalen Variablen natürlich Probleme ergeben. (Dynamische und statische Bindung(λ-Kalkül!))
  Das Programm:

$$
\begin{array}{lll}
\textbf{let} & x = 1 & \\
 & f = & \textbf{let} \quad x = 2 \\
 & & \textbf{in} \quad \textbf{fn}\ y \ \Rightarrow \ y + x \\
 & & \textbf{end} \\
\textbf{in} & f\ x & \\
\textbf{end} & &
\end{array}
$$

- ... berechnet offenbar etwas anderes als:

$$
\begin{array}{lll}
\textbf{let} & x = 1 & \\
 & f = & \textbf{let} \quad x = 2 \\
 & & \textbf{in} \quad \textbf{fn}\ y \ \Rightarrow \ y + x \\
 & & \textbf{end} \\
\textbf{in} & \boxed{\begin{array}{ll} \textbf{let} & y = x \\ \textbf{in} & y + x \\ \textbf{end} & \end{array}} & \\
\textbf{end} & &
\end{array}
$$

- rekursive Funktionen. In der Definition:

$$x = \mathsf{fix}\ (\mathsf{foo},\ \textbf{fn}\ y \ \Rightarrow \ \mathsf{foo}\ y)$$

  sollten wir foo besser nicht substituieren :-)

Idee 1:

→ Wir machen erstmal die Namen im Programm eindeutig.

→ Dann substituieren wir nur Funktionen, die statisch im Scope der selben globalen Variablen stehn, wie die Anwendung :-)

→ Wir berechnen für jeden Ausdruck alle Funktions-Definitionen mit dieser Eigenschaft :-)

Sei $D[e]$ die Menge der Definitionen, die in e statisch ankommen.

•• Für $e \equiv \textbf{let } x_1 = e_1 \ldots x_k = e_k \textbf{ in } e_0 \textbf{ end } D = D[e]$ haben wir:

$$\begin{aligned} D[e_1] &= D \\ &\ldots \\ D[e_k] &= D \cup \{x_1, \ldots, x_{k-1}\} \\ D[e_0] &= D \cup \{x_1, \ldots, x_k\} \end{aligned}$$

•• In den anderen Fällen propagiert sich D unverändert zu den Teilausdrücken :-)
Für $e \equiv \textbf{fn } x \Rightarrow e_1$ haben wir etwa:

$$D[e_1] = D$$

... im Beispiel:

$$\begin{array}{lll} \textbf{let} & x = 1 & \\ & f = & \textbf{let} \quad x_1 = 2 \\ & & \textbf{in} \quad \textbf{fn } y \Rightarrow y + x_1 \\ & & \textbf{end} \\ \textbf{in} & f\ x & \\ \textbf{end} & & \end{array}$$

Bemerkung : let-floating : Die Definition der globalen Variablen muss lokal bereitgestellt werden. Die Funktionsdefinition wird mit dem umfassenden let getauscht. Die Definiton von Variablen wird ver-

schoben.

... steht (nach Umbenennung :-) f für $f\ x$ statisch zur Verfügung.
Bei Substitution erhalten wir:

$$
\begin{array}{ll}
\textbf{let} & x = 1 \\
 & f = \ \textbf{let} \quad x_1 = 2 \\
 & \quad\ \ \textbf{in} \quad \textbf{fn}\ y \ \Rightarrow\ y + x_1 \\
 & \quad\ \ \textbf{end} \\
\textbf{in} & \boxed{\begin{array}{ll}
\textbf{let} & y = x \\
\textbf{in} & \ \ \textbf{let} \quad x_1 = 2 \\
 & \ \ \textbf{in} \quad y + x_1 \\
 & \ \ \textbf{end} \\
\textbf{end} &
\end{array}} \\
\textbf{end} &
\end{array}
$$

Ersetzen der Variablen-Variablen-Umbenennungen ergibt schließlich:

$$
\begin{array}{ll}
\textbf{let} & x = 1 \\
 & f = \ \textbf{let} \quad x_1 = 2 \\
 & \quad\ \ \textbf{in} \quad \textbf{fn}\ y \ \Rightarrow\ y + x_1 \\
 & \quad\ \ \textbf{end} \\
\textbf{in} & \boxed{\begin{array}{ll}
\textbf{let} & x_1 = 2 \\
\textbf{in} & x + x_1 \\
\textbf{end} &
\end{array}} \\
\textbf{end} &
\end{array}
$$

Idee 2:

→ Wir benutzen unsere Wert-Analyse.
Eine Wertanalyse für eine Funktion die über mehrere Etappen an Variablen gebunden und als Parameter weitergereicht wird.

→ Wir ignorieren globale Variablen :-)

→ Wir substituieren nur Funktionen ohne freie Variablen :-))

Bemerkung : λ-lifting: (Beseitigung von globalen Variablen)

$$
\begin{array}{ll}
\textbf{let} & x_1 = 2 \\
\textbf{in} & \textbf{f}\ y\ \Rightarrow\ y + x_1 \\
\textbf{end} &
\end{array}
$$

wird ersetzt durch:

$$
\begin{array}{ll}
\textbf{let} & \textbf{f}' x_1 y\ \Rightarrow\ y + x_1 \\
 & x_1 = 2 \\
\textbf{in} & \textbf{f}\ \Rightarrow\ f'\ x_1 \\
\textbf{end} &
\end{array}
$$

Beispiel: Die map-Funktion

$$
\begin{array}{l}
\textbf{let}\ f = \textbf{fn}\ x\ \Rightarrow\ x \cdot x \\
\quad \mathsf{map} = \mathsf{fix}\,(\mathsf{map},\ \textbf{fn}\ g\ \Rightarrow\ \textbf{fn}\ x\ \Rightarrow\ \textbf{case}\ x \\
\qquad\qquad\qquad\qquad \textbf{of}\ \ [\,]\ :\ \ [\,] \\
\qquad\qquad\qquad\qquad \mid\ \ ::\, z :\ \ \textbf{case}\ z\ \textbf{of}\ (x_1, x_2)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let}\ y_1 = g\ x_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad m = \mathsf{map}\ g \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad y_2 = m\ x_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad z_1 = (y_1, y_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ ::\, z_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{end} \\
\qquad\qquad\qquad\qquad \textbf{end}) \\
\quad h = \mathsf{map}\ \ f \\
\textbf{in}\ \ h\ list \\
\textbf{end}
\end{array}
$$

- Der formale Parameter g von map ist stets f :-)
- Wir können die Anwendung von f in der Definition von map ersetzen:

```
map = fix (map, fn g ⇒ fn x ⇒ case x
                of [] :  []
                 | ::z :  case z of (x1, x2) in
                          let y1 = let x = x1
                                   in x · x
                                   end
                              m = map g
                              y2 = m x2
                              z1 = (y1, y2)
                          in ::z1
                          end
          end)
h = map f
```

- Noch mehr könnten wir sparen, wenn wir die spezialisierte Funktion $h = \mathsf{map}\ f$ direkt definieren könnten :-)
  Man spezialisiert diese map-funktion im Hinblick auf den konstanten Parameter zu einer Funktion die sofort quadriert.
  Vorteil : Man spart das Übergeben des formalen Parameters
- Dazu müssen wir überall in der Definition von map das Muster $\boxed{\mathsf{map}\ g}$ durch h ersetzen ...

  $\Longrightarrow$ fold-Transformation :-)

- Alle weiteren Vorkommen von g müssen durch (die Definition von) f ersetzt werden ...
  // kommt hier nicht vor :-)

```
map = fix (map, fn g ⇒ fn x ⇒ case x
                 of [] : []
                  | ::z : case z of (x1, x2) in
                          let y1 = let x = x1
                                   in x · x
                                   end
                              m = map g
                              y2 = m x2
                              z1 = (y1, y2)
                          in ::z1
                          end
                 end)
h = map f

h = fix (h, fn x ⇒ case x
                 of [] : []
                  | ::z : case z of (x1, x2) in
                          let y1 = let x = x1
                                   in x · x
                                   end
                              m = h
                              y2 = m x2
                              z1 = (y1, y2)
                          in ::z1
                          end
                 end)
```

Beseitigung von Variablen-Variablen-Umspeicherungen liefert:

$$
\begin{array}{lll}
h = \mathsf{fix}\,(h, \textbf{fn}\ x & \Rightarrow & \textbf{case}\ x \\
\quad \textbf{of}\ \ [\,] & : & [\,] \\
\quad \mid\ \ :: z & : & \textbf{case}\ z\ \textbf{of}\ (x_1, x_2)\ \textbf{in} \\
 & & \quad \textbf{let}\ y_1 = x_1 \cdot x_1 \\
 & & \qquad\ \ y_2 = h\ x_2 \\
 & & \qquad\ \ z_1 = (y_1, y_2) \\
 & & \quad \textbf{in}\ \ :: z_1 \\
 & & \quad \textbf{end} \\
\textbf{end})
\end{array}
$$

Insbesondere bietet die Beseitigung von Zwischendatenstrukturen interessante Ansätze und Konstruktionen die bei der Optimierung von funktionalen Sprachen eingesetzt werden.

Weitere Gesetze:
map f (map g list)
ist äquivalent zu
map (f ∘ g) list
Diese algebraischen Gesetze werden benutzt um das Programm zu transformieren.
Man sucht im Programm solche Vorkommen dieser Teilausdrücke und ersetzt sie.
Die zweite Funktion läuft eben nur einmal durch die Liste und ist effizienter.

Dementsprechend für **fold**
foldr g a (map f list)
(foldr g a ersetzt das NIL durch das a und die Konstruktoren durch das g)
Auch hier wird die Liste zweimal durchlaufen.
Dies wird ersetzt durch:
let g'(x,y) = g(fx,y)
in foldr g' a list
end
Nochmals die Definition des foldr:
foldr g a [] = a
foldr g a (x,y) = g (x, foldr g a y)