Vorlesungsnotizen

Programmanalyse

Roland Meyer
Sebastian Wolff

Letzte Änderung: 9. Januar 2018

TCS Institute

Technische Universität Braunschweig

Vorwort

Dies sind Notizen zur Vorlesung *Programmanalyse*, die von uns im Wintersemester 2017/2018 an der TU Braunschweig gehalten wird.

Die Vorlesung basiert auf den folgenden Veranstaltungen:

- Vorlesung *Bäume, Ordnungen, und Anwendungen*, gehalten von Roland Meyer im Wintersemester 2015/2016 an der TU Kaiserslautern.
- Vorlesung *Formale Grundlagen der Programmierung*, gehalten von Roland Meyer im Sommersemester 2016 an der TU Kaiserslautern.

Dieses Skript basiert auf den von Roland Meyer erstellten handschriftlichen Vorlesungsausarbeitung, sowie auf dem von Jonathan Kolberg und Sebastian Muskalla geTEXten Skript zur Vorlesung *Bäume, Ordnungen, und Anwendungen*.

Dieses Dokument ist noch in Bearbeitung und wir geben keinerlei Garantie auf Vollständigkeit oder Korrektheit. Wir bitten darum, Fehler per E-Mail zu melden: sebastian.wolff@tu-bs.de .

Sebastian Wolff, Roland Meyer
Braunschweig, 9. Januar 2018

Inhaltsverzeichnis

1 Verbände und Fixpunke

In diesem Abschnitt studieren wir die formalen Grundlagen der statischen Programmanalyse: Verbände und Fixpunkte. Wir werden im Laufe der Vorlesung verschiedene Verfahren diskutieren, die Fixpunkte auf solchen Verbänden ausführen. Warum diese Verfahren auf Verbände zurückgreifen, wollen wir in folgendem Beispiel motivieren. Anschließend folgt eine formale Betrachtung.

1.1 Verbände in der Programmanalyse

Es folgt ein motivierendes Beispiel, welches ein grundlegendes Problem der statischen Programmanalyse aufdeckt: Berechenbarkeit. Wir skizzieren, wie wir trotz dieser Hürde korrekte Analysen durchführen können.

Ziel: Eine typische Aufgabe bei der Programmanalyse ist es, die Menge der Zustände zu ermitteln, die an einem Programmpunkt eingenommen werden können. Diese Zustände ergeben sich gegebenenfalls auf Grund verschiedener Ausführungen. Diese Aufgabe ist insofern relevant, als dass der Zustand eines Systems Aufschluss über sein Verhalten gibt.

Ansatz: Betrachte alle Ausführungen und bilde die Vereinigung über die Zustände, die an einem Punkt erreicht werden.

Beispiel 1.1

Betrachte folgendes Programm:

```
1 p := 5;
2 q := 2;
3 while (p > q) {
4     p := p + 1;
5     q := q + 2;
6 }
7 print q;
```

Es gibt nur eine Ausführung des Programms. Diese erreicht folgende Zustände:

- in Punkt 5: $\{(6,2),(7,4),(8,6)\}$
- in Punkt 6: $\{(6,2),(7,4),(8,6),(8,8)\}$

Damit ist gezeigt, dass nur gerade Werte ausgegeben werden.

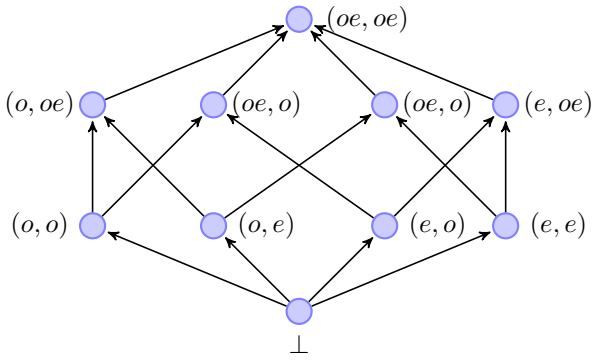
□

Problem: Vereiningung über alle Zustände ist nicht berechenbar (Satz von Rice).

Ansatz: *Abstraktion*

- Führe das Programm auf abstrakten Zuständen aus. Dazu *interpretiere* die Befehle in der abstrakten Domäne. Ziel ist es, das gewünschte Resultat in der abstrakten Domäne auszurechnen.
- Die konkreten Zustände an einem Punkt werden durch die abstrakten Zustände an diesem Punkt darstellt.
- Ein abstrakter Zustand beschreibt typischerweise mehrere (unendlich viele) konkrete Zustände.
- Bilde den Join ($\sqcup$) der abstrakten Zustände *Join-over-all-paths* (JOP) (in der Literatur auch *Meet-over-all-paths*)
- Falls die abstrakten Zustände einen *vollständigen Verband* bilden, existiert der Join.

Beispiel 1.2 (Fortsetzung)
Vollständiger Verband der abstrakten Werte:



Dabei repräsentiert der abstrakte Zustand (o, oe) *alle* konkreten Zustände mit

- `p` hat einen ungeraden Werte (*odd*),
- `q` hat irgendeinen Wert (*odd* oder *even*).

Der Join über alle abstrakten Ausführung, die zu Punkt 5 führen, ist:

$$\begin{aligned} \bot \sqcup (e, e) &= (e, e) \\ (e, e) \sqcup (o, e) &= (oe, e) \\ (oe, e) \sqcup (oe, e) &= (oe, e) \end{aligned}$$

Dementsrechend erhalten wir für Punkt 6 den abstrakten Wert (oe, e). Damit haben wir also ebenso einen Nachweis erbracht, dass lediglich gerade Werte ausgegeben werden. □

Warum benötigen wir Fixpunkte?

- Anstelle des JOP, berechne Fixpunkt von Funktionen auf dem Verband.
- Unter weiteren Annahmen ist garantiert, dass der Fixpunkt JOP überapproximiert.
- Satz von Knaster-Tarski sagt aus, wann Fixpunkte existieren, und in diesem Fall können sie mit Kleene-Iteration berechnet werden.

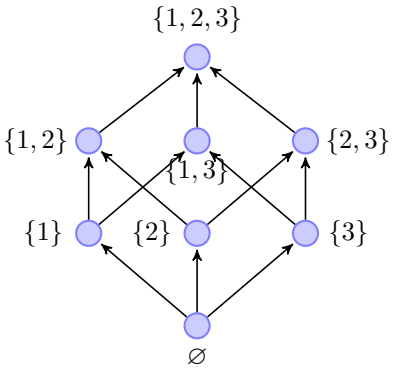
1.2 Partielle Ordnungen und Verbände

Beobachtung

- $(\mathbb{N}, \leq)$ ist total geordnet: jeweils zwei Elemente sind in der Ordnung vergleichbar
- Einige Domänen sind nur partiell geordnet □

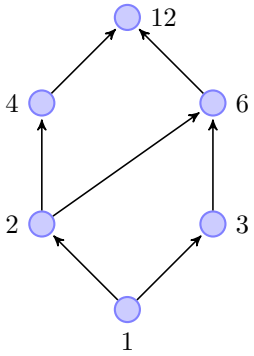
Beispiel 1.3 (Teilmengen von $\{1,2,3\}$, Teiler von 12)

Teilmengen von $\{1,2,3\}$ bezüglich $\subseteq$



$\{1,2\}$ und $\{2,3\}$ sind unvergleichbar.

Teiler von 12 bezüglich $|$ (Teilbarkeit)



2 und 3 sind unvergleichbar.

□

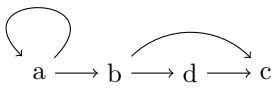
Definition 1.4 (Partielle Ordnung)

Eine *partielle Ordnung* $(D, \leq)$ besteht aus einer Menge $D \neq \varnothing$ und einer Relation $\leq \subseteq D \times D$ mit folgenden Eigenschaften

- reflexiv: $\forall d \in D : d \leq d$
- transitiv: $\forall d_1, d_2, d_3 \in D : d_1 \leq d_2 \wedge d_2 \leq d_3 \Rightarrow d_1 \leq d_3$
- antisymmetrisch: $\forall d_1, d_2 \in D : d_1 \leq d_2 \wedge d_2 \leq d_1 \Rightarrow d_1 = d_2$ □

Binäre Relationen lassen sich als *gerichtete Graphen* auffassen, z.B.

$$\{(a,a),(a,b),(b,c),(b,d),(d,c)\} =$$

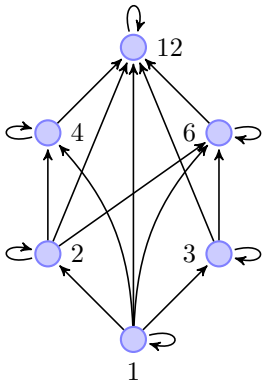


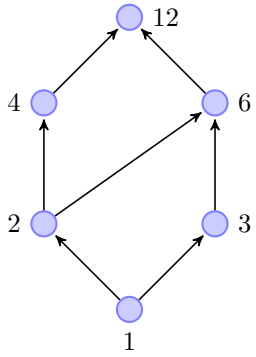
Partielle Ordnungen liefern besondere Graphen:

- Reflexivität $\rightsquigarrow$ Schleifen an Knoten
- Antisymmetrie $\rightsquigarrow$ keine nicht-trivialen Kreise
- Transitivität $\rightsquigarrow$ Transitivität der Kanten

Beispiel 1.5 (Teiler von 12)

Das Hasse-Diagramm (rechts) lässt Schleifen und induzierte Kanten des gerichteten Graphen (links) weg.





□

Definition 1.6 (Join und Meet)

Sei $(D, \leq)$ eine partielle Ordnung und $X \subseteq D$ eine Menge.

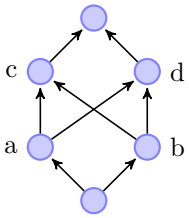
- Ein Element $o \in D$ heißt *obere Schranke* von X falls $x \leq o$ für alle $x \in X$.
- Ein Element $o \in D$ heißt *kleinste obere Schranke* bzw. *Join* von X falls
  - o ist obere Schranke und
  - $o \leq o'$ für alle oberen Schranken o' von X.

  Wir schreiben den Join wie folgt: $o = \bigsqcup X$.
- Ein Element $u \in D$ heißt *untere Schranke* von X falls $u \leq x$ für alle $x \in X$.
- Ein Element $u \in D$ heißt *größte untere Schranke* bzw. *Meet* von X, falls
  - u ist obere Schranke und
  - $u' \leq u$ für alle unteren Schranken u' von X.

  Wir schreiben: $u = \bigsqcap X$. □

Aus der Definition folgt, dass Join und Meet eindeutig sind, falls sie existieren. Angenommen sowohl o als auch o' sind kleinste obere Schranken. Dann gilt nach der zweiten definierenden Eigenschaft $o \leq o'$ und $o' \leq o$. Mit Antisymmetrie folgt $o = o'$.

Beispiel 1.7



a und b haben

- c und d als obere Schranken
- aber keine kleinste obere Schranke

□

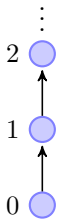
Definition 1.8 (Verband)

Ein *Verband* ist eine partielle Ordnung $(D, \leq)$ in der für jedes Paar $a, b \in D$ von Elementen Join $a \sqcup b$ und Meet $a \sqcap b$ existieren. Dabei ist $a \sqcup b$ Infixnotion für $\bigsqcup\{a, b\}$.

Ein Verband heißt *vollständig*, falls für jede Teilmenge $X \subseteq D$ von Elementen Join $\bigsqcup X$ und Meet $\bigsqcap X$ existieren. □

Beispiel 1.9

kein Verband



kein vollständiger Verband

□

Lemma 1.10

(1) Ein vollständiger Verband $(D, \leq)$ hat ein eindeutiges kleinstes Element (*Bottom*):

$$\bot := \bigsqcup \varnothing = \bigsqcap D$$

(2) Ein vollständiger Verband hat ein eindeutiges größtes Element (*Top*):

$$\top := \bigsqcap \varnothing = \bigsqcup D$$

(3) Jeder endliche Verband $(D, \leq)$ (mit D endlich) ist bereits vollständig. □

1.3 Monotone Funktionen und der Satz von Knaster&Tarski

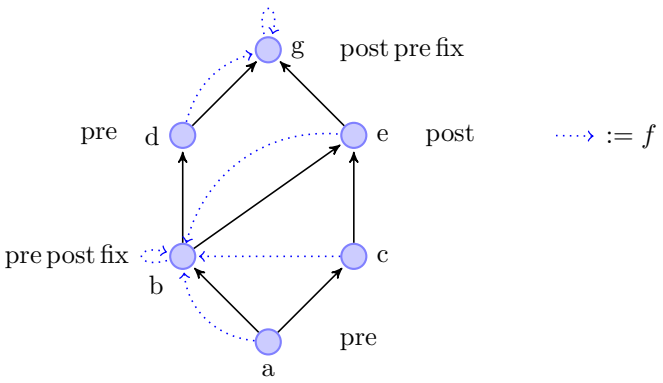
Definition 1.11 (Monotone Funktionen, Fixpunkte)
Sei $(D, \leq)$ eine partielle Ordnung.

- Eine Funktion $f : D \to D$ heißt *monoton*, falls für alle $x, y \in D$ gilt:

$$x \leq y \implies f(x) \leq f(y)$$

- Sei $f : D \to D$ eine Funktion auf $(D, \leq)$
  - Ein *Fixpunkt* von f ist ein Element $x \in D$ mit $f(x) = x$.
  - Ein *Pre-Fixpunkt* von f ist ein Element $x \in D$ mit $x \leq f(x)$.
  - Ein *Post-Fixpunkt* von f ist ein Element $x \in D$ mit $f(x) \leq x$. □

Beispiel 1.12



□

Satz 1.13 (Knaster&Tarski '55)
Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.
(1) Dann besitzt f einen (eindeutigen) *kleinsten* Fixpunkt, gegeben durch

$$\text{lfp}(f) := \bigsqcap \text{Postfix}(f)$$

(2) Ferner besitzt f einen (eindeutigen) *größten* Fixpunkt, gegeben durch

$$\text{gfp}(f) := \bigsqcup \text{Prefix}(f)$$

□

Beweis
Zeige die Behauptung für $\text{lfp}(f)$.
Dazu sei

$$l := \bigsqcap \text{Postfix}(f)$$

Zeige zunächst

$$f(l) \leq l$$

Da $l \leq l'$ für alle $l' \in \text{Postfix}(f)$ und da f monoton, folgt

$$f(l) \leq f(l') \leq l' \text{ für alle } l' \in \text{Postfix}(f)$$

Da $l = \bigsqcap \text{Postfix}(f)$ folgt

$$f(l) \leq l \qquad (*)$$

Zeige nun

$$l \leq f(l)$$

Mit $(*)$ gilt:

$$f(f(l)) \leq f(l)$$

Damit gilt

$$f(l) \in \text{Postfix}(f) \text{ und so } l \leq f(l) \qquad (*)$$

Mit Anti-Symmetrie folgt aus $(*)$ und $(*)$

$$l = f(l)$$

Damit ist gezeigt, dass l ein Fixpunkt ist. Beachte, dass jeder Fixpunkt von f auch ein Postfixpunkt ist und daher in $\text{Postfix}(f)$ enthalten ist. Da l als kleinste untere Schranke aller Postfixpunkte definiert war, ist l insbesondere kleiner als jeder andere Fixpunkt und damit der kleinste Fixpunkt.

Der Beweis für gfp geht analog. ■

1.4 Ketten und der Satz von Kleene

Sei $(D, \leq)$ eine partielle Ordnung.

- Eine total geordnete Teilmenge $K \subseteq D$ heißt *Kette* wenn sie total geordnet ist:

$$\forall k_1, k_2 \in K : k_1 \leq k_2 \text{ oder } k_2 \leq k_1$$

- Eine Folge $(k_i)_{i \in \mathbb{N}}$ heißt *aufsteigende Kette*, falls

$$k_i \leq k_{i+1} \text{ für alle } i \in \mathbb{N}$$

- Eine Folge $(k_i)_{i \in \mathbb{N}}$ heißt *absteigende Kette*, falls

$$k_i \geq k_{i+1} \text{ für alle } i \in \mathbb{N}$$

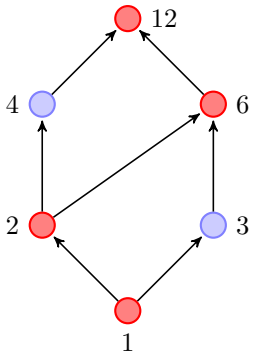
- Eine auf-/absteigende Kette $(k_i)_{i \in \mathbb{N}}$ wird *stationär*, falls

$$\exists n \in \mathbb{N} \colon \forall i \geq n \colon k_i = k_n$$

- $(D, \leq)$ hat *endliche Höhe*, falls jede Kette K in D endlich viele Elemente hat.
- $(D, \leq)$ hat *beschränkte Höhe*, falls es $n \in \mathbb{N}$ gibt, so dass jede Kette höchstens n Elemente hat.

Beispiel 1.14

1. In $(\mathbb{N}, \leq)$ wird jede absteigende Kette stationär.
2. Die rot markierten Knoten bilden eine Kette.



□

Definition 1.15 (Kettenbedingung)

Eine partielle Ordnung $(D, \leq)$

- erfüllt die *aufsteigende Kettenbedingung (ACC)*[1], falls jede aufsteigende Kette $k_0 \leq k_1 \leq \cdots$ stationär wird. (Man sagt auch $(D, \leq)$ ist *Artinsch*, nach Emil Artin.)
- erfüllt die *absteigende Kettenbedingung (DCC)*[2], falls jede absteigende Kette $k_0 \geq k_1 \geq \cdots$ stationär wird. (Man sagt auch $(D, \leq)$ ist *Noethersch*, nach Emmy Noether.) □

Beachte: ACC und DCC sind unabhängig von den Verbandsbedingungen.

Lemma 1.16

Eine partielle Ordnung hat endliche Höhe gdw. (ACC) und (DCC) erfüllt sind. □

[1] ascending chain condition
[2] descending chain condition

Definition 1.17 (Stetigkeit)
Sei $(D, \leq)$ ein vollständiger Verband. Eine Funktion $f : D \to D$ heißt

(1) $\sqcup$-*stetig* (aufwärtsstetig), falls für jede Kette K in D gilt

$$\begin{aligned} f\left(\bigsqcup K\right) &= \bigsqcup f(K) \\ &= \bigsqcup \{f(k) | k \in K\} \end{aligned}$$

(2) $\sqcap$-*stetig* (abwärtsstetig), falls für jede Kette K in D gilt

$$\begin{aligned} f\left(\bigsqcap K\right) &= \bigsqcap f(K) \\ &= \bigsqcap \{f(k) | k \in K\} \end{aligned}$$

□

Satz 1.18 (Monotonie impliziert Stetigkeit)
Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.

(1) Falls $(D, \leq)$ (ACC) erfüllt, dann ist f $\sqcup$-stetig.

(2) Falls $(D, \leq)$ (DCC) erfüllt, dann ist f $\sqcap$-stetig. □

Beweis
Wir zeigen (1). Der Beweis von (2) geht analog.
Sei K eine Kette in D. Es ist zu zu zeigen: $f(\bigsqcup K) = \bigsqcup f(K)$.

" $\leq$ " Für alle $k \in K$: $k \leq \bigsqcup K$.
Wegen Monotonie damit auch $f(k) \leq f(\bigsqcup K)$.
Da dies für alle k gilt, gilt auch $\bigsqcup f(K) \leq f(\bigsqcup K)$.

" $\geq$ " Wir zeigen zunächst, dass es in K ein größtes Element gibt, d.h. es existiert $k' \in K$, so dass für alle $k \in K$ gilt: $k \leq k'$.

Angenommen dies ist nicht der Fall, d.h. für alle k' gibt es ein $k'' \in K$, so dass k' und k'' unvergleichbar sind oder $k'' > k$ gilt. Da alle Elemente einer Kette vergleichbar sind, kann der erste Fall nie eintreten. Unter der Annahme, dass es zu jedem Element ein echt größeres gibt, können wir aber eine unendliche echt aufsteigende Kette konstruieren. Dies ist ein Widerspruch zur aufsteigenden Kettenbedingung (ACC).

Es gibt also ein größtes Element k' in der Kette. Damit gilt: $f(\bigsqcup K) = f(k') \leq \bigsqcup f(K)$. ■

Lemma 1.19
Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.

Die Folge

$$((f^i(\bot))_{i \in \mathbb{N}} \text{ mit } f^0(\bot) := \bot \text{ und } f^{i+1}(\bot) := f(f^i(\bot))$$

ist eine aufsteigende Kette. □

Beweis
Wir zeigen $f^i(\bot) \leq f^{i+1}(\bot)$ für alle $i \in \mathbb{N}$.

IA: $f^0(\bot) = \bot \leq f(\bot)$, da $\bot = \sqcap D$.

IV: Gelte $f^i(\bot) \leq f^{i+1}(\bot)$ für ein i.

IS: $f^{i+1}(\bot) \underset{\text{IV + Monotonie}}{\overset{=}{\leq}} f(f^i(\bot)) \leq f(f^{i+1}(\bot)) = f^{i+2}(\bot)$ ■

Satz 1.20 (Kleene)
Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.

(1) Ist f $\sqcup$-stetig, dann gilt: $\text{lfp}(f) = \sqcup\{f^i(\bot) | i \in \mathbb{N}\}$

(2) Ist f $\sqcap$-stetig, dann gilt: $\text{gfp}(f) = \sqcap\{f^i(\top) | i \in \mathbb{N}\}$ □

Beweis (Beweis von (1))
Zeige: $\bigsqcup\{f^i(\bot) | i \in \mathbb{N}\}$ ist Fixpunkt.

$$\begin{aligned} & f(\bigsqcup\{f^i(\bot) | i \in \mathbb{N}\} \\ (f \bigsqcup\text{-stetig}) \quad &= \quad \bigsqcup\{f^{i+1}(\bot) | i \in \mathbb{N}\} \\ (\bot = \sqcap D) \quad &= \quad \bigsqcup\{f^i(\bot) | i \in \mathbb{N}\} \end{aligned}$$

Zeige: $\bigsqcup\{f^i(\bot) | i \in \mathbb{N}\}$ ist kleinster Fixpunkt.

- Betrachte $d \in D$ mit $f(d) = d$ und zeige $\bigsqcup\{f^i(\bot) | i \in \mathbb{N}\}$ ist kleiner
- Induktion nach $i \in \mathbb{N}$ gibt $f^i(\bot) \leq d$ f.a. $i \in \mathbb{N}$.

  IA: $f^0(\bot) = \bot \leq d$, da $\bot = \sqcap D$

  IV: Angenommen $f^i(\bot) \leq d$ für ein i.

  IV: $\underline{i \to i+1}$

  $$f^{i+1}(\bot) = f(f^i(\bot)) \overset{\text{IV+Mon.}}{\leq} f(d) \overset{\text{Vor.}}{=} d$$

- Da $f^i(\bot) \leq d$ f.a. $i \in \mathbb{N}$ folgt

  $$\bigsqcup\{f^i(\bot) | i \in \mathbb{N}\} \leq d$$

Der Beweis der zweiten Aussage funktioniert analog. ■

Satz 1.21
Sei $(D, \leq)$ ein vollständiger Verband mit (ACC) und (DCC). Sei $f : D \to D$ monoton.
Dann ist

$$\begin{aligned} \text{lfp}(f) &= \bigsqcup\{f^i(\bot) | i \in \mathbb{N}\} \\ &= f^n(\bot) \qquad \text{mit } f^n(\bot) = f^{n+1}(\bot). \end{aligned}$$

□

$$\begin{aligned} \text{gfp}(f) &= \bigsqcap\{f^i(\top) | i \in \mathbb{N}\} \\ &= f^n(\top) \qquad \text{mit } f^n(\top) = f^{n+1}(\top). \end{aligned}$$

Beweis
Aus Monotonie folgt Stetigkeit wegen (ACC) und (DCC). Dann Satz von Knaster. ■

2 Intraprozedurale Datenflussanalyse

In diesem Abschnitt diskutieren wir Datenflussanalysen, die es erlauben, effizient Eigenschaften von Programmen nachzuweisen. Wir werden Analysen besprechen, wie sie in jedem Compiler zum Einsatz kommen.

Ziel: Analysiere das Verhalten von Programmen *statisch*, d.h. zur Compile-Zeit, nicht zur Laufzeit.

Ansatz: Fixpunktberechnung auf einer abstrakten Domäne.

2.1 While-Programme

Definition 2.1 (Syntax)
Die *Syntax von beschrifteten While-Programmen* ist durch folgende BNF[1] gegeben:

$$a ::= k \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2$$
$$\rightsquigarrow \text{Arithmetische Ausdrücke über ganzen Zahlen}$$

$$b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2$$
$$\rightsquigarrow \text{Boolsche Ausdrücke}$$

$$c ::= [\texttt{skip}]^l \mid [x := a]^l \mid c_1; c_2$$
$$\mid \texttt{if } [b]^l \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end}$$
$$\mid \texttt{while } [b]^l \texttt{ do } c \texttt{ end}$$
$$\rightsquigarrow \text{Befehle (Commands) mit Beschriftung (Label) } l$$

mit $k \in \mathbb{Z}$, $t \in \mathbb{B}$, und $x \in \text{Var}$. Wir nehmen an, dass alle Labels in Programmen verschieden sind.

Beschriftete Befehle werden *Blöcke* genannt. □

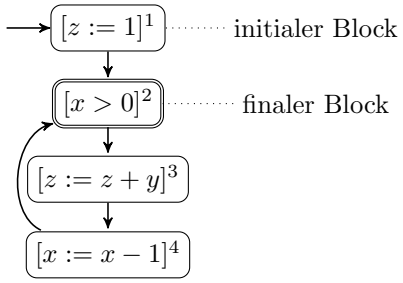
Definition 2.2 (Kontrollflussgraphen)
Programme lassen sich als *Kontrollflussgraphen* $G = (B, E, F)$ darstellen, dabei ist

B = *Blöcke* im Programm
E = Menge an *extremalen Blöcken* (initial oder final – je nach Analyse) $\subseteq B$
$F \subseteq B \times B$ = Flussrelation □

Beispiel 2.3
Typischerweise repräsentieren Kontrollflussgraphen die Struktur eines Programms

```
c = [z := 1]^1;
while [x > 0]^2 do
    [z := z+y]^3;
    [x := x-1]^4;
end
```

gibt



[1] Backus-Naur-Form, mehr Infos hier: https://de.wikipedia.org/wiki/Backus-Naur-Form

□

Bemerkung

Es gibt Datenflussanalysen, die Programme entgegen der Befehlsfolge (rückwärts) analysieren. Daher werden wir bei einer Datenflussanalyse den zugrundeliegenden Kontrollflussgraphen genau festlegen. □

Annahme

Für Kontrollflussgraphen wird – je nach Analyse – angenommen, dass

- der initiale Block keine eingehenden Kanten hat
- die finalen Blöcke keine ausgehenden Kanten

Diese Form lässt sich duch Hinzufügen von `skip`-Befehlen immer herstellen. Das obige Beispiel erfüllt die Bedingung für initiale Blöcke, verletzt aber die Bedingung für finale Blöcke. □

2.2 Monotone Frameworks

Monotone Frameworks nutzen einen vollständigen Verband als abstrakte Datendomäne und imitieren die Befehle des Programms durch monotone Funktionen.

Definition 2.4 (Datenflusssystem)
Ein *Datenflusssystem* ist ein Tupel $S = (G, (D, \leq), i, f)$ mit

- $G = (B, E, F)$ ein *Kontrollflussgraph*
- $(D, \leq)$ ein *vollständiger Verband* mit (ACC)
- $i \in D$ ein *Anfangswert* für Extremalblöcke
- $f = \{ f_b : D \to D \mid b \in D \}$ eine Familie von *Transferfunktionen*, eine für jeden Block, die alle *monoton* sind. □

Bemerkung
Falls man einen vollständigen Verband $(D, \leq)$ benutzen möchte, in dem (DCC) gilt, kann man den dualen Verband $(D, \geq)$ verwenden, in dem dann (ACC) gilt. □

Die Datenflussanalyse induziert ein *Gleichungssystem*

$$X_b = \begin{cases} i & \text{, falls } b \in E \\ \bigsqcup\{ f_{b'}(X_{b'}) \mid (b', b) \in F \} & \text{, sonst} \end{cases}$$

in dem Extremalblöcke durch den spezifizierten Initialwert repräsentiert werden und alle anderen Blöcke durch den Join der Werte, die man durch die eingehenden Kanten erhält.

Ein Vektor $(d_1, \ldots, d_{|B|}) \in D^{|B|}$ heißt *Lösung von S*, falls

$$d'_b = \begin{cases} i & \text{, falls } b \in E \\ \bigsqcup\{ f_{b'}(d_{b'}) \mid (b', b) \in F \ \} & \text{, sonst} \end{cases}.$$

Um den Zusammenhang zwischen den Lösungen des Gleichungssystems von S sowie Fixpunkten herzustellen, definiere die Funktion

$$g_S : D^{|B|} \longrightarrow D^{|B|}$$
$$(d_1, \ldots, d_{|B|}) \longmapsto (d'_1, \ldots, d'_{|B|})$$

durch

$$d'_b = \begin{cases} i & \text{, falls } b \in E \\ \bigsqcup\{ f_{b'}(d_{b'}) \mid (b', b) \in F \} & \text{, sonst} \end{cases}.$$

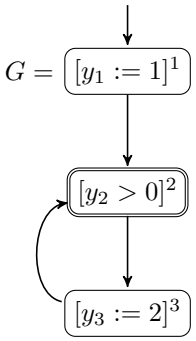
Satz 2.5
Vektor $\bar{d} = (d_1, \ldots, d_{|B|}) \in D^{|B|}$ löst das Gleichungssystem von S gdw. $g_S(\bar{d}) = \bar{d}$, d.h. $\bar{d}$ ist Fixpunkt von g_s □

Bemerkung
Mittels Kleene-Iteration kann der Kleinste Fixpunkt gefunden werden. Dieser liefert die präziseste Information. □

Beispiel 2.6

Es soll eine Programmanalyse definiert werden, die die Menge an Variablen berechnet, die an einem Programmpunkt geschrieben worden sind. Betrachte das Programm mit

```
c = [y1:= 1]1;
while [y2 >0]2 do
    [y3:= 2]3;
end
```



Das zugehörige Datenflusssystem ist

$$S = (G, \mathcal{P}(\{y_1, y_2, y_3\}, \subseteq), \varnothing, \{f_1, f_2, f_3\})$$

mit

$$\begin{aligned} f_1, f_2, f_3 : \mathcal{P}(\{y_1, y_2, y_3\}) &\to \mathcal{P}(\{y_1, y_2, y_3\}) \\ f_1(X) &:= X \cup \{y_1\} \\ f_2(X) &:= X \\ f_3(X) &:= X \cup \{y_3\} \end{aligned}$$

Das Datenflusssystem induziert das Gleichungssystem

$$\begin{aligned} X_1 &= \varnothing \\ X_2 &= \underbrace{X_1 \cup \{y_1\}}_{=f_1(X_1)} \cup \underbrace{X_3 \cup \{y_3\}}_{f_3(X_3)} \\ X_3 &= \underbrace{X_2}_{f_2(X_2)} \end{aligned}$$

Eine Lösung ist $(\varnothing, \{y_1, y_3\}, \{y_1, y_3\})$. □

2.3 Beispiele zu intraprozeduraler Datenflussanalyse

Datenflussanalysen lassen sich anhand von vier Parametern klassifizieren:

1. **Richtung der Analyse**

   Vorwärts Berechne Information über die Vergangenheit von Daten.

   Rückwärts Berechne Information über das zukünftige Verhalten von Daten.

2. **Approximation der Information**

   May Überapproximiere die Information über Daten.

   > May-Analysen spiegeln jede Information wider, die (möglicherweise) in einem realen Ablauf eintreten kann. Damit können May-Informationen nicht verletzt werden. Allerdings ist nicht garantiert, dass eine Information auch in einem realen Ablauf erreicht wird.

   Must Unterapproximiere die Information über Daten.

   > Must-Analysen spiegeln nur Information wider, die definitiv in jedem realen Ablauf eintritt. Damit liefern Must-Analysen verlässlich eintretende Informationen. Allerdings geben Must-Analysen nicht alle eintretenden Informationen wieder.

3. **Berücksichtigung von Prozeduren**

   Intraprozedural Analyse einer einzelnen Prozedur, typischerweise `main`. Um Programme intraprozedural zu analysieren, nutze *Inlining*. Inlining ist bei Rekursion nicht möglich. Intraprozedurale Analysen unterstützen keine Rekursion.

   Interprozedural Analyse eines ganzen Programms mit Rekursion.

4. **Berücksichtigung des Kontrollflusses**

   Control-flow sensitive Berücksichtige die Anordnung der Befehle im Programm. Die Analyse berechnet separate Information für jeden Block.

   > Vorteil: präzise. Nachteil: ineffizient.

   Control-flow insensitive Vergiss die Anordnung der Befehle im Programm. Die Analyse berechnet eine Information für alle Blöcke.

   > Vorteil: effizient. Nachteil: unpräzise.

Wir betrachten vier klassische Analysen, die alle vier Kombinationen aus Richtung und Approximation abdecken. Allerdings sind alle vier Analysen conrol-flow sensitiv und intraprozedural. Folgende Tabelle zeigt die Analysen und den Zusammenhang zwischen:

Richtung	$\leftrightarrow$	Wahl des Kontrollflussgraphen mit Extremalknoten
Approximation	$\leftrightarrow$	Wahl des Verbandes mit Join und Bottom.

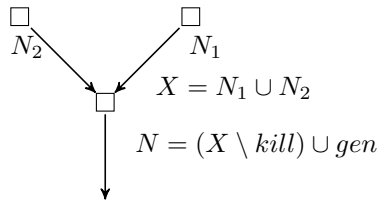
Instanz	**Reaching-Definitions**	**Available-Expr.**	**Live-Var.**	**Busy-Expr.**
Richtung	vorwärts		rückwärts	
Extremal (E)	initaler Block		finale Blöcke	
Fluss. (F)	in Programmordnung		gegen Programmordnung	
Approx.	may	must	may	must
Verband	$(\mathbb{P}(\mathit{Vars} \times \mathit{Blocks} \cup \{?\}), \subseteq)$	$(\mathbb{P}(\mathit{AExp}), \supseteq)$	$(\mathbb{P}(\mathit{Vars}), \subseteq)$	$(\mathbb{P}(\mathit{AExp}), \supseteq)$
Join ($\sqcup$)	$\cup$	$\cap$	$\cup$	$\cap$
Bottom ($\bot$)	$\varnothing$	AExp	$\varnothing$	AExp
Anfangsw. (i)	$\{ (x, ?) \mid x \in \mathit{Vars} \}$	$\varnothing$	Vars	$\varnothing$
Transferf. (f)	$f_b(X) := (X \setminus \mathit{kill}(b)) \cup \mathit{gen}(b)$			

2.3.1 Reaching-Definitions-Analyse

Ziel: Berechne für jeden Block die Zuweisungen, die es gegeben haben könnte (nicht überschrieben), wenn eine Ausführung den Block erreicht.

Klassifikation: Vorwärtsanalyse, May-Analyse.

Idee:



Anwendung: Berechnung von *Use-Definition-Chains*, die angeben, welche Zuweisungen (Definitions) von einem Block genutzt werden. Use-Definition-Chains sind die Grundlage für *Code-Motion-Optimierungen.*

Beispiel 2.7

Betrachte ein Programm mit Variablen *Vars* und Blöcken *Blocks*.
Definiere das Datenflusssystem $S = (G, (D, \preceq), i, \{ f_b : D \to D \mid b \in \mathit{Blocks} \})$ mit:

- Kontrollflussgraph $G = (B, E, F)$:
  $B = \mathit{Blocks}$, $E =$ initialer Block, $F =$ Kontrollfluss in Programmordnung.
- Verband $(D, \preceq) = (\mathbb{P}(\mathit{Vars} \times (\mathit{Blocks} \cup \{?\})), \subseteq)$.

  Es handelt sich um einen *Potenzmengenverband.* (ACC) gilt, da der Verband endlich ist.
- Elementen aus $\mathit{Vars} \times (\mathit{Blocks} \cup \{?\})$ mit folgender Bedeutung:

  $(x, ?)$ = x ist möglicherweise noch nicht initialisiert.
  (x, b) = x hat möglicherweise die letzte Zuweisung von Block b erhalten.

- Anfangswert $i = \{ (x, ?) \mid x \in \mathit{Vars} \}$.
- Transferfunktionen $f_b : D \to D$ mit

$$f_b : \mathbb{P}(\mathit{Vars} \times (\mathit{Blocks} \cup \{?\})) \to \mathbb{P}(\mathit{Vars} \times (\mathit{Blocks} \cup \{?\}))$$
$$X \mapsto (X \setminus \mathit{kill}(b)) \cup \mathit{gen}(b)$$

Die Mengen $kill(b), gen(b) \subseteq Vars \times (Blocks \cup \{?\})$ sind

$$kill(b) := \begin{cases} \{ (x, ?) \} \cup \{ (x, b') \mid b' \in Blocks \}, & \text{falls } b = [x := a]^b \\ \varnothing, & \text{sonst.} \end{cases}$$

$\rightsquigarrow$ Zuweisungen, die von Block b überschrieben werden.

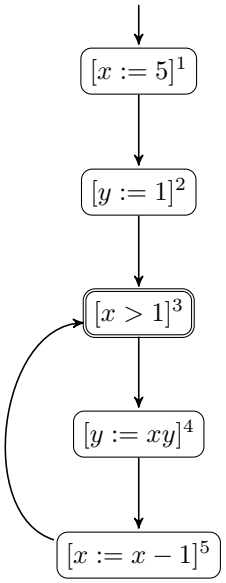
$$gen(b) := \begin{cases} \{ (x, b) \}, & \text{falls } b = [x := a]^b \\ \varnothing, & \text{sonst.} \end{cases}$$

$\rightsquigarrow$ Zuweisungen, die von Block b generiert werden.

Die Transferfunktionen sind monoton.

Betrachte folgendes Beispielprogramm und den dazugehörigen Kontrollflussgraphen:

```
[x:=5]1;
[y:=1]2;
while [x > 1]3 do
    [y:=xy]4;
    [x:=x-1]5;
end
```



Die Transferfunktionen sind

Block	$kill(b)$	$gen(b)$	$f_b(X)$
$[x := 5]^1$	$\{ (x, ?), (x, 1), (x, 5) \}$	$\{ (x, 1) \}$	$(X \setminus \{ (x, ?), (x, 1), (x, 5) \}) \cup \{(x, 1)\}$
$[y := 1]^2$	$\{ (y, ?), (y, 2), (y, 4) \}$	$\{ (y, 2) \}$	$(X \setminus \{ (y, ?), (y, 2), (y, 4) \}) \cup \{(y, 2)\}$
$[x > 1]^3$	$\varnothing$	$\varnothing$	X
$[y := xy]^4$	$\{ (y, ?), (y, 2), (y, 4) \}$	$\{ (y, 4) \}$	$(X \setminus \{ (y, ?), (y, 2), (y, 4) \}) \cup \{(y, 4)\}$
$[x := x - 1]^5$	$\{ (x, ?), (x, 1), (x, 5) \}$	$\{ (x, 5) \}$	$(X \setminus \{ (x, ?), (x, 1), (x, 5) \}) \cup \{(x, 5)\}$

In der Tabelle sind die $kill(b)$ Mengen auf die Blöcke eingeschränkt worden, die eine Zuweisung auf die jeweilige Variable durchführen.

Das vom Datenflusssystem induzierte Gleichungssystem ist:

$$
\begin{aligned}
X_1 &= \underbrace{\{(x,?),(y,?)\}}_{=i} \\
X_2 &= \underbrace{(X_1 \setminus \{(x,?),(x,1),(x,5)\}) \cup \{\,(x,1)\,\}}_{=f_1(X_1)} \\
X_3 &= \underbrace{((X_2 \setminus \{(y,?),(y,2),(y,4)\}) \cup \{\,(y,2)\,\})}_{=f_2(X_2)} \cup \underbrace{((X_5 \setminus \{(x,?),(x,1),(x,5)\}) \cup \{\,(x,5)\,\})}_{=f_5(X_5)} \\
X_4 &= X_3 \\
X_5 &= (X_4 \setminus \{(y,?),(y,2),(y,4)\}) \cup \{\,(y,4)\,\}
\end{aligned}
$$

Durch Vereinfachen erhalten wir:

$$
\begin{aligned}
X_1 &= \{(x,?),(y,?)\} \\
X_2 &= (X_1 \setminus \{(x,?),(x,1),(x,5)\}) \cup \{\,(x,1)\,\} \\
X_3 &= ((X_2 \setminus \{(y,?),(y,2),(y,4)\}) \cup \{\,(y,2)\,\}) \cup ((X_5 \setminus \{(x,?),(x,1),(x,5)\}) \cup \{\,(x,5)\,\}) \\
X_4 &= X_3 \\
X_5 &= (X_4 \setminus \{(y,?),(y,2),(y,4)\}) \cup \{\,(y,4)\,\}
\end{aligned}
$$

Berechne eine Lösung des Gleichungssystems durch Iteration von

$$g_S : \mathbb{P}(\mathit{Vars} \times (\mathit{Blocks} \cup \{?\}))^5 \to \mathbb{P}(\mathit{Vars} \times (\mathit{Blocks} \cup \{?\}))^5$$

auf $\bot$ von $(\mathbb{P}(\mathit{Vars} \times (\mathit{Blocks} \cup \{?\}))^5, \subseteq^5)$ bis zum kleinsten Fixpunkt:

Iter.	$(d_1$	d_2	d_3	d_4	$d_5)$
$g_S^0(\bot)$	$(\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing)$
$g_S^1(\bot)$	$(\{\,(x,?),(y,?)\,\}$	$\{\,(x,1)\,\}$	$\{\,(y,2),(x,5)\,\}$	$\varnothing$	$\{\,(y,4)\,\})$
$g_S^2(\bot)$	$(\{\,(x,?),(y,?)\,\}$	$\{\,(y,?),(x,1)\,\}$	$\{\,(x,1),(y,2),(y,4),(x,5)\,\}$	$\{\,(y,2),(x,5)\,\}$	$\{\,(y,4)\,\})$
$g_S^3(\bot)$	$(\{\,(x,?),(y,?)\,\}$	$\{\,(y,?),(x,1)\,\}$	$\{\,(x,1),(y,2),(y,4),(x,5)\,\}$	$\{\,(x,1),(y,2),(y,4),(x,5)\,\}$	$\{\,(x,5)(y,4)\,\})$
$g_S^4(\bot)$	$(\{\,(x,?),(y,?)\,\}$	$\{\,(y,?),(x,1)\,\}$	$\{\,(x,1),(y,2),(y,4),(x,5)\,\}$	$\{\,(x,1),(y,2),(y,4),(x,5)\,\}$	$\{\,(x,1),(x,5),(y,4)\,\})$
$g_S^5(\bot)$	$(\{\,(x,?),(y,?)\,\}$	$\{\,(y,?),(x,1)\,\}$	$\{\,(x,1),(y,2),(y,4),(x,5)\,\}$	$\{\,(x,1),(y,2),(y,4),(x,5)\,\}$	$\{\,(x,1),(x,5),(y,4)\,\})$

Es gilt $g_S(g_S^4(\bot)) = g_S^4(\bot)$. Also ist $g_S^4(\bot)$ der kleinste Fixpunkt.

Die kleinste Lösung des Gleichungssystems ist

$$
\begin{aligned}
X_1 &= \{\,(x,?),(y,?)\,\} & X_2 &= \{\,(y,?),(x,1)\,\} \\
X_3 &= \{\,(x,1),(y,2),(y,4),(x,5)\,\} & X_4 &= \{\,(x,1),(y,2),(y,4),(x,5)\,\} \\
X_5 &= \{\,(x,1),(x,5),(y,4)\,\}.
\end{aligned}
$$

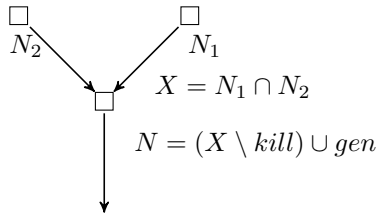
Die kleinste Lösung ist die gewünschte Information. Größere May-Information bedeutet Informationsverlust. □

2.3.2 Available-Expressions-Analyse

Ziel: Berechne für jeden Block die Ausdrücke, die auf allen Pfaden zu dem Block definitiv berechnet worden sind (nicht zwischendurch geändert).

Klassifikation: Vorwärtsanalyse, Must-Analyse.

Idee:



Anwendung: Vermeide erneute Berechnung bekannter Werte.

Beispiel 2.8

Betrachte ein Programm mit Teilausdrücken *AExp* und Blöcken *Blocks*. Mit $AExp(a)$ bezeichnen wir die Teilausdrücke von $a \in AExp$. Mit $Vars(a)$ bezeichnen wir die Variablen von $a \in AExp$.
Definiere das Datenflusssystem $S = (G, (D, \preceq), i, \{ f_b : D \to D \mid b \in Blocks \})$ mit:

- Kontrollflussgraph $G = (B, E, F)$:
  $B = Blocks$, $E =$ initialer Block, $F =$ Kontrollfluss in Programmordnung.
- Verband $(D, \preceq) = (D, \preceq) = (\mathbb{P}(AExp), \supseteq)$.

  Es handelt sich um einen (dualen) Potenzmengenverband. (ACC) gilt, da der Verband endlich ist.
- Anfangswert $i = \varnothing$.
- Transferfunktionen $f_b : D \to D$:

$$\begin{aligned} f_b : \mathbb{P}(AExp) &\to \mathbb{P}(AExp) \\ X &\mapsto (X \setminus kill(b)) \cup gen(b) \end{aligned}$$

  Die Mengen $kill(b), gen(b) \subseteq AExp$ sind

$$kill(b) := \begin{cases} \{ a' \in AExp \mid x \in Vars(a') \}, & \text{falls } b = [x := a]^b \\ \varnothing, & \text{sonst.} \end{cases}$$

  $\rightsquigarrow$ Teilausdrücke, die x enthalten und daher von Block b geändert werden.

$$gen(b) := \begin{cases} \{ a' \in AExp(a) \mid x \notin Vars(a') \}, & \text{falls } b = [x := a]^b \\ AExp(cond), & \text{falls } b = [cond]^b \\ \varnothing, & \text{sonst.} \end{cases}$$

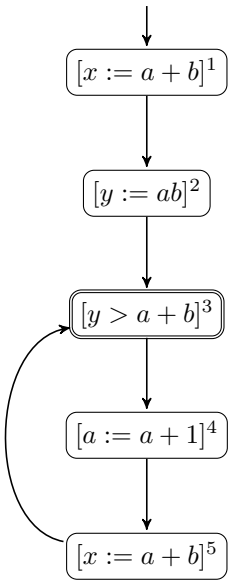
  $\rightsquigarrow$ Teilausdrücke, die von Block b genutzt werden.

  Beachte, dass bei einer Zuweisung, deren rechte Seite den zugewiesenen Wert beinhaltet, die entsprechenden Ausdrücke nicht available werden, da sich ihr Wert ändert (z.B. ändert sich durch die Zuweisung $a := a + 1$ der Wert von $a + 1$). Daher ist die Einschränkung $x \notin Vars(a')$ oben nötig.

  Die Transferfunktionen sind monoton.

Betrachte folgendes Beispielprogramm:

```
[x:=a+b]1;
[y:=ab]2;
while [y > a+b]3 do
    [a:=a+1]4;
    [x:=a+b]5;
end
```



Die Transferfunktionen sind

Block	$kill(b)$	$gen(b)$	$f_b(X)$
$[x := a + b]^1$	$\varnothing$	$\{ a + b \}$	$X \cup \{ a + b \}$
$[y := ab]^2$	$\varnothing$	$\{ ab \}$	$X \cup \{ ab \}$
$[y > a + b]^3$	$\varnothing$	$\{ a + b \}$	$X \cup \{ a + b \}$
$[a := a + 1]^4$	$\{ a + b, ab, a + 1 \}$	$\varnothing$	$X \setminus \{ a + b, ab, a + 1 \}$
$[x := a + b]^5$	$\varnothing$	$\{ a + b \}$	$X \cup \{ a + b \}$

Das vom Datenflusssystem induzierte Gleichungssystem ist

$$
\begin{aligned}
X_1 &= \underbrace{\varnothing}_{=i} \\
X_2 &= \underbrace{X_1 \cup \{ a + b \}}_{=f_1(X_1)} \\
X_3 &= \underbrace{(X_2 \cup \{ ab \})}_{=f_2(X_2)} \cap \underbrace{(X_5 \cup \{ a + b \})}_{=f_5(X_5)} \\
X_4 &= X_3 \cup \{ a + b \} \\
X_5 &= X_4 \setminus \{ a + b, ab, a + 1 \}
\end{aligned}
$$

Durch Vereinfachen erhalten wir:

$$
\begin{aligned}
X_1 &= \varnothing \\
X_2 &= X_1 \cup \{ a + b \} \\
X_3 &= (X_2 \cup \{ ab \}) \cap (X_5 \cup \{ a + b \}) \\
X_4 &= X_3 \cup \{ a + b \} \\
X_5 &= X_4 \setminus \{ a + b, ab, a + 1 \}
\end{aligned}
$$

Berechne eine Lösung des Gleichungssystems durch Iteration von

$$g_S : \mathbb{P}(AExp)^5 \to \mathbb{P}(AExp)^5$$

auf $\bot$ von $(\mathbb{P}(AExp)^5, \supseteq^5)$ bis zum kleinsten Fixpunkt:

Iter.	$(d_1$	d_2	d_3	d_4	$d_5)$
$g_S^0(\bot)$	$(\{ a+b, ab, a+1 \}$	$\{ a+b, ab, a+1 \}$	$\{ a+b, ab, a+1 \}$	$\{ a+b, ab, a+1 \}$	$\{ a+b, ab, a+1 \})$
$g_S^1(\bot)$	$(\varnothing$	$\{ a+b, ab, a+1 \}$	$\{ a+b, ab, a+1 \}$	$\{ a+b, ab, a+1 \}$	$\varnothing)$
$g_S^2(\bot)$	$(\varnothing$	$\{ a+b \}$	$\{ a+b \}$	$\{ a+b, ab, a+1 \}$	$\varnothing)$
$g_S^3(\bot)$	$(\varnothing$	$\{ a+b \}$	$\{ a+b \}$	$\{ a+b \}$	$\varnothing)$
$g_S^4(\bot)$	$(\varnothing$	$\{ a+b \}$	$\{ a+b \}$	$\{ a+b \}$	$\varnothing)$

Es gilt $g_S(g_S^3(\bot)) = g_S^3(\bot)$. Also ist $g_S^3(\bot)$ der kleinste Fixpunkt.

Die kleinste Lösung des Gleichungssystems ist

$$X_1 = \varnothing = X_5 \qquad\qquad X_2 = \{ a+b \} = X_3 = X_4.$$

Die kleinste Lösung ist die gewünschte Information. Größere (bzgl. $\supseteq$) Must-Information bedeutet Informationsverlust. □

Bemerkung

Wir haben hier den größten Fixpunkt auf dem Potenzmengenverband $(\mathbb{P}(AExp), \subseteq)$ berechnet. Durch Dualisierung des Verbandes zu $(\mathbb{P}(AExp), \supseteq)$ konnten wir eine kleinste Fixpunktberechnung und so unser Framework mit (ACC) nutzen. □

2.3.3 Live-Variables-Analyse

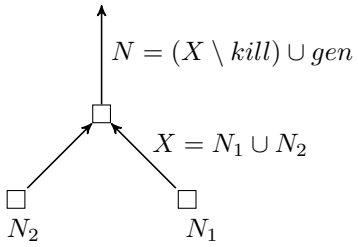
Definition 2.9 (Liveness)

Eine Variable heißt *lebendig* (*live*) am Ausgang eines Blocks, falls es einen Ablauf von diesem Block zu einem anderen Block geben könnte (nicht überschrieben), der die Variable in einer Bedingung oder Zuweisung (rechte Seite) nutzt. □

Ziel: Berechne für jeden Block die Variablen, die am Ausgang lebendig sind.

Klassifikation: Rückwärtsanalyse, May-Analyse.

Idee:



Anwendung:

(1) *Register-Allocation*: Falls x lebendig ist, wird die Variable vermutlich bald genutzt und sollte ein Register erhalten. Ist x nicht mehr lebendig, kann das Register neu vergeben werden.

(2) *Dead-Code-Elimination*: Ist x am Ausgang einer Zuweisung (zu x) nicht lebendig, kann die Zuweisung entfernt werden. Auf ähnliche Weise lassen sich Variablen zusammenfassen: sind x und y nie gemeinsam lebendig, verwende eine Variable z.

Beispiel 2.10

Betrachte ein Programm mit Blöcken *Blocks* und Variablen *Vars*. Ferner sei $Vars(a)$ die Menge der Variablen in einem Ausdruck a.
Definiere das Datenflusssystem $S = (G, (D, \preceq), i, \{\, f_b : D \to D \mid b \in \mathit{Blocks} \,\})$ mit

- Kontrollflussgraph $G = (B, E, F)$:
  $B = \mathit{Blocks}$, $E =$ finale Blöcke, $F =$ Kontrollfluss gegen die Programmordnung.
- Verband $(D, \preceq) = (\mathbb{P}(\mathit{Vars}), \subseteq)$.

  Es handelt sich um einen (Potenzmengen)verband. (ACC) gilt, da der Verband endlich ist.
- Anfangswert $i = \mathit{Vars}$ (am Ende des Programms sind per Definition alle Variablen lebendig).
- Transferfunktionen $f_b : D \to D$:

$$f_b : \mathbb{P}(\mathit{Vars}) \to \mathbb{P}(\mathit{Vars})$$
$$X \mapsto (X \setminus \mathit{kill}(b)) \cup \mathit{gen}(b)$$

  Die Mengen $\mathit{kill}(b), \mathit{gen}(b) \subseteq \mathit{Vars}$ sind

$$\mathit{kill}(b) := \begin{cases} \{\, x \,\}, & \text{falls } b = [x := a]^b \\ \varnothing, & \text{sonst.} \end{cases}$$

  $\rightsquigarrow$ Variablen, die von Block b überschrieben werden.

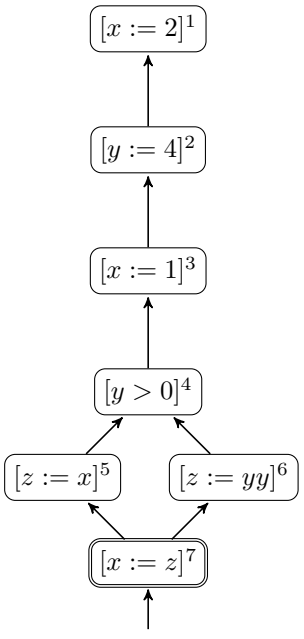
$$\mathit{gen}(b) := \begin{cases} \mathit{Vars}(a), & \text{falls } b = [x := a]^b \\ \mathit{Vars}(\mathit{cond}), & \text{falls } b = [\mathit{cond}]^b \\ \varnothing, & \text{sonst.} \end{cases}$$

  $\rightsquigarrow$ Variablen, die von Block b genutzt werden.

  Die Transferfunktionen sind monoton.

Betrachte folgendes Beispielprogramm:

```
[x:=2]1;
[y:=4]2;
[x:=1]3;
if [y>0]4 then
    [z:=x]5
else
    [z:=yy]6;
endif;
[x:=z]7;
```



Die Transferfunktionen sind

Block	$kill(b)$	$gen(b)$	$f_b(X)$
$[x := 2]^1$	$\{ x \}$	$\varnothing$	$X \setminus \{ x \}$
$[y := 4]^2$	$\{ y \}$	$\varnothing$	$X \setminus \{ y \}$
$[x := 1]^3$	$\{ x \}$	$\varnothing$	$X \setminus \{ x \}$
$[y > 0]^4$	$\varnothing$	$\{ y \}$	$X \cup \{ y \}$
$[z := x]^5$	$\{ z \}$	$\{ x \}$	$(X \setminus \{ z \}) \cup \{ x \}$
$[z := yy]^6$	$\{ z \}$	$\{ y \}$	$(X \setminus \{ z \}) \cup \{ y \}$
$[x := z]^7$	$\{ x \}$	$\{ z \}$	$(X \setminus \{ x \}) \cup \{ z \}$

Das vom Datenflusssystem induzierte Gleichungssystem ist

$$
\begin{aligned}
X_1 &= X_2 \setminus \{y\} \\
X_2 &= X_3 \setminus \{x\} \\
X_3 &= X_4 \cup \{ y \} \\
X_4 &= \underbrace{((X_5 \setminus \{ z \}) \cup \{ x \})}_{=f_5(X_5)} \cup \underbrace{((X_6 \setminus \{ z \}) \cup \{ y \})}_{=f_6(X_6)} \\
X_5 &= (X_7 \setminus \{ x \}) \cup \{ z \} \\
X_6 &= (X_7 \setminus \{ x \}) \cup \{ z \} \\
X_7 &= \underbrace{\{ x, y, z \}}_{=i}
\end{aligned}
$$

Durch Vereinfachen erhalten wir:

$$
\begin{aligned}
X_1 &= X_2 \setminus \{y\} \\
X_2 &= X_3 \setminus \{x\} \\
X_3 &= X_4 \cup \{ y \} \\
X_4 &= ((X_5 \setminus \{ z \}) \cup \{ x \}) \cup ((X_6 \setminus \{ z \}) \cup \{ y \}) \\
X_5 &= (X_7 \setminus \{ x \}) \cup \{ z \} \\
X_6 &= (X_7 \setminus \{ x \}) \cup \{ z \} \\
X_7 &= \{ x, y, z \}
\end{aligned}
$$

Berechne eine Lösung des Gleichungssystems durch Iteration von

$$g_S : \mathbb{P}(\mathit{Vars})^7 \to \mathbb{P}(\mathit{Vars})^7$$

auf $\bot$ von $(\mathbb{P}(\mathit{Vars})^7, \subseteq^7)$ bis zum kleinsten Fixpunkt:

Iter.	$(d_1$	d_2	d_3	d_4	d_5	d_6	$d_7)$
$g_S^0(\bot)$	$(\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing)$
$g_S^1(\bot)$	$(\varnothing$	$\varnothing$	$\{ y \}$	$\{ y, x \}$	$\{ z \}$	$\{ z \}$	$\{ x, y, z \})$
$g_S^2(\bot)$	$(\varnothing$	$\{ y \}$	$\{ y, x \}$	$\{ y, x \}$	$\{ y, z \}$	$\{ y, z \}$	$\{ x, y, z \})$
$g_S^3(\bot)$	$(\varnothing$	$\{ y \}$	$\{ y, x \}$	$\{ y, x \}$	$\{ y, z \}$	$\{ y, z \}$	$\{ x, y, z \})$

Es gilt $g_S(g_S^2(\bot)) = g_S^2(\bot)$. Also ist $g_S^2(\bot)$ der kleinste Fixpunkt.

Die kleinste Lösung des Gleichungssystems ist

$$X_1 = \varnothing \qquad X_2 = \{ y \}$$
$$X_3 = \{ y, x \} = X_4 \qquad X_5 = \{ y, z \} = X_6$$
$$X_7 = \{ x, y, z \}.$$

Die kleinste Lösung ist die gewünschte Information. Größere May-Information bedeutet Informationsverlust.

Block $[x := 2]^1$ kann entfernt werden. □

2.3.4 Very-Busy-Expressions-Analyse

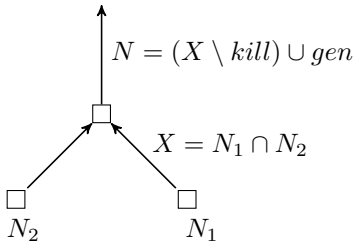
Definition 2.11

Ein Ausdruck heißt *very busy* am Ausgang eines Blocks, falls der Ausdruck auf jedem Pfad, der von diesem Block ausgeht, verwendet wird, bevor eine der enthaltenen Variablen neu geschrieben wird. □

Ziel: Berechne für jeden Block die Ausdrücke, die am Ausgang very busy sind.

Klassifikation: Rückwärtsanalyse, Must-Analyse.

Idee:



Anwendung: *Hoisting-Expressions:* Betrachte eine Schleife mit einem Block $x := (a + b)y$, wobei $a + b$ von der Schleife nicht geändert wird. Dann lässt sich eine Zuweisung $t := a + b$ vor der Schleife einfügen und $x := (a + b)y$ durch $x := ty$ ersetzen.

Beispiel 2.12

Betrachte ein Programm mit Teilausdrücken *AExp* und Blöcken *Blocks*. Nutze $AExp(a)$ für die Teilausdrücke von $a \in AExp$. Nutze $Vars(a)$ für die Variablen von $a \in AExp$.

Definiere das Datenflusssystem $S = (G, (D, \preceq), i, \{ f_b : D \to D \mid b \in Blocks \})$ mit

- Kontrollflussgraph $G = (B, E, F)$:
  $B = Blocks$, $E =$ finale Blöcke, $F =$ Kontrollfluss gegen die Programmordnung.
- Verband $(D, \preceq) = (\mathbb{P}(AExp), \supseteq)$.

  Es handelt sich um einen (dualen Potenzmengen)verband. (ACC) gilt, da der Verband endlich ist.
- Anfangswert $i = \varnothing$.
- Transferfunktionen $f_b : D \to D$:

$$f_b : \mathbb{P}(AExp) \to \mathbb{P}(AExp)$$
$$X \mapsto (X \setminus kill(b)) \cup gen(b)$$

Die Mengen $kill(b), gen(b) \subseteq AExp$ sind

$$kill(b) := \begin{cases} \{\, a' \in AExp \mid x \in Vars(a') \,\}, & \text{falls } b = [x := a]^b \\ \varnothing, & \text{sonst.} \end{cases}$$

$\rightsquigarrow$ Teilausdrücke, die x enthalten und daher von Block b geändert werden.

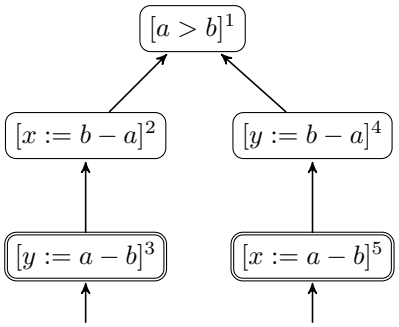
$$gen(b) := \begin{cases} AExp(a), & \text{falls } b = [x := a]^b \\ AExp(cond), & \text{falls } b = [cond]^b \\ \varnothing, & \text{sonst.} \end{cases}$$

$\rightsquigarrow$ Teilausdrücke, die von Block b genutzt werden.

Die Transferfunktionen sind monoton.

Betrachte folgendes Beispielprogramm:

```
if [a>b]^1 then
    [x:=b-a]^2;
    [y:=a-b]^3
else
    [y:=b-a]^4;
    [x:=a-b]^5
endif
```



Die Transferfunktionen sind:

Block	$kill(b)$	$gen(b)$	$f_b(X)$
$[a > b]^1$	$\varnothing$	$\varnothing$	X
$[x := b - a]^2$	$\varnothing$	$\{\, b - a \,\}$	$X \cup \{\, b - a \,\}$
$[y := a - b]^3$	$\varnothing$	$\{\, a - b \,\}$	$X \cup \{\, a - b \,\}$
$[y := b - a]^4$	$\varnothing$	$\{\, b - a \,\}$	$X \cup \{\, b - a \,\}$
$[x := a - b]^5$	$\varnothing$	$\{\, a - b \,\}$	$X \cup \{\, a - b \,\}$

Das vom Datenflusssystem induzierte Gleichungssystem ist

$$\begin{aligned} X_1 &= \underbrace{(X_2 \cup \{\, b - a \,\})}_{=f_2(X_2)} \cap \underbrace{(X_4 \cup \{\, b - a \,\})}_{=f_4(X_4)} \\ X_2 &= X_3 \cup \{\, a - b \,\} \\ X_3 &= \underbrace{\varnothing}_{=i} \\ X_4 &= X_5 \cup \{\, a - b \,\} \\ X_5 &= \underbrace{\varnothing}_{=i} \end{aligned}$$

Durch Vereinfachen erhalten wir:

$$\begin{aligned} X_1 &= (X_2 \cup \{\, b - a \,\}) \cap (X_4 \cup \{\, b - a \,\}) \\ X_2 &= X_3 \cup \{\, a - b \,\} \\ X_3 &= \varnothing \\ X_4 &= X_5 \cup \{\, a - b \,\} \\ X_5 &= \varnothing \end{aligned}$$

Berechne eine Lösung des Gleichungssystems durch Iteration von

$$g_S : \mathbb{P}(AExp)^5 \to \mathbb{P}(AExp)^5$$

auf $\bot$ von $(\mathbb{P}(AExp)^5, \supseteq^5)$ bis zum kleinsten Fixpunkt:

Iter.	$(d_1$	d_2	d_3	d_4	$d_5)$
$g_S^0(\bot)$	$(\{a-b, b-a\}$	$\{a-b, b-a\}$	$\{a-b, b-a\}$	$\{a-b, b-a\}$	$\{a-b, b-a\})$
$g_S^1(\bot)$	$(\{a-b, b-a\}$	$\{a-b, b-a\}$	$\varnothing$	$\{a-b, b-a\}$	$\varnothing)$
$g_S^2(\bot)$	$(\{a-b, b-a\}$	$\{a-b\}$	$\varnothing$	$\{a-b\}$	$\varnothing)$
$g_S^3(\bot)$	$(\{a-b, b-a\}$	$\{a-b\}$	$\varnothing$	$\{a-b\}$	$\varnothing)$

Es gilt $g_S(g_S^2(\bot)) = g_S^2(\bot)$. Also ist $g_S^2(\bot)$ der kleinste Fixpunkt.

Die kleinste Lösung des Gleichungssystems ist:

$$X_1 = \{a-b, b-a\} \qquad X_2 = \{a-b\} = X_4 \qquad X_3 = \varnothing = X_5.$$

Die kleinste Lösung ist die gewünschte Information. Größere (bzgl. $\supseteq$) Must-Information bedeutet Informationsverlust. □

Bemerkung

Wir haben hier den größten Fixpunkt auf dem Potenzmengenverband $(\mathbb{P}(AExp), \subseteq)$ berechnet. Durch Dualisierung des Verbandes zu $(\mathbb{P}(AExp), \supseteq)$ konnten wir eine kleinste Fixpunktberechnung und so unser Framework mit (ACC) nutzen. □

2.3.5 Distributive Frameworks

Definition 2.13 (Distributive Funktionen)

Eine Funktion f auf einem Verband $(D, \leq)$ heißt *distributiv*, falls für alle $a, b \in D$ gilt: $f(a) \sqcup f(b) = f(a \sqcup b)$. (Beachte, dass " $\leq$ " für monotone Funktionen immer gilt.) □

Satz 2.14

Sei f eine Funktion auf einem Verband $(D, \leq)$. Falls f distributiv ist, so ist f auch monoton. □

Werden Datenflusssysteme $S = (G, (D, \preceq), i, \{f_b : D \to D \mid b \in Blocks\})$ betrachtet, deren Transferfunktionen f_b nicht nur monoton sondern *distributiv* sind, dann spricht man von einem *distributiven Framework*.

In den obigen vier Beispielen nutzten alle Verbände die Domäne $(\mathbb{P}(A), \sqsubseteq)$ über einer endlichen Menge A und mit $\sqsubseteq \in \{\subseteq, \supseteq\}$. Ferner waren die Transferfunktionen $f_b : \mathbb{P}(A) \to \mathbb{P}(A)$ definiert durch

$$f_b(X) := (X \setminus kill(b)) \cup gen(b) \qquad \text{mit} \qquad kill(b), gen(b) \subseteq A.$$

Werden nur Datenflusssysteme der Form $S = (G, (\mathbb{P}(A), \sqsubseteq), i, f)$ mit f bestehend aus Gen/Kill-Transferfunktionen betrachtet, spricht man von einem *Bitvektor-Framework*. Der Grund für den Namen ist, dass sich die Datenflussmengen in $\mathbb{P}(A)$ als Bitvektoren darstellen lassen.

Satz 2.15

Bitvektor-Frameworks sind distributive Frameworks. □

2.3.6 Effizientere Fixpunktberechnung

Beobachtung

Die Fixpunktberechnung bestimmt den Wert von X_b in jedem Schritt neu — auch wenn sich die Belegung der Variablen der Vorgängerblöcke nicht geändert hat. □

Idee: Modifiziere die Fixpunktberechnung, so dass Variablen X_b nur bei Änderung der Eingabe neu berechnet werden.

Ansatz: Führe Worklist in die Fixpunktberechnung ein.

Algorithmus: Worklist-Algorithmus für *lfp*

Eingabe: Datenflusssystem $S = (G, (D, \preceq), i, f)$ mit $G = (B, E, F)$
Variablen: X_b für Blöcke $b \in B$, initial $X_b = \bot$

$W \leftarrow \varepsilon$; `// Worklist, initial leer`
forall $(b, b') \in F$ **do** $W \leftarrow W.(b, b')$;
forall $b \in E$ **do** $X_b \leftarrow i$;
while $W \neq \varepsilon$ **do**
  pop (b, b') **from** W;
  if $f_b(X_b) \not\preceq X_{b'}$ **then**
    $X_{b'} \leftarrow X_{b'} \sqcup f_b(X_b)$;
    forall $(b', b'') \in F$ **do**
      if $(b', b'') \notin W$ **then** $W := W.(b', b'')$;
    end
  end
end

Ausgabe: X_b für jeden Block $b \in B$

Satz 2.16

Sei das Datenflusssystem S die Eingabe für obigen Algorithmus. Der Algorithmus terminiert und berechnet $\mathit{lfp}(g_S)$. □

Beispiel 2.17

Available-Expressions-Analyse am Beispielprogramm mittels Worklist:

Nach Initialisierung:

$$W = (1,2).(2,3).(3,4).(4,5).(5,3)$$

$$X_1 = \varnothing$$
$$X_2 = \mathit{AExp}$$
$$X_3 = \mathit{AExp}$$
$$X_4 = \mathit{AExp}$$
$$X_5 = \mathit{AExp}$$

Es gilt $f_1(X_1) = \{\, a + b \,\} \not\supseteq \mathit{AExp} = X_2$, also $X_2 := \mathit{AExp} \cap \{a + b\}$.
Die Kante $(2, 3)$ ist noch in der Worklist enthalten.

Nach Iteration 1:

$$W = (2,3).(3,4).(4,5).(5,3)$$

$$\begin{aligned} X_1 &= \varnothing \\ X_2 &= \{\, a+b \,\} \\ X_3 &= \mathit{AExp} \\ X_4 &= \mathit{AExp} \\ X_5 &= \mathit{AExp} \end{aligned}$$

Es gilt $f_2(X_2) = \{\, a+b, ab \,\} \not\supseteq \mathit{AExp} = X_3$, also $X_3 := \mathit{AExp} \cap \{a+b, ab\}$.
Die Kante $(3,4)$ ist noch in der Worklist enthalten.

Nach Iteration 2:

$$W = (3,4).(4,5).(5,3)$$

$$\begin{aligned} X_1 &= \varnothing \\ X_2 &= \{\, a+b \,\} \\ X_3 &= \{\, a+b, ab \,\} \\ X_4 &= \mathit{AExp} \\ X_5 &= \mathit{AExp} \end{aligned}$$

Es gilt $f_3(X_3) = \{\, a+b, ab \,\} \not\supseteq \mathit{AExp} = X_4$, also $X_4 := \mathit{AExp} \cap \{a+b, ab\}$.
Die Kante $(4,5)$ ist noch in der Worklist enthalten.

Nach Iteration 3:

$$W = (4,5).(5,3)$$

$$\begin{aligned} X_1 &= \varnothing \\ X_2 &= \{\, a+b \,\} \\ X_3 &= \{\, a+b, ab \,\} \\ X_4 &= \{\, a+b, ab \,\} \\ X_5 &= \mathit{AExp} \end{aligned}$$

Es gilt $f_4(X_4) = \varnothing \not\supseteq \mathit{AExp} = X_5$, also $X_5 := \mathit{AExp} \cap \varnothing$.
Die Kante $(5,3)$ ist noch in der Worklist enthalten.

Nach Iteration 4:

$$W = (5,3)$$

$$\begin{aligned} X_1 &= \varnothing \\ X_2 &= \{\, a+b \,\} \\ X_3 &= \{\, a+b, ab \,\} \\ X_4 &= \{\, a+b, ab \,\} \\ X_5 &= \varnothing \end{aligned}$$

Es gilt $f_5(X_5) = \{\, a+b \,\} \not\supseteq \{\, a+b, ab \,\} = X_3$, also $X_3 := \{\, a+b, ab \,\} \cap \{\, a+b \,\}$.
Die Kante $(3,4)$ wird der Worklist hinzugefügt.

Nach Iteration 5:

$$W = (3, 4)$$

$$\begin{aligned} X_1 &= \varnothing \\ X_2 &= \{\, a + b \,\} \\ X_3 &= \{\, a + b \,\} \\ X_4 &= \{\, a + b, ab \,\} \\ X_5 &= \varnothing \end{aligned}$$

Es gilt $f_3(X_3) = \{\, a + b \,\} \not\supseteq \{\, a + b, ab \,\} = X_4$, also $X_4 := \{\, a + b, ab \,\} \cap \{\, a + b \,\}$.
Die Kante $(4, 5)$ wird der Worklist hinzugefügt.

Nach Iteration 6:

$$W = (4, 5)$$

$$\begin{aligned} X_1 &= \varnothing \\ X_2 &= \{\, a + b \,\} \\ X_3 &= \{\, a + b \,\} \\ X_4 &= \{\, a + b \,\} \\ X_5 &= \varnothing \end{aligned}$$

Es gilt $f_4(X_4) = \varnothing \supseteq \varnothing = X_5$.
Außerdem ist die Worklist nun leer.

Damit terminiert der Algorithmus. □

2.4 Join-over-all paths

Bisher haben wir die Datenflussanalyse durch Lösen des Gleichungssystems, das von einem Datenflusssystem S induziert wird, durchgeführt.

Problem:

- Die Fixpunktlösung ist manchmal unpräzise.
- Sie bildet den Join der Datenflussinformationen in jedem Berechnungsschritt

$$X_{b_2}^{LFP,iter2} = f_{b_1}\left(X_{b_1}^{LFP,iter1}\right) \sqcup f_{b_0}\left(X_{b_0}^{LFP,iter1}\right) \ .$$

- Damit sind die zukünftigen Berechnungen von dieser zwischenzeitlichen Abstraktion betroffen und werden ebenfalls unpräzise (und durch weitere Abstraktion noch unpräziser).

Idee: Abstrahiere (Join) nur am Ende der Berechnung.

Definition 2.18
Sei $S = (G, (D, \leq), i, f)$ mit $G = (B, E, F)$ ein Datenflusssystem. Für jeden Block $b \in B$ sei

$$\text{paths}(b) := \{\pi = b_1 \dots b_{n-1} \in B^* \mid k \geq 1, b_1 \in E, b_k = b, (b_i, b_{i+1}) \in F \ \forall \ 1 \leq i < k\}$$

die Menge der Pfade, die von einem Extremalknoten zu b führen.

Gegeben einen Pfad $\pi = b_1 \dots b_{k-1} \in \text{paths}(b)$, definieren wir die Transferfunktion $f_\pi : D \to D$ mittels

$$f_\pi := f_{b_{k-1}} \circ \dots \circ f_{b_1} \circ \text{id} \ .$$

(Dementsprechend also $f_\epsilon = \text{id}$.) □

Definition 2.19
Die *join-over-all-paths (JOP)-Lösung von S* ist

$$\text{JOP}(S) = (X_{b_1}^{JOP}, \dots, X_{b_{|B|}}^{JOP})$$

mit

$$X_b^{JOP} := \cup\{f_\pi(i) \mid \pi \in \text{paths}(b)\} \ .$$ □

Beispiel 2.20 (Fixpunktlösung vs. JOP-Lösung)
Betrachte das Programm c

```
if [z>0] then
    [x:=2]^2;
    [y:=3]^3;
else
    [x:=3]^4;
    [y:=2]^5;
end
[z:=x+y]^6;
[skip]^7;
```

Wir führen eine Constant-Propagation-Analyse durch.

Sei S das Datenflusssystem.

Die *Fixpunktlösung von S* lautet

$$\begin{aligned}
X_1^{LFP} &= (\bot, \bot, \bot) \\
X_2^{LFP} &= (\bot, \bot, \bot) \\
X_3^{LFP} &= (2, \bot, \bot) \\
X_4^{LFP} &= (\bot, \bot, \bot) \\
X_5^{LFP} &= (3, \bot, \bot) \\
X_6^{LFP} &= (2, 3, \bot) \sqcup (3, 2, \bot) \\
&= (\top, \top, \bot) \\
X_7^{LFP} &= (\top, \top, \top)
\end{aligned}$$

Die JOP-Lösung von S für Block 7 liefert:

$$\begin{aligned}
X_7^{JOP} &= f_{b_1 b_2 b_3 b_6}(\bot, \bot, \bot) \cup f_{b_1 b_4 b_5 b_6}(\bot, \bot, \bot) \\
&= (2, 3, 5) \cup (3, 4, 5) \\
&= (\top, \top, 5)
\end{aligned}$$

□

Bemerkung

Typischerweise ist paths(b) unendlich. Es ist daher nich klar, ob X_b^{JOP} berechnet werden kann.

Tatsächlich ist JOP oft zu gut, um berechenbar zu sein. □

Satz 2.21 (Kann, Ullman 1977)

Die JOP-Lösung für Constant-Propagation ist nicht berechenbar. □

Beweis

Reduktion (einer modifizierten Version) von Posts Korrespondenzproblems auf die Berechnung der JOP-Lösung.

Eingabe von PCP: Paare $(u_1, v_1), \ldots, (u_n, v_n)$ von Worten über $\{0, \ldots, 9\}$

Frage:

- Gibt es eine Indexfolge $i_1 \ldots i_k$ in $\{1, \ldots, n\}$ mit $i_1 = 1$ (dies macht das Problem nicht leichter) so dass

  $$u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$$

  Zur Reduktion sei $|u|$ für die Länge des Wortes $u \in \{0, ..., 9\}^*$ und $[\![u]\!]$ die von u dargestellte natürliche Zahl.

  Betrachte folgendes Programm:

```
x := u_1;
y := v_1;
while (true) do
    if (true) then
        x := x*10^|u_1| + [[ u_1 ]];
        y := y*10^|v_1| + [[ v_1 ]];
    else
        ...
        // analoger Code fuer u_2 bis u_{n-1}
```

```
            ...
            if (.true) then
                x := x*10^|u_n| + [[ u_n ]];
                y := y*10^|v_n| + [[ v_n ]];
            else
                skip;
            end // endif zu u_n
            ...
        end // endif zu u_1
    end // zu while
    b = [z:=sign((x − y)^2)]^b // 1 wenn x ≠ y, 0 sonst
    b' = [skip]'
```

- Beachte: Constant-Propagation ist eine syntaktische Analyse, d.h. die Schleifenbedingungen und die If-Bedingungen spielen keine Rolle.
- Falls PCP eine Lösung hat, dann gibt es einen Pfad, der an Block b $x = y$ liefert und damit bei b' $z = 0$ liefert
- Ansonsten ist z konstant 1 bei b'.

Es gilt also $X_{b'}^{JOP} = 1$ gdw. PCP hat keine Lösung. ■

Allerdings ist X_b^{LFP} immer eine sichere Approximation von X_b^{JOP}.

Satz 2.22 (Zusammenhang zwischen LFP und JOP)
Sei $S = (G, (D, \leq), i, f)$ ein Datenflusssystem mit $G = (B, E, F)$.

Sei $\text{lfp}(g_S) = (X_{b_1}^{LFP}, \ldots, X_{b_{|B|}}^{LFP})$ die Fixpunktlösung und sei $\text{JOP}(S) = (X_{b_1}^{JOP}, \ldots, X_{b_{|B|}}^{JOP})$ die JOP-Lösung.

(1) Für alle $b \in B$ gilt $X_b^{JOP} \leq X_b^{LFP}$

(2) Falls alle Transferfunktionen *distributiv* sind (distributive Frameworks), gilt sogar

$$X_b^{JOP} = X_b^{LFP}$$

□

Beweis
Zu (1): Definiere

$$X_b^{JOP,n} := \sqcup\{f_\pi(i) \mid \pi \in \text{paths}(b) \text{ mit } |\pi| \leq n\}$$

Dann gilt:

$$X_b^{JOP} = \sqcup\{X_b^{JOP,n} \mid n \in \mathbb{N}\}$$

Zeige nun:

$$X_b^{JOP,n} \leq X_b^{LFP} \text{ f.a. } n \in \mathbb{N}$$

Dann folgt

$$\sqcup\{X_b^{JOP,n} \mid n \in \mathbb{N}\} \leq X_b^{LFP}$$

Zeige: $X_b^{JOP,n} \leq X_b^{LFP}$ f.a. $n \in \mathbb{N}$ durch Induktion nach n (simultan für alle Blöcke b).

IA: Falls es keinen Pfad der Länge 0 gibt, so ist $X_b^{JOP,n} = \sqcup\varnothing = \bot \leq X_b^{LFP}$.

Falls es einen Pfad der Länge 0 gibt, gilt $b \in$ Extremalknoten und somit

$$f_\epsilon(i) = \text{id}(i) = i = X_b^{LFP} \ .$$

IV: Angenommen die Behauptung gilt für $X_b^{JOP,n}$

IS: Dann gilt

$$
\begin{aligned}
& X_b^{LFP} \\
& = \sqcup\{f_{b'}(X_{b'}^{LFP}) \mid (b',b) \in F\} \\
\text{(IV und Monotonie)} \quad & \geq \sqcup\{f_{b'}(X_{b'}^{JOP,n}) \mid (b',b) \in F\} \\
\text{(Def. JOP)} \quad & = \sqcup\{f_{b'}(\sqcup\{f_\pi(i) \mid |\pi| < n, \pi \in \text{paths}(b')\}) \mid (b',b) \in F\} \\
f(a) \sqcup f(b) \leq f(a \sqcup b) \quad & \geq \sqcup\{\sqcup\{f_{b'}(f_\pi(i)) \mid |\pi| < n, \pi \in \text{paths}(b')\} \mid (b',b) \in F\} \\
& = \sqcup\{f_{\pi'}(i) \mid |\pi'| \leq n+1, \pi' \in \text{paths}(b)\}
\end{aligned}
$$

Zu (b): Hausaufgabe. ■

3 Interprozedurale Datenflussanalyse

In diesem Abschnitt erweitern wir die bisherige Analysemethode so, dass auch Programme mit Funktionen behandelt werden können.

Ziel: Datenflussanalyse für rekursive Programme.

Problem: Berücksichtigung der Call-Return-Beziehung bei Prozeduraufrufen (Return zum richtigen Call-Block im Kontrollflussgraphen).

Idee: Berechne *JOVP-Lösung* (join-over-all-valid-paths). Dazu betrachten wir zwei Techniken:

- *Procedure-Summaries*: Berechne den Effekt $f_p : D \to D$ einer Prozedur p.
- *Call-Strings*: Führe eine abstrakte Version des Stacks als Datenflussinformation mit.

3.1 Rekursive Programme

Definition 3.1 (Rekursives Programm)
Ein *rekursives Programm* ist definiert als Folge von Prozeduren:

$$\begin{aligned} \mathit{prog} ::= \quad & \texttt{proc}\ [\mathit{main}()]^{\mathit{entry}}\ \texttt{begin}\ c\ [\texttt{end}]^{\mathit{exit}} \\ \mid\ & \mathit{prog}\ \ \texttt{proc}\ [p()]^{\mathit{entry}}\ \texttt{begin}\ c\ [\texttt{end}]^{\mathit{exit}} \\ c ::= \quad & [p()]^{\mathit{call}}_{\mathit{return}} \\ \mid\ & \text{... Befehle wie bisher} \end{aligned}$$

wobei *main* der Eintrittspunkt des Programms und p eine Prozedur ist.

Es gilt:

- Prozeduren haben keine Parameter und keine Return-Werte.
- Alle Variablen in `main()` sind *global*, d.h. auch sichtbar innerhalb anderer Prozeduren. (Damit lassen sich Parameter und Return-Werte nachbilden.)
- Prozedur `main()` kann nicht explizit aufgerufen werden, sondern wird zu Beginn des Programms implizit ausgeführt.
- Prozeduren können *lokale* Variablen definieren.
- Es wird angenommen, dass alle Prozeduren verschieden heißen.
- Entry- und Exit-Blöcke garantieren, dass es einen Anfangs- und Endblock gibt.
- Blöcke $[\text{p}()]^{\mathit{call}}_{\mathit{return}}$ haben zwei Label. □

Auch rekursive Programme werden als Kontrollflussgraph dargestellt:

- Für jede Prozedur p in *prog*, sei
$$G_p := (B_p, E_p, F_p)$$
der Kontrollflussgraph, welcher wie bisher konstruiert wird.

- Dann ist

$$G_{prog} := (\biguplus_{p \in prog} B_p, \biguplus_{p \in prog} E_p, \biguplus_{p \in prog} F_p, IF)$$

der *Kontrollflussgraph von prog.*

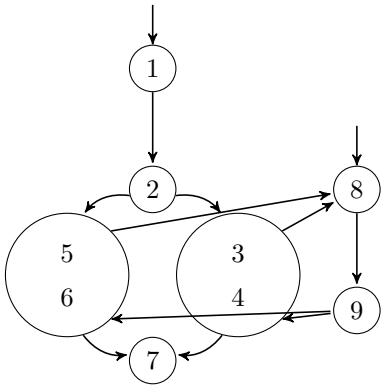
Dabei ist der *interprozedurale Flow IF* definiert als

$$IF := \{(call, entry, exit, return) \mid \text{ eine Prozedur von } prog \text{ enthält } [p()]^{call}_{return} \text{ mit } \texttt{proc}\ [p()]^{entry}\ \texttt{begin}\ c\ [\texttt{end}]^{exit}\}$$

Beispiel 3.2

Warum brauchen wir die 4-Tupel als interprozeduralen Flow?

```
proc[main()]^1 begin
    if[(x)]^2 then
        [p()]^1_4;
    else
        [p()]^5_6;
    end
    [end]^7
proc[p()]^8 begin
[end]^9
```



- 123894 ist *gültiger* Pfad
- 123896 ist *kein gültiger* Pfad
- ähnliche Probleme treten mit dem Rücksprung aus geschachtelten Rekursionen auf:

□

Definition 3.3 (Gültige Pfade)
Sei $G = (B, E, F, IF)$ ein Kontrollflussgraph.

Dann ist die *Menge der gültigen Pfade von l_1 zu l_2*, definiert durch die kontextfreie Grammatik (CFG)

$$\Gamma = (\underbrace{\{N_{l,l'} \mid l, l' \in Lab\}}_{\text{nicht-Terminale}}, \underbrace{Lab}_{\text{Terminale}}, P, \underbrace{N_{l_1,l_2}}_{\text{Startsymbol}})$$

mit Produktionen

$$\begin{aligned} N_{l,l} &\to l \\ N_{l,l''} &\to l \cdot N_{l',l''} \text{ mit } (l, l') \in F \\ N_{call,l} &\to call \cdot N_{entry,exit} \cdot N_{return,l} \text{ mit } (call, entry, exit, return) \in IF \end{aligned}$$ □

Definition 3.4 (JOVP-Lösung)
Sei $S = (G, (D, \leq), i, f)$ ein (rekursives) Datenflusssystem.

Die *JOVP-Lösung (join-over-all-valid-paths)* ist

$$\text{JOVP}(S) = (X_1^{\text{JOVP}}, \ldots, X_{|B|}^{\text{JOVP}}) \in D^{|B|}$$

mit

$$X_b^{\text{JOVP}} = \sqcup \{f_\pi(i) | \pi \in \text{validpaths}(l_{min}, l_{b'}) \text{ mit } (b', b) \in F\}$$

//Alle Pfade bis zu und einschließlich dem Vorgängerblock von b. □

Korollar 3.5

1. $\text{JOVP}(S) \leq \text{JOP}(S)$

   (Hier werden *alle* Pfade betrachtet, Call-Return-Beziehungen nicht berücksichtigt.)
2. $\text{JOVP}(S)$ ist nicht berechenbar, da die intraprozedurale Analyse ein Spezialfall ist. □

3.2 Der funktionale Ansatz

Ziel: Fixpunktgleichungen, die ungültige Pfade vermeiden.

Idee:

- Berechne Transferverhalten von Prozeduren (*Procedure-Summary*).
- Vermeide dann Analyse innerhalb von Prozedurrümpfen

Idee (genauer):

- Jeder gewöhnliche Block $b = [x := a]^l$ hat eine Transferfunktion

  $$f_b : D \to D$$

  die die Datenflussinformation ändert.
- Angenommen für Prozedur p hätten wir eine Transferfunktion

  $$f_p : D \to D$$

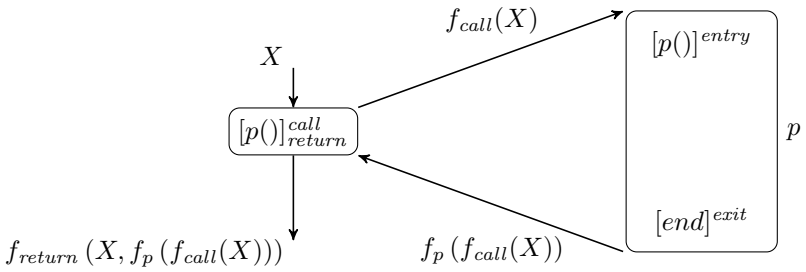
  die das Verhalten von p zusammenfasst.
- Dann ließe sich ein Block

  $$b = [p()]^{call}_{return}$$

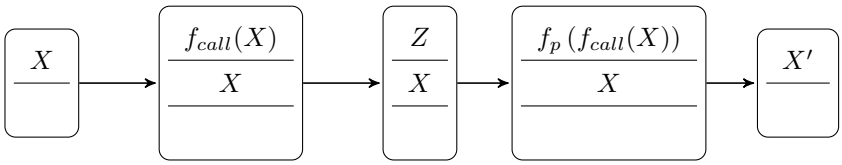
  durch die Transferfunktion

  $$f_b(X) = f_{return}(\, X,\, f_p(\, f_{call}(X)\,)\,)$$

  darstellen.
- Sowohl f_{call} als auch f_{return} sind als Teil des Datenflusssystems gegeben.
- Intuition der Funktionen:



  - $f_{call} : D \to D$
    - Initialisiere Datenflusswert bei Eintritt in die Prozedur
    - abhängig vom aktuellen Datenflusswert
  - $f_{return} : D \times D \to D$
    - Kombiniere Datenflusswert am Ende der Prozedur (2ter Parameter)
    - mit Datenflusswert bei Prozedureintritt.
- Warum ist X Parameter von f_{return}?



– Call berechnet neuen Top-of-Stack

– übliche Operationen ändern den *nur* obersten Stackinhalt.

– Return kombiniert den aktuellen Top-of-Stack mit vorherigen Top-of-Stacks.

Problem:

- f_p vor der Analyse nicht bekannt
- Bestimme f_p so, dass $f_p \geq f_\pi$ für alle $\pi \in \text{validpaths}(entry, exit)$

Ansatz: Fixpunktberechnung auf dem vollständigen Verband der monotonen Funktionen in $D \to D$

$$(\text{MonFun}(D \to D), \leq) \text{ mit } f \leq g, \text{ falls } f(d) \leq g(d) \text{ für alle } d \in D$$

Definition 3.6
Sei $S = (G, (D, \leq), i, f)$ das Datenflusssystem eines rekursiven Programms. Dann induziert S das folgende *Summary-Gleichungssystem*:

$$\begin{aligned} Y_{entry} &= id \\ Y_b &= \sqcup \{ f_{b'} \circ Y_{b'} \mid (b', b) \in F \} \\ Z_p &= f_{exit} \circ Y_{exit} \end{aligned}$$

mit

$$\begin{aligned} f_{b'} &= \text{callret}(Z_q), \text{ falls } b' = [q()]^{call}_{return} \\ f_{b'} &= \text{ wie in Familie } f \text{ von } S \text{ angegeben, sonst} \end{aligned}$$

und

$$\begin{aligned} \text{callret}(Z) &: D \to D \\ \text{callret}(Z)(d) &= f_{return}(d, Z(f_{call}(d))) \ . \end{aligned}$$

□

Bemerkung
Man beachte, dass das Summary-Gleichungssystem eine Variable Y_b für jeden Block b und eine Variable Z_p für jede Prozedur enthält. Eine Variable Y_b fasst dabei den Effekt der Prozedur, in der Block b vorkommt, vom *entry* Block bis b zusammen. Dementsprechend, spiegelt Z_p den Transfereffekt der gesamten Prozedur wieder.

Satz 3.7
Sei $(Y_1, \ldots, Y_{|B|}) \in (\text{MonFun}(D \to D))^{|B|}$ kleinste Lösung des Summary-Gleichungssystems.

(1) Sei b Block der Prozedur p, dann gilt $Y_b \geq f_\pi$ für jeden gültigen Pfad von entry(p) bis zu b.

(2) $Z_p \geq f_\pi$ für alle $\pi \in \text{validpaths}(\text{entry}(p), \text{exit}(p))$ □

Problem: Monotone Funktionen müssen effektiv dargestellt werden.

- Falls D endlich: nutze Wertetabelle.
- Falls D unendlich: problematisch.

Bemerkung (Modellierung Summary-Gleichungssystems)
Blöcke mit Prozeduraufrufen haben zwei Labels, allerdings führen wir nur eine Variable für diesen Block ein. (Vgl. bisher: eins-zu-eins Korrespondenz zwischen Labeln und Variablen im Gleichungssystem.)

Im folgenden diskutieren wir eine alternative Modellierungsmöglichkeit und zeigen die Äquivalenz zur gewählten Definition des Summary-Gleichungssystems.

Dazu betrachte folgendes Programm:

```
proc [p()]^1 begin
   [x := 2]^2;
   [q()]_6^5;
   [z := a]^5;
   ...
[end];
```

Seien f_{call}, f_{return} die Transferfunktionen, die $b = [q()]_4^3$ zugeordnet sind.

Summary-Gleichungssystem Die oben gewählte Definition führt für einen Prozeduraufruf nur eine Variable Y_b ein, welche wir mit dem Call-Label identifizieren.

Im Beispiel also: $Y_b = Y_3$ und $Y_5 = f_3 \circ Y_3 = \text{callret}(Z_q) \circ Y_3$.

Alternative (äquivalent) Es werden zwei Variablen zu Blöcken, die Funktionsaufrufe darstellen, assoziiert. Y_3 erhält die Transferfunktion

$$f_3 = Y_q \circ f_{call} \ .$$

Für Y_4 erhält man also

$$Y_4 = f_3 \circ Y_3 = Z_q \circ f_{call} \circ Y_3 \ .$$

Als Transferfunktion für Y_4 erhält man $f_4 = f_{return}$. Damit ergibt sich für Block 5 folgende Gleichung

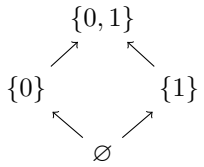
$$\begin{aligned} Y_5 &= f_{return}\,(Y_3(\cdot), Y_4(\cdot)) \\ &= f_{return}\,(Y_3(\cdot), Z_q(f_{call}(Y_3(\cdot)))) \\ &= \text{callret}(Z_q) \circ Y_3 \end{aligned}$$

und somit das selbe Ergebnis wie im vorherigen Fall. □

Beispiel 3.8 (Reachable-Values-Analyse)

Betrachte die Prozedur *pos*(), welche eine globale boolsche Variable x manipuliert, und den vollständigen Verband der Wertemengen für x.

```
proc [pos()]^1 begin
    if [x = 0]^2 then
        [assert (x=0)]^3;
        [x := ¬x]^4;
        [pos()]_6^5;
    else
        [assert (x =1)]^7;
    end
[end]^8;
```



Wir verwenden folgende Transferfunktionen:

$$\begin{array}{lll} f_1 = \text{id} & f_4 = f_{invert} & f_7 = f_{get1} \\ f_2 = \text{id} & f_{callpos} = \text{id} & f_8 = \text{id} \\ f_3 = f_{get0} & f_{returnpos}(d_1, d_2) = d_2 & \end{array}$$

Dabei drückt f_{get0} aus, dass nach einem `assert(x=0)`, der Wert für x nicht 1 sein kann. Analog für f_{get1}. Die Funktion f_{invert} kehrt den Wert um. Des Weiteren definieren wir die Hilfsfunktionen f_{make0}, $f_{invert0}$, und $f_\perp$ Die Werte all dieser Funktionen sind durch die folgende Tabelle gegeben.

X	$id(X)$	$f_\perp(X)$	$f_{get0}(X)$	$f_{get1}(X)$	$f_{invert}(X)$	$f_{make0}(X)$	$f_{invert0}(X)$
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$\{0\}$	$\{0\}$	$\varnothing$	$\{0\}$	$\varnothing$	$\{1\}$	$\{1\}$	$\{1\}$
$\{1\}$	$\{1\}$	$\varnothing$	$\varnothing$	$\{1\}$	$\{0\}$	$\{1\}$	$\varnothing$
$\{0,1\}$	$\{0,1\}$	$\varnothing$	$\{0\}$	$\{1\}$	$\{0,1\}$	$\{1\}$	$\{1\}$

Beachte, dass die folgenden Beziehungen gelten:

$$f_{invert} \circ f_{get0} = f_{invert0} \qquad f_{get1} \circ f_{invert0} = f_{invert0}$$
$$f_{make0} \circ f_{invert0} = f_{invert0} \qquad f_{invert0} \sqcup f_{get1} = f_{make0}$$

Das **Summary-Gleichungssystem** lautet:

$$Y_2 = id \qquad Y_6 = Z_{pos} \circ Y_5$$
$$Y_3 = id \circ Y_2 \qquad Y_7 = id \circ Y_2$$
$$Y_4 = f_{get0} \circ Y_3 \qquad Y_8 = (id \circ Y_6) \sqcup (f_{get1} \circ Y_7)$$
$$Y_5 = f_{inv} \circ Y_4 \qquad Z_{pos} = id \circ Y_8$$

Löse das Gleichungssystem durch Fixpunktiteration:

Iteration	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Z_{pos}
$g_S^0(\perp^9)$	$f_\perp$	$f_\perp$	$f_\perp$	$f_\perp$	$f_\perp$	$f_\perp$	$f_\perp$	$f_\perp$
1	id	$f_\perp$	$f_\perp$	$f_\perp$	$f_\perp$	$f_\perp$	$f_\perp$	$f_\perp$
2	id	$f_\perp$	$f_\perp$	$f_\perp$	id	$f_\perp$	$f_\perp$	$f_\perp$
3	id	id	f_{get0}	$f_\perp$	$f_\perp$	id	f_{get1}	$f_\perp$
4	id	id	f_{get0}	$f_{invert0}$	$f_\perp$	id	f_{get1}	f_{get1} [1]
5	id	id	f_{get0}	$f_{invert0}$	$f_{invert0}$	id	f_{get1}	f_{get1}
6	id	id	f_{get0}	$f_{invert0}$	$f_{invert0}$	id	f_{make0}	f_{get1} [2]
7	id	id	f_{get0}	$f_{invert0}$	$f_{invert0}$	id	f_{make0}	f_{make0} [3]
8	id	id	f_{get0}	$f_{invert0}$	$f_{invert0}$	id	f_{make0}	f_{make0} [4] $\surd$

□

Beobachtung

Sind die Transferfunktionen f_p für alle Prozeduren p berechnet, so lässt sich auch für rekursive Datenflusssysteme ein Gleichungssystem aufstellen. Sei dazu $S = (G, (D, \leq), i, f)$ mit $G = (B, E, F)$. Das *Gleichungssystem zur Datenflussanalyes* ist:

$$X_b = \sqcup \{ f_{b'}(X_{b'}) \mid (b', b) \in F \}, \text{ falls } b \notin E$$
$$X_b = \sqcup \{ f_{callp}(X_{b'} \mid (b', b, *, *) \in IF \}, \text{ falls } b \in E_p \setminus E_{main}$$
$$X_b = i, \text{ falls } b \in E_{main}$$

[1] $f_\perp \sqcup f_{get1} = f_{get1}$
[2] $f_{get1} \circ f_\perp \sqcup f_{get1} = f_\perp \sqcup f_{get1} = f_{get1}$
[3] $f_{get1} \circ f_{invert0} \sqcup f_{get1} \circ id = f_{invert0} \sqcup f_{get1} = f_{make0}$
[4] $f_{make0} \circ f_{invert0} \sqcup f_{get1} = f_{invert0} \sqcup f_{get1} = f_{make0}$

mit

$$f_{b'}(X_{b'}) = \begin{cases} \text{wie in } f \text{ angegeben}, & \text{falls } b' \text{ gewöhnlicher Block} \\ f_{returnp}(X_{b'}, f_p(f_{callp}(X_{b'}))), & \text{falls } b' = [p()]^{callp}_{returnp} \end{cases} .$$

Beachte hier, dass der Datenfluss innerhalb der Prozeduren berechnet wird. Mit den zuvor berechneten Summaries haben wir lediglich das Transferverhalten berechnet. Die aktuellen Datenflusswerte innerhalb der Prozeduren können erst jetzt berechnet werden. (Falls nur Datenflusswert innerhalb der Prozedur `main` von Interesse ist, kann man das Gleichungssystem entsprechend vereinfachen.) □

Satz 3.9 (Sharir & Pnueli '81)
Sei $S = (G, (D, \leq), i, f)$ ein rekursives Datenflusssystem. Sei $\text{lfp}(g_S) = (X_1^{LFP}, \ldots, X_{|B|}^{LPF})$ die Fixpunktlösung des assoziativen Gleichungssystems und sei $\text{JOVP}(S) = (X_1^{JOVP}, \ldots, X_{|B|}^{JOVP})$ die JOVP-Lösung.

(a) Für alle $b \in B$ gilt $X_b^{JOVP} \leq X_b^{LFP}$.

(b) Falls alle Transferfunktionen distributiv sind, gilt sogar $X_b^{JOVP} = X_b^{LFP}$ für alle $b \in B$. □

3.3 Der Call-String-Ansatz

Beobachtung
Der funktionale Ansatz ist kontextinsensitiv, d.h. er berücksichtigt *keine* Information über den Kontext, in dem eine Prozedur aufgerufen wird. Das macht die Analyse gegebenenfalls unpräzise.

Zum Beispiel, falls $p()$ von $q()$ aus aufgerufen wird, könnte x immer 1 sein. □

Ziel: Entwickle eine kontextsensitive interprozedurale Datenflussanalyse.

Ansatz: Reichere die Domäne der Datenflusswerte um Informationen über den Stackinhalt an.

Sei dazu $(D, \leq)$ der vollständige Verband der Datenflusswerte, und Γ die Menge der Prozedurnamen im zu analysierenden Programm. Nutze die *neue Domäne:*

$$(\Gamma^* \to D, \leq^*)$$

mit $cs_1 \leq^* cs_2$, falls $cs_1(\alpha) \leq cs_2(\alpha)$ für alle $\alpha \in \Gamma^*$. Es werden *Call-Strings* also Datenflussinformationen zugewiesen.

Es lässt sich prüfen, dass $(\Gamma^* \to D, \leq^*)$ wieder vollständiger Verband ist.

Ansatz (formal): Um ein gegebenes Datenflusssystem $S = (G, (D, \leq), i, f)$, unter Berücksichtigung von Call-Strings zu analysieren, modifiziere (1) den Initialwert, und (2) die Transferfunktionen.

ad 1) Aus $i \in D$ wird

$$\begin{aligned} cs_i &: \Gamma^* \to D \text{ mit} \\ cs_i(\epsilon) &:= i \text{ und} \\ cs_i(\alpha) &:= \bot \text{ für } \epsilon \neq \alpha \in \Gamma^* \end{aligned}$$

ad 2) Aus $f_b : D \to D$ wird

$$\widetilde{f}_b : (\Gamma^* \to D) \to (\Gamma^* \to D)$$

mit

$$\begin{aligned} \widetilde{f}_b(cs) &:= f_b \circ cs && \text{für gewöhnlichen Block } b \\ \widetilde{f}_{call_p}(cs)(\gamma.p) &:= cs(\gamma) && \\ \widetilde{f}_{call_p}(cs)(\alpha) &:= \bot && \text{für } \gamma.p \neq \alpha \in \Gamma^* \\ \widetilde{f}_{return_p}(cs)(\gamma) &:= f_{return_p}(cs(\gamma.p)) && \end{aligned}$$

Definition 3.10
Sei $S = (G, (D, \leq), i, f)$ ein rekursives Datenflusssystem.

Dann ist das induzierte *Call-String-Gleichungssystem*

$$\begin{aligned} X_b &:= cs_i, \text{ falls } b \in E_{main} \\ X_b &:= \sqcup \Big\{ \widetilde{f}_{b'}(X_{b'}) \mid (b', b) \in F \text{ oder} \\ &\quad (*, *, b', b) \in IF \text{ und } \widetilde{f}_{b'} = \widetilde{f}_{return_p} \text{ oder} \\ &\quad (b', b, *, *) \in IF \text{ und } \widetilde{f}_{b'} = \widetilde{f}_{call_p} \Big\} \end{aligned}$$ □

Satz 3.11
Die Call-String-Lösung überapproximiert JOVP(S). □

Problem: Γ^* ist unendlich.

Ansatz:

- *Bounded Call-Strings*:
  - Stelle nur obersten n Elemente des Stacks dar, nutze also Call-Strings aus $\Gamma^{\leq n} := \cup_{i=0}^{n} \Gamma^i$.
  - Es gibt Sätze über ausreichende Call-String-Länge, die exakte Analyse garantiert.
- *Regular Abstraction*: Anstelle von $\Gamma^{\leq n}$, nutze endliche Automaten $A^{\leq n}$ der Größe $\leq n$ um den Stack-Inhalt darzustellen.

4 Semantik

Ziel: Formale Diskussion der Semantik von while-Programmen; unter anderem, um später ein formales Framework für Analysen definieren zu können, welches die Semantik von Programmen (korrekt) approximiert (anstelle von *ad hoc* Transferfunktionen wie bisher).

4.1 Strukturelle Operationelle Semantik (SOS)

Ziel: Definiere *operationelle* Semantik.

Idee:

- *Zustände* eines Programmes haben (syntaktische) Struktur: Ein Programm ist Komposition atomarer Elemente mittels einer Menge von Operatoren
- Damit lassen sich *Beweissysteme* (Kalküle) nutzen, um das Verhalten von Zuständen zu definieren: Transitionen existieren gdw. sie im Beweissystem herleitbar sind.
- Technisch nutzt das Beweissystem *Induktion nach der Struktur von Zuständen:*
  - *Axiome* definieren Transitionen von atomaren Elementen.
  - *Beweisregeln* definieren die Transition zusammengesetzter Zustände über die Transitionen der Operanden.
- Vorteile:
  - Einfachheit und Eleganz
  - Möglichkeit Eigenschaften von Transitionen über Induktion entlang der Ableitung herzuleiten.

Ansatz: Zwei Arten der strukturellen Semantik.

- *Small-Step Semantik*: spezifiziert den Effekt jeder einzelnen Operation.
- *Big-Step Semantik*: fasst den Effekt einer kompletten Programmausführung zusammen.

Die Semantik ordnet jedem syntaktischen Ausdruck eine *Bedeutung* zu. Dabei ist die Bedeutung ein Element eines *semantischen Bereichs*. Formal ist der semantische Bereich gegeben als *Sig-Struktur*.

Definition 4.1 (Signatur, Struktur)
Eine (logische) *Signatur* ist ein Tupel $Sig = (Func, Pred)$ mit

- *Funktionssymbolen* $Func$, und
- *Prädikatssymbolen* $Pred$.

Jedes Funktions- und Prädikatssymbol hat eine Stelligkeit $n \in \mathbb{N}$. Dementsprechend schreiben wir $f/_n \in Func$ bzw. $p/_n \in Pred$, falls Funktionssymbol f bzw. Prädikatssymbol p Stelligkeit n hat. Funktionen und Prädikate mit Stelligkeit 0 heißen Konstanten.

Eine (logische) *Struktur* der Signatur Sig, auch Sig-Struktur genannt, ist ein Tupel $\mathcal{S} = (D, \mathcal{I})$ mit

- einer Wertemenge D, und

- einer Interpretation $\mathcal{I}$, welche jedem Funktions- und Prädikatssymbol eine tatsächliche Funktion zuordnet:

$$\begin{aligned} \mathcal{I}(f) &: D^n \to D && \text{für alle } f/_n \in \mathit{Func} \\ \mathcal{I}(p) &: D^n \to \mathbb{B} && \text{für alle } p/_n \in \mathit{Pred} \end{aligned}$$ □

Bemerkung (Sig-Struktur von While-Programmen)
Die Signatur von While-Programmen ist $\mathit{Sig} = (\mathit{Func}, \mathit{Pred})$ mit $\mathit{Func} = \{ +/_2, -/_2, */_2 \} \cup \{ k/_0 \mid k \in \mathbb{Z} \}$, und $\mathit{Pred} = \{ > /_2 \}$.

Die *Sig-Struktur* von While-Programmen ist $\mathcal{S} = (\mathbb{Z}, \mathcal{I})$, wobei $\mathcal{I}(+)$, $\mathcal{I}(-)$, $\mathcal{I}(*)$, und $\mathcal{I}(>)$ die natürliche Interpretation der Symbole ist.

Beachte: Das Prädikat $=$ ist nicht in *Pred* enthalten. Dies folgt der Konvention der *Prädikatenlogik mit Gleichheit*: wir definieren die Semantik von $=$ *fest* als Identität auf der Wertemenge, lassen also keine Interpretation des Symbols $=$ zu. Diese Konvention findet man auch in Programmiersprachen: In Java, zum Beispiel, entspricht `==` immer Identität auf Objekten, wobei in C++ der Operator `==` überladen werden kann, und somit eine *beliebige* Bedeutung erhalten kann.

Ebenso werden die Junktoren $\neq$, $\wedge$, und $\vee$ nicht interpretiert. Dies liegt allerdings daran, dass sie boolsche Ausdrücke, und nicht wie die Funktions- bzw. Prädikatssymbole arithmetische Ausdrücke, verknüpfen.□

Das Verhalten eines Programmes hängt von der *Belegung der Variablen* ab, auch *Zustand* oder *State* genannt, also einem $\sigma \in \mathit{State} := \mathbb{Z}^{\mathrm{Vars}}$. Wir können σ somit als Belegung für Variablen auffassen:

$$\sigma : \mathrm{Vars} \to \mathbb{Z} \ .$$

Wir schreiben $\sigma[x \mapsto k]$, mit $x \in \mathrm{Vars}$ und $k \in \mathbb{Z}$, um die Belegung von x in σ zu k zu ändern:

$$\sigma[x \mapsto k](x') = \begin{cases} \sigma(x') & \text{falls } x' \neq x \\ k & \text{sonst} \end{cases}$$

Die Semantik von arithmetischen Ausdrücken bezüglich der *Sig*-Struktur $\mathcal{S}$ für while-Programme ist eine Funktion

$$\mathcal{S}[\![\, a \,]\!] : \mathbb{Z}^{\mathrm{Vars}} \to \mathbb{Z}$$

die wie folgt induktiv definiert ist:

$$\begin{aligned} \mathcal{S}[\![\, x \,]\!](\sigma) &= \sigma(x) && \text{falls } x \in \mathrm{Vars} \\ \mathcal{S}[\![\, f(a_1, \ldots, a_n) \,]\!](\sigma) &= \mathcal{I}(f)(\mathcal{S}[\![\, k \,]\!](a_1), \ldots, \mathcal{S}[\![\, k \,]\!](a_n)) && \text{für } f/_n \in \mathit{Func} \end{aligned}$$

Analog ist die Semantik für boolsche Ausdrücke bezüglich $\mathcal{S}$ eine Funktion

$$\mathcal{S}[\![\, b \,]\!] : \mathbb{Z}^{\mathrm{Vars}} \to \mathbb{B}$$

wie folgt:

$$\begin{aligned} \mathcal{S}[\![\, a_1 = a_2 \,]\!](\sigma) = 1 &\iff \mathcal{S}[\![\, a_1 \,]\!](\sigma) = \mathcal{S}[\![\, a_2 \,]\!](\sigma) \\ \mathcal{S}[\![\, p(a_1, \ldots, a_n) \,]\!](\sigma) &= \mathcal{I}(p)(\mathcal{S}[\![\, a_1 \,]\!](\sigma), \ldots, \mathcal{S}[\![\, a_n \,]\!](\sigma)) && \text{für } p/_n \in \mathit{Pred} \\ \mathcal{S}[\![\, \neg b \,]\!](\sigma) &= 1 - \mathcal{S}[\![\, b \,]\!](\sigma) \\ \mathcal{S}[\![\, b_1 \wedge b_2 \,]\!](\sigma) &= \mathit{min}(\mathcal{S}[\![\, b_1 \,]\!](\sigma), \mathcal{S}[\![\, b_2 \,]\!](\sigma)) \\ \mathcal{S}[\![\, b_1 \vee b_2 \,]\!](\sigma) &= \mathit{max}(\mathcal{S}[\![\, b_1 \,]\!](\sigma), \mathcal{S}[\![\, b_2 \,]\!](\sigma)) \end{aligned}$$

4.1.1 Small-Step Semantik

Ziel: Definiere die Semantik des Programms für jede einzelne Anweisung.

Ansatz:

- Definiere Beweisregeln, die auf Konfigurationen angewendet werden, bis ein finaler Zustand abgeleitet ist.
- Wir schreiben Regelanwendungen als Transition zwischen Konfigurationen:
  - $cf \to cf'$ bedeutet, dass cf' in einem Schritt aus cf abgeleitet werden kann,
  - $cf \to^n cf'$ bedeutet, dass cf' in genau n Schritten aus cf abgeleitet werden kann (für $n = 0$ erhalten wir $cf = cf'$), und
  - $cf \to^* cf'$ bedeutet, dass es ein n gibt sodass $cf \to^n cf'$ gilt.
- Verwenden spezielle Konfiguration σ um finale Zustände zu kennzeichnen.

Definition 4.2
Eine *Konfiguration* ist ein Paar $(c, \sigma) \in \mathit{Prog} \times \mathit{State}$, wobei

- c das (noch) auszuführende Programm ist, und
- σ der Zustand des Programms ist. □

Definition 4.3
Die *Small-Step Transitionsrelation* zwischen Konfigurationen

$$\to \subseteq (\mathit{Prog} \times \mathit{State}) \times \Big(\underbrace{(\mathit{Prog} \times \mathit{State})}_{\text{nicht final}} \cup \underbrace{\mathit{State}}_{\text{final}} \Big)$$

ist die kleinste Relation, die folgenden Regeln genügt:

$$\frac{}{(\texttt{skip},\, \sigma) \to \sigma}\ (\textsc{skip})$$

$$\frac{}{(x = a,\, \sigma) \to \sigma[x \mapsto \mathcal{S}[\![\, a \,]\!](\sigma)]}\ (\textsc{assign})$$

$$\frac{(c_1,\, \sigma) \to \sigma'}{(c_1; c_2,\, \sigma) \to (c_2,\, \sigma')}\ (\textsc{seq}1)$$

$$\frac{(c_1,\, \sigma) \to (c_1',\, \sigma')}{(c_1; c_2,\, \sigma) \to (c_1'; c_2,\, \sigma')}\ (\textsc{seq}2)$$

$$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 1}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end},\, \sigma) \to (c_1,\, \sigma)}\ (\textsc{if\_true})$$

$$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 0}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end},\, \sigma) \to (c_2,\, \sigma)}\ (\textsc{if\_false})$$

$$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 1}{(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end},\, \sigma) \to (c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end},\, \sigma)}\ (\textsc{while\_true})$$

$$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 0}{(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end},\ \sigma) \to \sigma}\ (\textsc{while\_false})$$

$$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 1}{(\texttt{assume}(b),\ \sigma) \to \sigma}\ (\textsc{assume})$$

□

Bemerkung

- Die Definition der Transitionsrelation benutzt Regeln der Form

  $$\frac{P}{C}$$

  wobei die Schlussfolgerung C nur gezogen werden kann, falls die Vorbedingung P gilt.
- Regeln ohne Vorbedingung heißen *Axiome*.
- Ein `assume` Statement hat nur eine Regel, nämlich für den Fall, dass die Bedingung erfüllt ist. Falls die Bedingung nicht erfüllt ist, kann das `assume` nicht abgeleitet werden, blockiert somit also jegliche weitere Ausführung.
- Man hätte auch folgende, äquivalente Regeln benutzen können, welche allerdings zu längeren Ableitungen führen:

  $$\frac{}{(x = a,\ \sigma) \to (\texttt{skip},\ \sigma[x \mapsto \mathcal{S}[\![\, a \,]\!](\sigma)])}\ (\textsc{assign}')$$

  $$\frac{}{(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end},\ \sigma) \to (\texttt{if}\ b\ \texttt{then}\ c; \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}\ \texttt{else}\ \texttt{skip}\ \texttt{end},\ \sigma)}\ (\textsc{while}')$$

  $$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 1}{(\texttt{assume}(b),\ \sigma) \to (\texttt{skip},\ \sigma)}\ (\textsc{assume}')$$

Beispiel 4.4

Betrachte folgendes Programm:

$$\begin{aligned} c &\equiv x = x + 1;\ \mathit{loop} \\ &\equiv x = x + 1;\ \texttt{while}\ x > 0 \wedge y < 1\ \texttt{do}\ y = y + 1\ \texttt{end} \end{aligned}$$

Wir schreiben Zustände σ des Programms wie folgt: $\sigma = (i, j)$ bedeutet $\sigma(x) = i$ und $\sigma(y) = j$.

Dann liefern die obigen Regeln folgende Ableitungen:

(1) $(x = x + 1,\ (0,0)) \to (1,0)$ nach (ASSIGN)

(2) $(x = x + 1; \mathit{loop},\ (0,0)) \to (\mathit{loop},\ (1,0))$ nach (SEQ1) zusammen mit (1)

(3) $\mathcal{S}[\![\, x > 0 \,]\!]((1,0)) => (1,0) = 1$

(4) $\mathcal{S}[\![\, y < 1 \,]\!]((1,0)) =< (0,1) = 1$

(5) $\mathcal{S}[\![\, x > 0 \wedge y < 1 \,]\!]((1,0)) = \mathit{min}(1,1) = 1$ nach (3) und (4)

(6) $(\mathit{loop},\ (1,0)) \to (y = y + 1; \mathit{loop},\ (1,0))$ nach (WHILE_TRUE) zusammen mit (5)

(7) $(y = y + 1,\ (1,0)) \to (1,1)$ nach (ASSIGN)

(8) $(y = y + 1; \mathit{loop},\ (1,0)) \to (\mathit{loop},\ (1,1))$ nach (SEQ1) zusammen mit (7)

(9) $\mathcal{S}[\![\, y < 1 \,]\!]((1,1)) =< (1,1) = 0$

(10) $\mathcal{S}[\![x > 0 \wedge y < 1]\!]((1,0)) = min(\cdot, 0) = 0$ nach (9)

(11) $(loop, (1,1)) \to (1,1)$ nach (WHILE_FALSE) zusammen mit (10)

Oft stellt man die Ableitung als Transitionssystem $((Prog \times State) \cup State, \to, init)$ dar, wobei *init* die initial Konfiguration ist. Für gewöhnlich stellt man nur die tatsächlich erreichbaren Konfigurationen dar. Obiges Beispiel liefert:

$$\begin{aligned} init &= (x = x + 1; loop, (0,0)) \\ &\to (loop, (1,0)) && \text{nach (2)} \\ &\to (y = y + 1; loop, (1,0)) && \text{nach (6)} \\ &\to (loop, (1,1)) && \text{nach (8)} \\ &\to (1,1) && \text{nach (11)} \end{aligned}$$ □

Beispiel 4.5

Betrachte folgende Modifikation

$$\begin{aligned} c &\equiv x = x + 1;\ loop \\ &\equiv x = x + 1;\ \texttt{while}\ x > 0\ \texttt{do}\ y = y + 1\ \texttt{end} \end{aligned}$$

des obigen Programmes, sodass es nicht terminiert.

Dann erhalten wir einen unendliche/nicht-terminierende Ableitung wie folgt:

$$\begin{aligned} init &= (x = x + 1; loop, (0,0)) \\ &\to (loop, (1,0)) \\ &\to (y = y + 1; loop, (1,0)) \\ &\to (loop, (1,1)) \\ &\to \ldots \\ &\to (loop, (1,k)) \\ &\to (y = y + 1; loop, (1,k)) \\ &\to (loop, (1,k+1)) \\ &\to \ldots \end{aligned}$$ □

Lemma 4.6

Die Small-Step Transitionsrelation ist *deterministisch*: Für alle Ableitungen $(c, \sigma) \to cf'$ und $(c, \sigma) \to cf''$ gilt $cf' = cf''$. □

4.1.2 Big-Step Semantik

Ziel: Definiere die Semantik des Programmes, sodass die gesamten Ausführung zusammen gefasst wird.

Definition 4.7

Die *Big-Step Transitionsrelation* zwischen Konfigurationen

$$\Downarrow \subseteq (Prog \times State) \times State$$ □

ist die kleinste Relation, die folgenden Regeln genügt:

$$\frac{}{(\texttt{skip}, \sigma) \Downarrow \sigma}\ (\textsc{Bskip})$$

$$\frac{}{(x = a,\ \sigma) \Downarrow \sigma[x \mapsto \mathcal{S}[\![\, a \,]\!](\sigma)]}\ (\textsc{Bassign})$$

$$\frac{(c_1,\ \sigma) \Downarrow \sigma' \qquad (c_2,\ \sigma') \Downarrow \sigma''}{(c_1; c_2,\ \sigma) \Downarrow \sigma''}\ (\textsc{Bseq})$$

$$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 1 \qquad (c_1,\ \sigma) \Downarrow \sigma'}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end},\ \sigma) \Downarrow \sigma'}\ (\textsc{Bif\_true})$$

$$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 0 \qquad (c_2,\ \sigma) \Downarrow \sigma'}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end},\ \sigma) \Downarrow \sigma'}\ (\textsc{Bif\_false})$$

$$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 1 \qquad (c,\ \sigma) \Downarrow \sigma' \qquad (\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end},\ \sigma') \Downarrow \sigma''}{(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end},\ \sigma) \Downarrow \sigma''}\ (\textsc{Bwhile\_true})$$

$$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 0}{(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end},\ \sigma) \Downarrow \sigma}\ (\textsc{Bwhile\_false})$$

$$\frac{\mathcal{S}[\![\, b \,]\!](\sigma) = 1}{(\texttt{assume}(b),\ \sigma) \Downarrow \sigma}\ (\textsc{Bassume})$$

Beispiel 4.8
Betrachte wieder folgendes Programm:

$$\begin{aligned} c &\equiv x = x + 1;\ \mathit{loop} \\ &\equiv x = x + 1;\ \texttt{while}\ x > 0 \wedge y < 1\ \texttt{do}\ y = y + 1\ \texttt{end} \end{aligned}$$

Dann liefern die obigen Regeln folgende Ableitungen:

(1) $(x = x + 1,\ (0,0)) \Downarrow (1,0)$ nach (Bassign)

(2) $(y = y + 1,\ (1,0)) \Downarrow (1,1)$ nach (Bassign)

(3) $(\mathit{loop},\ (1,1)) \Downarrow (1,1)$ nach (Bwhile_false) und $\mathcal{S}[\![\, x > 0 \wedge y < 1 \,]\!]((1,1)) = 0$

(4) $(\mathit{loop},\ (1,0)) \Downarrow (1,1)$ nach (Bwhile_true) zusammen mit $\mathcal{S}[\![\, x > 0 \wedge y < 1 \,]\!]((1,0)) = 1$, (2), und (3)

(5) $(c,\ (0,0)) \Downarrow (1,1)$ nach (Bseq) zusammen mit (1) und (4)

Wir können die Ableitungen als Beweisbaum darstellen:

$$\dfrac{\dfrac{}{(x = x + 1,\ (0,0)) \Downarrow (1,0)} \qquad \dfrac{\mathcal{S}[\![\, x > 0 \wedge y < 1 \,]\!]((1,0)) = 1 \qquad \dfrac{}{(y = y + 1,\ (1,0)) \Downarrow (1,1)} \qquad \dfrac{\mathcal{S}[\![\, x > 0 \wedge y < 1 \,]\!]((1,1)) = 0}{(\mathit{loop},\ (1,1)) \Downarrow (1,1)}}{(\mathit{loop},\ (1,0)) \Downarrow (1,1)}}{(c,\ (0,0)) \Downarrow (1,1)}$$

□

Beobachtung
In der Big-Step Semantik kann Divergenz (Nicht-Terminierung) nicht erkannt werden — es gibt lediglich keinen finalen Zustand, da der Beweisbaum nicht *abgeschlossen* werden kann. Im Gegensatz dazu ist Divergenz in der Small-Step Semantik erkennbar, da die Zwischenkonfigurationen *sichtbar* sind — man erhält einen unendlichen/nicht-kreisfreien Konfigurationsgraphen. □

Satz 4.9 (Zusammenhang Small-Step und Big-Step Semantik)
Für alle c, σ gilt: $(c, \sigma) \to^* \sigma'$ gdw. $(c, \sigma) \Downarrow \sigma'$. □

Beweis (Idee)

"⇒" Führe eine Induktion nach der Länge n der Ableitung durch. Zeige:

$$\forall\, c \in \mathit{Prog}\ \forall\, \sigma, \sigma' \in \mathit{State}\,.\, (c, \sigma) \to^n \sigma' \implies (c, \sigma) \Downarrow \sigma'$$

Induktionsanfang: Betrachte Ableitungen der Länge $n \leq 1$. Nach Definition, trifft dies auf die Regeln (SKIP), (ASSIGN), (WHILE_FALSE), und (ASSUME) zu. Diese haben Entsprechungen in der Big-Step Semantik.

Induktionsschritt: Betrachte eine Ableitung der Länge $n + 1$: $(c_0, \sigma_0) \to (c, \sigma) \to^n \sigma'$. Nach Induktion existiert $(c, \sigma) \Downarrow \sigma'$. Führe nun eine Fallunterscheidung über die Möglichen Transition $(c_0, \sigma_0) \to (c, \sigma)$, um die Aussage zu zeigen.

"⇐" Ähnlich zum ersten Fall. Führe eine Induktion nach der Höhe n des Beweisbaums durch. Zeige: Falls $(c, \sigma) \Downarrow \sigma'$ mittels eines Beweisbaums der Höhe höchstens n nachgewiesen werden kann, so existiert eine Ableitung der Form $(c, \sigma) \to^* \sigma'$. ■

Bemerkung (Vergleich von Small-Step und Big-Step Semanitk)

Small-Step Semantik:

- Komplexe Programmkonstrukte sind einfach zu modellieren. Z.B.: Nebenläufigkeit, Speicher, Laufzeitfehler, Divergenz (Nicht-Terminierung).
- Potenziell aufwändig Eigenschaften von Programmen nachzuweisen.

Big-Step Semantik:

- Typischerweise einfacher Programmeigenschaften nachzuweisen.
- Information über Zwischenzustände wird nicht explizit dargestellt. Dadurch sehen Programme ohne Endzustände alle gleich aus (Schleifen, Fehler, Deadlocks).
- Einige Programmeigenschaften können nicht nachgewiesen werden. □

4.2 Axiomatische Semantik

Ziel: Zeige, dass Programme einen gewünschten Effekt erzielen, bzw. *korrekt* sind. Dazu müssen alle möglichen Eingaben betrachtet werden (nicht wie bisher nur einzelne Startzustände).

Ansatz:

- Zeige Aussagen der Form:

  > Falls $x < 1$ vor der Ausführung von Programm c gilt, dann erhalten wir $x > 2$ nach der Ausführung von c — vorausgesetzt das Programm terminiert.

- Diese Aussage stellt *partielle Korrektheit* dar, wobei partiell hier die Tatsache beschreibt, dass das Programm nicht terminieren muss und in einem solchen Falle keinerlei Garantien gegeben werden.
- Um Terminierung mit einzubeziehen, macht man Aussagen über *totale Korrektheit*. Diese sind von der Form:

  > Falls $x < 1$ vor der Ausführung von Programm c gilt, dann terminiert c garantiert und wir erhalten $x > 2$ nach der Ausführung.

  (In dieser Vorlesung werden wir nicht auf totale Korrektheit eingehen.)
- Aussagen der partiellen Korrektheit werden als *Hoare Triple* (nach Sir Tony Hoare) dargestellt

  $$\{\, x < 1 \,\}\, c \,\{\, c > 2 \,\}$$

  mit
  - *Vorbedingung (Precondition)* $x < 1$, die die Startzustände σ von Interesse charakterisiert.
  - Programm c.
  - *Nachbedingung (Postcondition)* $x > 2$, die angibt, welche Eigenschaften die Endzustände σ', $(\mathsf{c}, \sigma) \Downarrow \sigma'$, erfüllen.

  Vor- und Nachbedingungen werden auch *Zusicherungen (Assertions)* genannt.

Bemerkung

- Dieser Ansatz ist die *Alte Schule* der Programmverifikation und geht zurück bis zu Turings Paper *Checking a large routine* aus dem Jahre 1949. Popularität erlangt der Ansatz erst später durch die Arbeiten von Robert Floyd und Tony Hoare.
- Ursprünglich war der Ansatz nicht zum Nachweis von Eigenschaften von Programmen entwickelt worden, sondern um die Bedeutung von Programmkonstrukten zu erklären. Die Bedeutung ist dabei beschrieben durch Axiome und Regeln, ähnlich zur SOS-Semantik. Daher der Name *axiomatische Semantik*. □

Definition 4.10

Assertions sind Formeln der Prädikatenlogik erster Stufe über der Signatur von While-Programmen:

$$A ::= \mathit{true} \mid \mathit{false} \mid a_1 < a_2 \mid \neg A \mid A_1 \wedge A_2 \mid \exists\, i.\ A$$

mit

- i ist eine ganzzahlige Variable über $\mathbb{Z}$, die von den im Programm vorkommenden Variablen unterschiedliche ist,
- a_1 und a_2 sind arithmetische Ausdrücke, die neben Programmvariablen auch ganzzahlige Variablen enthalten dürfen, und
- Assertions sind abgeschlossen bezüglich ganzzahligen Variablen (d.h. diese treten nur quantifiziert auf).

Wir schreiben $\mathcal{S}, \sigma \vDash A$, falls $\mathcal{S}[\![A]\!](\sigma) = \mathit{true}$. Da es sich bei $\mathcal{S}$ immer um die Struktur von While-Programmen handelt, schreiben wir auch $\sigma \vDash A$ — sprich σ *erfüllt* A. □

Definition 4.11

Ein *Hoare Triple* hat die Form $\{ A \}\, c\, \{ B \}$ wobei A, B Assertions sind und `c` ein Programm ist.

Ein Hoare Triple ist *gültig* (bzgl. $\mathcal{S}$), geschrieben $\vDash \{ A \}\, c\, \{ B \}$, falls gilt

$$\forall \sigma, \sigma' \in \mathit{State}\,.\; \sigma \vDash A \wedge (c, \sigma) \Downarrow \sigma' \implies \sigma' \vDash B\;. \qquad \square$$

Beispiel 4.12

- Betrachte $c =$ `while true do skip end`. Dann gilt $\vDash \{ \mathit{true} \}\, c\, \{ A \}$ für alle Assertions A, da c nicht terminiert.
- Für $c =$ `while` $x \leq 2$ `do skip end` gilt $\vDash \{ \mathit{true} \}\, c\, \{ x > 2 \}$, da c nur für $x > 2$ terminiert.
- $\{ s = 0, n = 1 \}$ `while` $n < 101$ `do` $s = s + n;\ n = n + 1$ `end` $\{ s = \sum_{i=1}^{100} i \}$ □

4.2.1 Der Hoare Kalkül

Ziel: Prüfe die Validität von Hoare Tripeln algorithmisch.

Ansatz: Benutze ein Beweissystem (also mit Regeln syntaktischer Natur), welches genau die validen Hoare Tripel (Eigenschaften semantischer Natur) fasst.

Definition 4.13

Die Regeln des *Hoare Kalküls* sind die folgenden:

$$\frac{}{\{ A \}\, \texttt{skip}\, \{ A \}}\ (\textsc{skip})$$

$$\frac{}{\{ A[x \mapsto a] \}\, x = a\, \{ A \}}\ (\textsc{assign})$$

$$\frac{\{ A \}\, c_1\, \{ C \} \qquad \{ C \}\, c_2\, \{ B \}}{\{ A \}\, c_1;\, c_2\, \{ B \}}\ (\textsc{seq})$$

$$\frac{\{ A \wedge b \}\, c_1\, \{ B \} \qquad \{ A \wedge \neg b \}\, c_2\, \{ B \}}{\{ A \}\, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}\, \{ B \}}\ (\textsc{if})$$

$$\frac{\{ A \wedge b \}\, c\, \{ A \}}{\{ A \}\, \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}\, \{ A \wedge \neg b \}}\ (\textsc{while})$$

$$\frac{A \implies A' \qquad \{ A' \}\, c\, \{ B' \} \qquad B' \implies B}{\{ A \}\, c\, \{ B \}}\ (\textsc{consequence})$$

Falls es einen Beweis in obigem Kalkül für $\{ A \}\, c\, \{ B \}$ gibt, so schreiben wir $\vdash \{ A \}\, c\, \{ B \}$. □

Bemerkung

- Die Regel (ASSIGN) ist die Gewünschte. Betrachte $x = x + 2$. Welche Vorbedingung benötigen wir, um $x > 2$ nach der Ausführung zu garantieren? Der Regel zufolge:

$$\{\, (x > 2)[x \mapsto x + 2] \,\}\; x = x + 2 \;\{\, x > 2 \,\}$$

  Tatsächlich:

$$(x > 2)[x \mapsto x + 2] \equiv x + 2 > 2 \iff x > 0$$

- In Regel (WHILE) wird A auch *Schleifeninvariante* genannt, da die Voraussetzung $\{\, A \wedge b \,\}\; c \;\{\, A \,\}$ der Regel verlangt, dass das Ausführen des Schleifenrumpfes die Assertion A bewahrt.
- Das Finden von Schleifeninvarianten ist schwer!
- Intuitiv gilt Regel (CONSEQUENCE), da
  - die Vorbedingung verstärkt wird (sodass sie von *weniger* Zuständen erfüllt wird), und
  - die Nachbedingung abschwächt wird (sodass sie von *mehr* Zuständen erfüllt wird). □

Beispiel 4.14

Betrachte das folgende Programm:

$$\begin{aligned} c \equiv\ & \texttt{while}\ x > 0\ \texttt{do} \\ & \quad y = y * x; \\ & \quad x = x - 1; \\ & \texttt{end} \end{aligned}$$

Wir wollen nachweisen, dass c die Fakultät $n!$ berechnet, mit $n! = n * (n-1) * (n-2) * \cdots * 2 * 1$ und $0! = 1$. Genauer bedeutet dies, dass wir die Validität des folgenden Hoare Triples nachweisen wollen:

$$\vdash \{\, x = n \wedge n \geq 0 \wedge y = 1 \,\}\; c \;\{\, y = n! \,\}$$

Dazu gehen wir wie folgt vor:

- Da c eine Schleife enthält, benötigen wir eine adäquate Schleifeninvariante I:

$$I \equiv y * x! = n! \wedge x \geq 0$$

  Um zu zeigen, dass es sich bei I tatsächlich um eine Schleifeninvariante hält, müssen wir

$$\vdash \{\, I \wedge x > 0 \,\}\; y = y * x; x = x - 1; \;\{\, I \,\}$$

  nachweisen. Dazu:
  - Regel (ASSIGN) liefert
$$\vdash \{\, I[x \mapsto x - 1] \,\}\; x = x - 1 \;\{\, I \,\}$$
  - Eine weitere Anwendung von (ASSIGN) gibt
$$\vdash \{\, I[x \mapsto x - 1][y \mapsto y * x] \,\}\; y = y * x \;\{\, I[x \mapsto x - 1] \,\}$$
  - Damit folgt
$$\vdash \{\, I[x \mapsto x - 1][y \mapsto y * x] \,\}\; y = y * x; x = x - 1 \;\{\, I \,\}$$
    aus (SEQ)

– Es gilt:

$$\begin{aligned} I[x \mapsto x-1][y \mapsto y*x] &\equiv \big(y*(x-1)! = n! \wedge x-1 \geq 0\big)[y \mapsto y*x] \\ &\equiv y*x*(x-1)! = n! \wedge x-1 \geq 0 \end{aligned}$$

– Ferner gilt:

$$\begin{aligned} I \wedge x > 0 &\implies y*x! = n! \wedge x \geq 0 \wedge x > 0 \\ &\implies y*x! = n! \wedge x \geq 1 \\ &\implies y*x*(x-1)! = n! \wedge x-1 \geq 0 \\ &\iff I[x \mapsto x-1][y \mapsto y*x] \end{aligned}$$

– Also folgt mit (CONSEQUENCE) das Gewünschte:

$$\vdash \{\, I \wedge x > 0 \,\}\; y = y*x; x = x-1;\; \{\, I \,\}$$

- Eine Anwendung von (WHILE) liefert

$$\vdash \{\, I \,\}\; c\; \{\, I \wedge \neg(x > 0) \,\}$$

- Es gilt:

$$x = n \wedge n \geq 0 \wedge y = 1 \implies y*x! = n! \wedge x \geq 0 \iff I$$

- Des Weiteren gilt:

$$\begin{aligned} I \wedge \neg(x > 0) &\implies y*x! = n! \wedge x \geq 0 \wedge \neg(x > 0) \\ &\implies y*x! = n! \wedge x = 0 \\ &\implies y*0! = n! \\ &\implies y = n! \end{aligned}$$

- Damit lässt sich Vorbedingung des obigen Hoare Triples verstärken und die Nachbedingung abschwächen, sodass wir mit (CONSEQUENCE) erhalten:

$$\vdash \{\, x = n \wedge n \geq 0 \wedge y = 1 \,\}\; c\; \{\, y = n! \,\}$$

□

4.2.2 Vollständigkeit und Korrektheit des Hoare Kalküls

Ziel: Zeige, dass der Hoare Kalkül (1) vollständig und (2) korrekt ist. Das bedeutet, dass (1) die Validität eines jeden Hoare Tripels im Kalkül nachweisbar ist, und dass (2) jedes im Kalkül ableitbarer Hoare Tripel tatsächlich valide ist.

Problem: Um die Vollständigkeit des Hoare Kalküls zu zeigen, müssen wir für jedes gültige Hoare Triple $\{\, A \,\}\; c\; \{\, B \,\}$ zeigen, dass sich im Kalkül herleiten lässt. Typischerweise bietet es sich an, einen solchen Beweis per Induktion nach der Struktur der Programme zu führen. Wenn man in einer solchen strukturellen Induktion den Fall

$$\vDash \{\, A \,\}\; c_1; c_2\; \{\, B \,\}$$

betrachtet, benötigt man, um die Induktionsvoraussetzung anwenden zu können, eine Assertion C mit

$$\vDash \{\, A \,\}\; c_1\; \{\, C \,\} \qquad \text{und} \qquad \vDash \{\, C \,\}\; c_2\; \{\, B \,\}\;.$$

Existiert ein solches C, so würde man über die Induktionsvoraussetzung

$$\vdash \{\, A \,\}\; c_1\; \{\, C \,\} \qquad \text{und} \qquad \vdash \{\, C \,\}\; c_2\; \{\, B \,\}$$

erhalten und die Behauptung mittels (CONSEQUENCE) folgern.

Wie können wir die Existenz eines solchen C nachweisen?

Lösung: Zeigen, dass für jedes Programm c und jede Nachbedingung B diejenige Menge an Zuständen

- von denen aus die Ausführung divergiert, oder
- von denen aus die Ausführung in einem B-erfüllenden Zustand terminiert

mittels der zuvor definierten Assertions ausdrückbar ist.

Definition 4.15

Sei c ein Programm und B eine Nachbedingung. Die *weakest liberal precondition (wlp)* ist die Menge an Zuständen definiert durch:

$$wlp(c,\, B) := \{\, \sigma \in \mathit{State} \mid \forall\, \sigma' \in \mathit{State}\,.\, (c,\, \sigma) \Downarrow \sigma' \implies \sigma' \vDash B \,\}\ .$$

Der Ausdruck *liberal* gibt hier an, dass es sich um partielle Korrektheit handelt. Für totale Korrektheit nennt man die entsprechende Definition *weakes precondition (wp)*. □

Bemerkung

Es ist nicht klar/offensichtlich, dass die von uns definierten Assertions für jedes beliebige c und B ein Element enthält, welches $wlp(c,\, B)$ charakterisiert. □

Definition 4.16

Eine Assertion-Sprache $\mathcal{L}$ ist *ausdrucksmächtig*, falls es für alle $c \in \mathit{Prog}$ und $B \in \mathcal{L}$ eine Assertion $A \in \mathcal{L}$ gibt mit

$$wlp(c,\, B) = \mathcal{S}\,[\![\, A \,]\!] := \{\, \sigma \in \mathit{State} \mid \sigma \vDash A \,\}$$ □

Lemma 4.17

Sei $\mathcal{S}\,[\![\, A \,]\!] = wlp(c,\, B)$. Dann gilt:

(1) Das Hoare Triple $\{\, A \,\}\, c\, \{\, B \,\}$ ist gültig, also $\vDash \{\, A \,\}\, c\, \{\, B \,\}$.

(2) Falls $\vDash \{\, A' \,\}\, c\, \{\, B \,\}$ gilt, so gilt auch $A' \implies A$.

Intuitiv bedeutet Aussage (1), dass es sich bei A um eine Vorbedingung handelt, und Aussage (2), dass es sich bei A um die schwächste Vorbedingung handelt (bezüglich logischer Implikation $\implies$; faktorisiert nach logischer Äquivalenz $\iff$). □

Satz 4.18 (Dijkstra'76)

Die obige, von uns definierte Assertion-Sprache ist ausdrucksmächtig. □

Bemerkung

Der Satz zeigt, dass die schwächste Vorbedingung in der Assertion-Sprache fassbar ist.

Der Satz zeigt noch mehr, nämlich *wie* man $wlp(\cdot,\, \cdot)$ berechnet. Wir beschränken uns auf Schleifen-freie Programme, und definieren die Funktion $pred(c,\, B)$ wie folgt:

$$\begin{aligned}
pred(\texttt{skip},\, B) &:= B \\
pred(x = a,\, B) &:= B[x \mapsto a] \\
pred(c_1; c_2,\, B) &:= pred(c_1,\, pred(c_2,\, B)) \\
pred(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end},\, B) &:= (b \wedge pred(c_1,\, B)) \vee (\neg b \wedge pred(c_2,\, B)) \\
pred(\texttt{assume}(b),\, B) &:= b \implies B
\end{aligned}$$

Man kann zeigen

$$\forall\, \sigma \in \mathit{State}\,.\, \sigma \vDash pred(c,\, B) \iff \sigma \in wlp(c,\, B)$$

und somit $\mathcal{S}\,[\![\, pred(c,\, B) \,]\!] = wlp(c,\, B)$. Aus diesem Grunde schreiben wir zumeist einfach $wlp(c,\, B)$ und meinen damit $pred(c,\, B)$. □

Mit der gezeigten Ausdrucksstärke des Hoare Kalküls kann man nun die Vollständigkeit zeigen. Das folgende Lemma spielt dabei eine zentrale Rolle.

Lemma 4.19

Für alle Programme c und Assertions b gilt: $\vdash \{\, \mathit{pred}(c,\, B) \,\}\, c\, \{\, B \,\}$. □

Beweis (Idee)

Induktion nach der Struktur von c.

Induktionsanfang: Für `skip` und Zuweisungen folgt die Behauptung direkt aus den Beweisregeln des Hoare Kalküls.

Induktionsschritt: Fallunterscheidung über die zusammengesetzten Anweisungen, dann Induktionsvoraussetzung plus Regeln des Hoare Kalküls. ■

Satz 4.20 (Vollständigkeit, Cook'74)

Für alle A, B, c gilt: aus $\vDash \{\, A \,\}\, c\, \{\, B \,\}$ folgt $\vdash \{\, A \,\}\, c\, \{\, B \,\}$. □

Beweis

Nach Lemma 4.19 gilt $\vdash \{\, \mathit{pred}(c,\, B) \,\}\, c\, \{\, B \,\}$. Satz 4.18 liefert $\mathcal{S}\, [\![\, \mathit{pred}(c,\, B) \,]\!] = \mathit{wlp}(c,\, B)$. Damit ergibt sich $A \implies \mathit{pred}(c,\, B)$ aus Lemma 4.17. Mit (CONSEQUENCE) folgt dann $\vdash \{\, A \,\}\, c\, \{\, B \,\}$. ■

Satz 4.21 (Korrektheit)

Für alle A, B, c gilt: aus $\vdash \{\, A \,\}\, c\, \{\, B \,\}$ folgt $\vDash \{\, A \,\}\, c\, \{\, B \,\}$. □

Beweis (Idee)

Standardinduktion nach der Höhe des Beweisbaums. ■

4.3 Verification Condtions

Ziel: Zeige Korrektheit von Programmen, also Validität von Hoare Tripeln, (vollständig) automatisch.

Idee:

- Wir haben bisher gezeigt, dass $\vDash \{ A \}\, c \,\{ B \}$ äquivalent ist zu $\vDash A \implies \mathit{pred}(c, B)$.
- Beachte, dass es sich bei $A \implies \mathit{pred}(c, B)$ um eine logische Formel handelt, die komplett frei vom Programm c ist.
- Wir hoffen, dass *Solver* (Theorem Prover, SAT-Solver, SMT-Solver, Constraint-Solver, ...) die Validität solcher Formeln automatisch zeigen können.

Ansatz: Annotiere das Programm mit Schleifen-Invarianten und ggf. mit weiteren Assertions, um die Aufgabe des Solvers zu vereinfachen. (Wir haben bereits gesehen, dass das Finden von Schleifen-Invarianten nicht trivial ist.)

Definition 4.22
Die Syntax von *annotierten While-Programmen* ist durch folgende BNF gegeben:

$$\begin{aligned} c ::= \quad & \texttt{skip} \mid x := a \mid c_1; c_2 \mid c_1; \{ D \}\, c_2 \\ \mid\ & \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end} \\ \mid\ & \texttt{while}\ b\ \texttt{do}\ \{ D \}\, c\ \texttt{end} \end{aligned}$$

wobei D eine Assertion ist, und die arithmetischen und boolschen Ausdrücke wie bisher definiert sind.

Wie bisher schreiben wir $\vDash \{ A \}\, c \,\{ B \}$ für annotierte While-Programme c. Dieses Hoare Triple ist gültig, falls das entsprechende Hoare Triple ohne Annotationen gültig ist. □

Bemerkung
Beachte, dass Schleifen-Invarianten annotiert werden müssen. Annotationen in Sequenzen sind dagegen optional.

Der Einfachhalt halber führen wir keine Annotationen in If-Then-Else-Blöcken ein. Um eine Annotation zu erhalten, kann eine Sequenz mit `skip` verwendet werden. □

Beispiel 4.23

$$\begin{aligned} c \equiv\ & \texttt{while}\ x > 0\ \texttt{do} \\ & \quad \{ y * x! = n! \wedge x \geq 0 \} \\ & \quad y = y * x; \\ & \quad x = x - 1; \\ & \texttt{end} \end{aligned}$$

□

Ansatz (im Detail): Wir wollen jedem Hoare Triple $\{ A \}\, c \,\{ B \}$, wobei c ein annotiertes Programm ist, eine Menge an Zusicherungen zuordnen, deren Validität die Validität des zugehörigen Hoare Triples impliziert. Wir nennen diese Menge an Zusicherungen *Verification Conditions*, geschrieben $vc(\{ A \}\, c \,\{ B \})$.

- Betrachte das Hoare Triple $\{ A \}\, c_1; c_2 \,\{ B \}$, wobei c_1 und c_2 Schleifen-frei sind. Dann können wir eine Annotation generieren:

$$\{ A \}\, c_1; \{ D \} c_2 \,\{ B \} \quad \text{mit} \quad D = \mathit{pred}(c_2, B)\ .$$

- Somit annotieren wir das gesamte Programm, und müssen nur noch die Validität der einzelnen $\{ A \}\, c\, \{ B \}$ prüfen, wobei c primitive Anweisungen sind. (Wie diese Validität zu prüfen ist, wird sofort aus dem Hoare Kalkül klar.)
- Problematisch sind allerdings Schleifen.
  (1) Wie generieren wir Verification Contions für Schleifen?
  (2) Wie generieren wir Annotationen für nicht-annotierte Sequenzen, die Schleifen enthalten?

ad 1) Betrachte $\{ A \}\, \texttt{while}\ b\ \texttt{do}\ \{ D \}\, c\, \texttt{end}\ \{ B \}$. Dabei soll D eine Schleifeninvariante sein. Das heißt, es soll gelten:

$$\vDash \{ D \wedge b \}\, c\, \{ D \}\ .$$

Wenn nun zusätzlich gilt:

$$A \implies D \quad \text{und} \quad D \wedge \neg b \implies B$$

dann erhalten wir mit (CONSEQUENCE)

$$\vDash \{ A \}\, \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}\ \{ B \}\ .$$

Dementsprechend sollte die Verification Condition für While-Schleifen wie folgt lauten:

$$\begin{aligned} & vc(\{ A \}\, \texttt{while}\ b\ \texttt{do}\ \{ D \}\, c\, \texttt{end}\ \{ B \}) \\ := {} & vc(\{ D \wedge b \}\, c\, \{ D \}) \cup \{ A \implies D \} \cup \{ D \wedge \neg b \implies B \} \end{aligned}$$

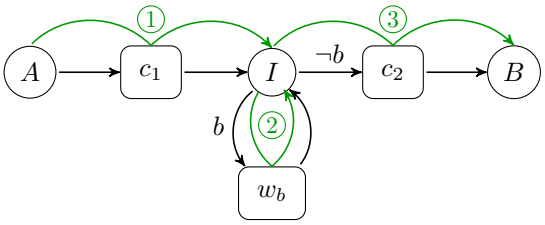
ad 2) Betrachte nun $\{ A \}\, c; w\, \{ B \}$, wobei w eine While-Schleife mit Invariante I ist. Wie zuvor suchen wir eine Annotation D, so dass $\vDash \{ A \}\, c; \{ D \}\, w\, \{ B \}$ gilt. Nach obiger Diskussion wissen wir, dass falls $\vDash vc(\{ D \}\, w\, \{ B \})$ gilt, dann

- ist I eine Schleifen-Invariante, und
- $D \implies I$

Wählen also schwächste Annotation, nämlich $D := I$.

Anschaulich:

- Teile das Programm in sogenannte *Basic Paths* auf.
- Ein Basic Path enthält keine Schleifen (keine Kreise im Kontrollflussgraphen; If-Blöcke sind erlaubt).
- Genauer: ein Basic Path geht von Start/Invariant nach Ende/Invariante
- Illustration:



- Im Bild sind die Basic Paths:
  (1) $A - c_1 - B$
  (2) $I - wb - I$
  (3) $I - c_2 - B$
- Wenn wir diese als Hoare Triple notieren, nehmen wir die Schleifenbedingung mit auf (vgl. Assume-Statement-Konstruktion statt Bedingungen in While-Schleifen und If-Then-Else-Blöcken).
  (1) $\{ A \}\, c_1\, \{ I \}$

(2) $\{\, I \wedge b \,\}\, wb \,\{\, I \,\}$

(3) $\{\, I \wedge \neg b \,\}\, c_2 \,\{\, I \,\}$

- Diese Methode erlaubt es uns, die unendlich vielen Pfade durch das Programm *aufzuschneiden*, in endlich viele Basic Paths.

Definition 4.24

Die Funktion vc generiert zu jedem annotiertem Hoare Triple $\{\, A \,\}\, c \,\{\, B \,\}$ eine Menge an *Verification Conditions* wie folgt.

$$
\begin{aligned}
vc(\{\, A \,\}\, \texttt{skip} \,\{\, B \,\}) &:= \{\, A \implies B \,\} \\
vc(\{\, A \,\}\, x = a \,\{\, B \,\}) &:= \{\, A \implies B[x \mapsto a] \,\} \\
vc(\{\, A \,\}\, c_1; \{\, D \,\}\, c_2 \,\{\, B \,\}) &:= vc(\{\, A \,\}\, c_1 \,\{\, D \,\}) \cup vc(\{\, D \,\}\, c_2 \,\{\, B \,\}) \\
vc(\{\, A \,\}\, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end} \,\{\, B \,\}) &:= vc(\{\, A \wedge b \,\}\, c_1 \,\{\, B \,\}) \cup vc(\{\, A \wedge \neg b \,\}\, c_2 \,\{\, B \,\}) \\
vc(\{\, A \,\}\, \texttt{while}\ b\ \texttt{do}\ \{\, D \,\}\, c\ \texttt{end} \,\{\, B \,\}) &:= vc(\{\, D \wedge b \,\}\, c \,\{\, D \,\}) \cup \{\, A \implies D \,\} \cup \{\, D \wedge \neg b \implies B \,\} \\
vc(\{\, A \,\}\, c_1; c_2 \,\{\, B \,\}) &:= vc(\{\, A \,\}\, c_1; \{\, pred'(c_2,\, B) \,\}\, c_2 \,\{\, B \,\})
\end{aligned}
$$

mit

$$
pred'(c, B, :) = \begin{cases} D & , \text{ falls } c \equiv \texttt{while}\ \_\ \texttt{do}\ \{\, D \,\}\ \_\ \texttt{end} \\ pred(c,\, B) & , \text{ sonst} \end{cases}
$$

□

Satz 4.25 (Korrektheit)

$\vDash vc(\{\, A \,\}\, c \,\{\, B \,\})$ impliziert $\vDash \{\, A \,\}\, c \,\{\, B \,\}$ □

Beweis (Idea)

Induktion nach der Struktur von Programmen. ■

Beispiel 4.26

Betrachte folgendes Programm

$$
\begin{aligned}
c \equiv\ & \texttt{while}\ x > 0\ \texttt{do} \\
& \quad \{\, y * x! = n! \wedge x \geq 0 \,\} \\
& \quad y = y * x; \\
& \quad x = x - 1; \\
& \texttt{end}
\end{aligned}
$$

und

$$
\begin{aligned}
A &\equiv x = n \wedge n \geq 0 \wedge y = 1 \\
I &\equiv y * x! = n! \wedge x \geq 0 \\
B &\equiv y = n!
\end{aligned}
$$

Dann liefert vc folgende Verification Conditions:

$$
\begin{aligned}
& vc(\{\, A \,\}\, c \,\{\, B \,\}) \\
=\ & vc(\{\, I \wedge x > 0 \,\}\, y = y * x; x = x - 1;\, \{\, I \,\}) \cup \{\, A \implies I \,\} \cup \{\, I \wedge x \leq 0 \implies B \,\} \\
=\ & vc(\{\, I \wedge x > 0 \,\}\, y = y * x; \{\, I[x \mapsto x - 1] \,\}\, x = x - 1;\, \{\, I \,\}) \\
& \quad \cup \{\, A \implies I \,\} \cup \{\, I \wedge x \leq 0 \implies B \,\} \\
=\ & vc(\{\, I \wedge x > 0 \,\}\, y = y * x \,\{\, I[x \mapsto x - 1] \,\}) \cup vc(\{\, I[x \mapsto x - 1] \,\}\, x = x - 1;\, \{\, I \,\}) \\
& \quad \cup \{\, A \implies I \,\} \cup \{\, I \wedge x \leq 0 \implies B \,\} \\
=\ & \{\, I \wedge x > 0 \implies I[x \mapsto x - 1][y \mapsto y * x] \,\} \cup \{\, I[x \mapsto x - 1] \implies I[x \mapsto x - 1] \,\} \\
& \quad \cup \{\, A \implies I \,\} \cup \{\, I \wedge x \leq 0 \implies B \,\}
\end{aligned}
$$

Es gilt also $\vDash vc(\{\, A \,\}\, c \,\{\, B \,\})$ (vgl. Beispiel 4.14). Mit obigem Satz dann auch $\vDash \{\, A \,\}\, c \,\{\, B \,\}$. □

Bemerkung (Vorteile von Verification Conditions)

Die Validität von Verification Conditions lässt sich mit SAT/SMT Solvern (z.B. `rise4fun.com/z3`) zeigen. Das Generieren und Prüfen der Verification Conditions ist somit automatisierbar. □

Lemma 4.27

Aus $\vDash \{ A \}\, c \,\{ B \}$ folgt *nicht* unbedingt $\vDash vc(\{ A \}\, c \,\{ B \})$. □

Beweis

Betrachte $\vDash \{ \mathit{true} \}\, \texttt{while}\ \mathit{false}\ \texttt{do}\ \{ \mathit{false} \}\, \texttt{skip}\ \texttt{end}\ \{ \mathit{true} \}$. Es gilt:

$$\begin{aligned} & vc(\{ \mathit{true} \}\, \texttt{while}\ \mathit{false}\ \texttt{do}\ \{ \mathit{false} \}\, \texttt{skip}\ \texttt{end}\ \{ \mathit{true} \}) \\ = {} & vc(\{ \mathit{false} \wedge \mathit{false} \}\, \texttt{skip}\ \{ \mathit{false} \}) \cup \{ \mathit{true} \implies \mathit{false} \} \cup \{ \mathit{false} \wedge \neg \mathit{false} \implies \mathit{true} \} \end{aligned}$$

Da $\mathit{true} \implies \mathit{false}$ keine valide Formel ist, ist die generierte Verification Condition nicht valide, also $\nvDash vc(\{ \mathit{true} \}\, \texttt{while}\ \mathit{false}\ \texttt{do}\ \{ \mathit{false} \}\, \texttt{skip}\ \texttt{end}\ \{ \mathit{true} \})$, obwohl das dazugehörige Hoare Triple valide ist.■

5 Abstrakte Interpretation

Ziel: Entwickle eine Theorie der *korrekten* Approximation der Semantik von Programmen. Die bisherigen Datenflussanalysen haben die Semantik von Programmen *nicht* berücksichtigt (bestenfalls intuitiv).

Idee (Cousot&Cousot):

- Führe Programm auf abstrakten Werten aus
- Beispiele: $\mathcal{P}(\{-, 0, +\})$ anstatt $\mathbb{Z}$
- Berücksichtige alle konkreten Eingaben
- *Ersetze* konkrete Operationen durch *abstrakte Operationen*. Dabei müssen alle konkreten Werte berücksichtigt werden, die durch den abstrakten Wert dargestellt werden.

Beispiel 5.1
Für eine konkrete Operation $op(x) = x - 2$ erhalten wir, zum Beispiel, folgende abstrakte Operation:

$$op^{\#}(\{+\}) = \{-, 0, +\}$$
$$op^{\#}(\{0\}) = \{-\} = op^{\#}(\{-\})$$

Wir schreiben typischerweise $op^{\#}$ für diejenige abstrakte Operation, die der konkreten Operation op entspricht. □

Bemerkung
Vorteile:

- *Mächtigkeit:*
  - Abstrakte Interpretation ist *unabhängig vom Kontrollflussgraphen*
  - Funktioniert für viele Klassen von Programmen (if/while, parallel, Objekt-orientiert, Aktoren, funktional, ...)
- *Korrektheit:* durch Theorie garantiert.
- Balance zwischen Präzision und Komplexität: wählbar durch Granularität der abstrakten Domäne.

Nachteile:

- *Komplexität:* bei abstrakter Interpretation oft höher als bei Datenflussanalysen. □

Um ein solches Framework zu verstehen, müssen wir zunächst die abstrakten Domänen studieren, die sich für eine korrekte Abstraktion eignen.

5.1 Galois-Verbindungen

Ziel:

- Beschreibe geeignete *Abstraktionsfunktionen* $\alpha : L \to M$, die jedem *konkreten* Wert in L einen *abstrakten* Wert in M zuordnet.
- Definiere neben α auch *Konkretisierungsfunktion* $\gamma : M \to L$, die jedem abstrakten Wert alle konkreten Werte zuordnet, für die er steht.

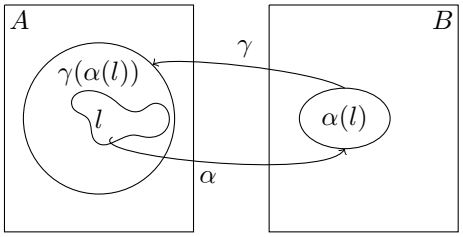
Definition 5.2 (Galois-Verbindung)
Seien $(L, \leq_L)$ und $(M, \leq_M)$ vollständige Verbände.

Ein Paar (α, γ) von monotonen Funktionen $\alpha : L \to M, \gamma : M \to L$ heißt *Galois-Verbindung*, falls

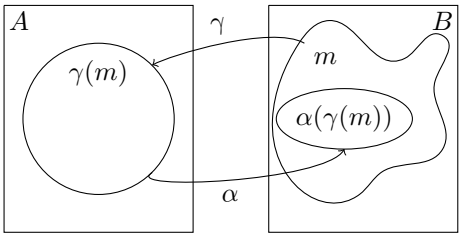
$$\begin{array}{lrlll} \text{(G1)} & \forall l \in L. & l & \leq_L & \gamma(\alpha(l)) \\ \text{(G2)} & \forall m \in M. & \alpha(\gamma(m)) & \leq_M & m \end{array}$$

Wir schreiben $(L, \leq_L) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \leq_M)$. □

Bemerkung (Galois-Verbindungen anschaulich)
Seien $L = \mathcal{P}(A)$ und $M = \mathcal{P}(B)$, wobei A die Menge der konkreten und und B die Menge der abstrakten Werte ist.



($G1$) $l \subseteq \gamma(\alpha(l))$:
α erzeugt Überapproximationen.



($G2$) $\alpha(\gamma(m)) \subseteq m$
Kein Präzisionsverlust durch Abstraktion nach Konkretisierung.

□

Bemerkung
Typisch ist:

- $l \neq \gamma(\alpha(l))$, d.h. Präzisionsverlust durchs Abstrahieren von l.
- $m = \alpha(\gamma(m))$, d.h. kein Präzisionsverlust durch erneutes abstrahieren nach konkretisieren von m (in diesem Fall nennt man die Galois-Verbindung auch *Galois-Insertion*). □

Satz 5.3
Sei $(L, \leq_L) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \leq_M)$ eine Galois-Verbindung. Seien ferner $l \in L, m \in M, L' \subseteq L, M' \subseteq M$. Dann gilt:

(1) $\alpha(l) \leq_M m$ gdw. $l \leq_L \gamma(m)$

(Aus dieser Äquivalenz folgt auch wieder, dass es sich um eine Galois-Verbindung handelt.)

(2a) Die Konkretisierung γ ist *eindeutig* durch α bestimmt durch:

$$\gamma(m) = \sqcup\{l \in L \mid \alpha(l) \leq_M m\} \ .$$

(2b) α ist eindeutig durch γ bestimmt durch:

$$\alpha(l) = \sqcap\{m \in M \mid l \leq_L \gamma(m)\} \ .$$

(3a) α ist *vollständig distributiv*:

$$\alpha(\sqcup L') = \sqcup\alpha(L') \ .$$

(Diese Eigenschaft wird auch *vollständig additiv* genannt.)

(3b) γ ist *vollständig multiplikativ*:

$$\gamma(\sqcap M') = \sqcap\gamma(M') \ .$$

(4a) Zu jeder vollständig distributiven Funktion $\alpha' : L \to M$ gibt es ein $\gamma' : M \to L$, so dass $(L, \leq_L) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (M, \leq_M)$ eine Galois-Verbindung ist.

(4b) Zu jeder vollständig multiplikativen Funktion $\gamma' : M \to L$ gibt es ein $\alpha' : L \to M$, so dass $(L, \leq_L) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (M, \leq_M)$ eine Galois-Verbindung ist. □

Beweis

(1) Es gelte $\alpha(l) \leq_M m$. Dann folgt wie gewünscht:

$$l \overset{(G1)}{\leq_L} \gamma(\alpha(l)) \overset{\text{Mon. } \gamma}{\leq_L} \gamma(m)$$

Also gilt $\alpha(l) \leq_M m \implies l \leq_L \gamma(m)$.

Die verbleibende Richtung, $l \leq_L \gamma(m) \implies \alpha(l) \leq_M m$ folgt analog.

(2a) Zeigen: $\gamma(m) = \sqcup\{l \in L \mid \alpha(l) \leq m\}$ für jede Galois-Verbindung.

$\leq_L$: Falls $\alpha(l) \leq m$, dann $l \leq_L \gamma(m)$ nach (1.) Damit folgt:

$$\sqcup\{\, l \in L \mid \alpha(l) \leq m \,\} \leq \gamma(m)$$

$\geq_L$: Da $\alpha(\gamma(m)) \leq_M m$, gilt

$$\gamma(m) \in \{l \in L \mid \alpha(l) \leq m\}$$

Also

$$\gamma(m) \leq_L \sqcup\{l \in L \mid \alpha(l) \leq_M m\}$$

(2b) analog

(3a) $\geq_M$ Für $l \in L'$, gilt $l \leq_L \sqcup L'$. Mit der Monotonie von α folgt

$$\sqcup\alpha(L') = \sqcup\{\alpha(l) \mid l \in L'\} \leq \alpha(\sqcup L')$$

$\leq_M$ Um $\alpha(\sqcup L') \leq_M \sqcup\alpha(L')$ zu zeigen nutze (1.) und zeige

$$\sqcup L' \leq_L \gamma(\sqcup\alpha(L'))$$

Damit gilt dann

$$\alpha(\sqcup L') \leq_M \sqcup\alpha(L')$$

(3b) analog ■

5.2 Konstruktion von Galois-Verbindungen

Ziel:

1. Definiere zwei elementare Galois-Verbindungen: Intervallabstraktion und Kongruenzabstraktion.
2. Definiere Galois-Verbindungen mittels Extraktionsfunktionen.
3. Komponiere bestehende Galois-Verbindungen.

5.2.1 Elementare Galois-Verbindungen

Intervallabstraktion

Intervallabstraktionen erlauben Aussagen über die absolute Größe von Datenwerten. Allerdings lässt sich mit abstrakten Intervallwerten nur unpräzise rechnen.

Beispiel 5.4
Sei $(L, \subseteq)$ mit $L = \mathcal{P}(\mathbb{Z})$ die konkrete Domäne der Teilmengen von $\mathbb{Z}$.

Sei $(M, \subseteq)$ mit

$$M = \big((\mathbb{Z} \cup \{ -\infty \}) \times (\mathbb{Z} \cup \{ +\infty \})\big) \cup \{ \varnothing \}$$

die abstrakte Domäne der Intervalle.

Wir definieren eine Galois-Verbinung $(L, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \subseteq)$ durch

$$\alpha : L \to M$$

$$\alpha(Z) := \begin{cases} \varnothing, & \text{falls } Z = \varnothing \\ [\sqcap Z, \sqcup Z], & \text{sonst} \end{cases}$$

und

$$\gamma : M \to L$$

$$\gamma(I) := \begin{cases} \varnothing, & \text{falls } I = \varnothing \\ \{ z \in \mathbb{Z} \mid z_1 \leq z \leq z_2 \}, & \text{falls } I = [z_1, z_2] \end{cases}$$

Zum Beispiel gilt

- $\gamma(\alpha(\{1, 3, 5 \ldots\} = \gamma([1, \infty]) = \{1, 2, 3, \ldots\} \subseteq \{1, 3, 5, \ldots\}$
- $\alpha(\gamma([-1, 1])) = \alpha(\{-1, 0, 1\}) = [-1, 1]$ □

Bemerkung
Intervalle benötigen immer noch unbeschränkt viel Information, um die untere und obere Grenze zu speichern. (Beides sind beliebig große ganze Zahlen.) Daher wird in der Praxis oft

$$M = \big((\{ k_1, ..., k_n \} \cup \{ -\infty \}) \times (\{ k_1, ..., k_n \} \cup +\infty)\big) \cup \{ \varnothing \}$$

verwendet. D.h. es werden nur endlich viele Werte werden exakt gespeichert; alle kleineren Werte werden durch $-\infty$ und alle größeren durch $+\infty$ repräsentiert. □

Mehrdimensional: Eine Menge $\Sigma \subseteq \mathbb{R}^n$ lässt sich zu ihrer konvexen Hülle $conv(\Sigma) \subseteq \mathbb{R}^n$ abstrahieren, also dem kleinsten Polyeder, der Σ enthält. Dieser Polyeder lässt sich endlich darstellen (z.B. durch ein Ungleichungssystem, oder durch Eckpunkte und Strahlen).

Kongruenzabstraktion

Die Intervallabstraktion gibt die absoluten Größe der Datenwerte an. Alternativ kann man die absolute Größe der Werte *vergessen*, und erhält eine Galois-Verbinung, bei der man mit den abstrakten Werten sehr schnell Rechnen kann.

Beispiel 5.5

Seien $(L, \subseteq)$ und $(M, \subseteq)$ Verbände mit $L = \mathcal{P}(\mathbb{Z})$ und $M = \mathcal{P}(\{0, \ldots, k-1\})$, wobei $k \in \mathbb{N} \backslash \{0\}$.

Wir definieren eine Galois-Verbinung $(L, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \subseteq)$ durch

$$\begin{aligned} \alpha &: L \to M \\ \alpha(Z) &:= \{ z \mod k \mid z \in Z \} \end{aligned}$$

und

$$\begin{aligned} \gamma &: M \to L \\ \gamma(I) &:= \{ z \in \mathbb{Z} \mid z \equiv m \mod k, \text{ für ein } m \in M \} \end{aligned}$$

(Beachte: $-3 \mod 5 = 2$.) □

5.2.2 Galois-Verbindungen aus Extraktionsfunktionen

Ferner sei $\beta : V \to D$ eine Funktion. Dann ist $(\mathcal{P}(V), \subseteq) \underset{\alpha_\beta}{\overset{\gamma_\beta}{\leftrightarrows}} (\mathcal{P}(D), \subseteq)$ mit

$$\begin{aligned} \alpha &: \mathcal{P}(V) \to \mathcal{P}(D) \\ \alpha(V') &:= \beta(V') = \{ \beta(v) \mid v \in V' \} \end{aligned}$$

und

$$\begin{aligned} \gamma &: \mathcal{P}(D) \to \mathcal{V} \\ \gamma(D') &:= \beta^{-1}(D') = \{ v \in V \mid \beta(v) \in D' \} \end{aligned}$$

eine *Galois-Verbindung.*

Beobachtung

Die oben definierte Kongruenzabstraktion ergibt sich aus der Extraktionsfunktion $z \mapsto z \mod k$. □

Oft ist $\mathcal{P}(V) = \mathcal{P}(State)$, wobei wie bisher $State = \mathbb{Z}^{\mathrm{Var}}$. Ist nun $\beta : \mathbb{Z} \to D$ als Extraktionsfunktion bekannt, ergibt sich eine Abstraktionsfunktion für ganz $\mathcal{P}(State)$.

Definition 5.6 (Liften von Extraktionsfunktionen)

Sei $State = \mathbb{Z}^{\mathrm{Var}}$ die Menge aller Variablenbelegungen und sei $\beta : \mathbb{B} \to D$. Durch *Liften von β auf State* entsteht die Abstraktion

$$\begin{aligned} \alpha &: \mathcal{P}(State) \to \mathcal{P}\left(D^{\mathrm{Var}}\right) \\ \alpha(\Sigma) &:= \{ \beta \circ \sigma \mid \sigma \in \Sigma \} \end{aligned}$$

Es lässt sich zeigen, dass α vollständig distributiv und damit eine Galois-Verbindung definiert. □

5.2.3 Komposition von Galois-Verbindungen

Sequentielle Komposition:

Idee: Führe Abstraktionsfunktionen nacheinander aus.

Seien $(L_1, \leq_1) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (L_2, \leq_2)$ und $(L_2, \leq_2) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (L_3, \leq_3)$ Galois-Verbindungen. Dann erhalten wir durch sequentielle Komposition die Galois-Verbindung: $(L_1, \leq_1) \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} (L_3, \leq_3)$.

Wir schreiben: $(\alpha_2, \gamma_2) \circ (\alpha_1, \gamma_1) := (\alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2)$.

Parallelkomposition von Galois-Verbindungen

Idee: Führe zwei Abstraktionen auf demselben Element aus.

Es gibt zwei Produkte.

a) Unabhängiges (direktes) Produkt Seien $(L, \leq_L) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (M_1, \leq_{M_1})$ und $(L, \leq_L) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (M_2, \leq_{M_2})$ zwei Galois-Verbindungen. Dann ist das *direkte Produkt*

$$(\alpha_1, \gamma_1) \times (\alpha_2, \gamma_2) := (\alpha, \gamma)$$

mit

$$\begin{aligned} \alpha : L &\to M_1 \times M_2 & \qquad \gamma : M_1 \times M_2 &\to L \\ \alpha(l) :&= (\alpha_1(l), \alpha_2(l)) & \qquad \gamma((m_1, m_2)) :&= \gamma_1(m_1) \sqcap \gamma_2(m_2) \end{aligned}$$

wieder eine Galois-Verbindung, und zwar zwischen L und $M_1 \times M_2$.

Das unabhängige Produkt ist schnell berechenbar, dafür aber ggf. unpräzise.

b) Abhängiges (Tensor-) Produkt Seien $(\mathcal{P}(V), \subseteq) \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} (\mathcal{P}(D_i), \subseteq)$, $i = 1, 2$, zwei Galois-Verbindungen. Dann ist auch das *Tensorprodukt*

$$(\alpha_1, \gamma_1) \otimes (\alpha_2, \gamma_2) = (\alpha, \gamma)$$

mit

$$\begin{aligned} \alpha : \mathcal{P}(V) &\to \mathcal{P}(D_1 \times D_2) & \qquad \gamma : \mathcal{P}(D_1 \times D_2) &\to \mathcal{P}(V) \\ \alpha(V') :&= \bigcup_{v \in V'} \alpha_1(\{v\}) \times \alpha_2(\{v\}) & \qquad \gamma(D') :&= \{ v \in V \mid \alpha_1(\{v\}) \times \alpha_2(\{v\}) \subseteq D' \} \end{aligned}$$

eine Galois-Verbindung, und zwar zwischen $\mathcal{P}(V)$ und $\mathcal{P}(D_1 \times D_2)$.

Das Tensorprodukt berücksichtigt das Zusammenspiel der Abstraktionfunktionen.

Beispiel 5.7 (Direktes Produkt vs. Tensor-Produkt)
Wir definieren die Extraktionenfunktionen *sign* und *parity* durch

$$\begin{aligned} \mathit{sign} &: \mathbb{Z} \to \{-, 0, +\} & \qquad \mathit{parity} &: \mathbb{Z} \to \{o, e\} \mathrel{\widehat{=}} \{\mathit{odd}, \mathit{even}\} \\ \mathit{sign}(x) :&= \begin{cases} +, & \text{falls } x > 0 \\ 0, & \text{falls } x = 0 \\ -, & \text{falls } x < 0 \end{cases} & \qquad \mathit{parity}(x) :&= \begin{cases} e, & \text{falls } x = 0 \mod 2 \\ o, & \text{falls } x = 1 \mod 2 \end{cases} \end{aligned}$$

Diese Funktionen induzieren Galois-Verbindungen:

$$(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha_{sign}}{\overset{\gamma_{sign}}{\leftrightarrows}} (\mathcal{P}(\{+, 0, -\}), \subseteq) \qquad \text{und} \qquad (\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha_{parity}}{\overset{\gamma_{parity}}{\leftrightarrows}} (\mathcal{P}(\{e, o\}), \subseteq).$$

Wir erhalten neuen Galois-Verbindungen

$$\mathcal{P}(\mathbb{Z}) \underset{\alpha_{precise}}{\overset{\gamma_{precise}}{\leftrightarrows}} \mathcal{P}(\{+, 0, -\} \times \{e, o\}) \qquad \text{bzw.} \qquad \mathcal{P}(\mathbb{Z}) \underset{\alpha_{imprecise}}{\overset{\gamma_{imprecise}}{\leftrightarrows}} \mathcal{P}(\{+, 0, -\}) \times \mathcal{P}(\{e, o\})$$

durch

$$(\alpha_{precise}, \gamma_{precise}) := (\alpha_{sign}, \gamma_{sign}) \otimes (\alpha_{parity}, \gamma_{parity})$$

bzw.

$$(\alpha_{imprecise}, \gamma_{imprecise}) := (\alpha_{sign}, \gamma_{sign}) \times (\alpha_{parity}, \gamma_{parity}).$$

Sei die Menge $\{-4,3\} \subseteq \mathbb{Z}$ gegeben. Wir erhalten

$$\begin{aligned}\alpha_{imprecise}(\{-4,3\}) &= \alpha_{sign}(\{-4,3\}) \times \alpha_{parity}(\{-4,3\}) \\ &= \{-,+\} \times \{e,o\} \\ &= \{(-,e),(+,e),(-,o),(+,o)\}\end{aligned}$$

sowie

$$\begin{aligned}&\alpha_{precise}(\{-4,3\}) \\ &= (\alpha_{sign}(\{-4\}) \times \alpha_{parity}(\{-4\})) \cup (\alpha_{sign}(\{3\}) \times \alpha_{parity}(\{3\})) \\ &= (\{-\} \times \{e\}) \cup (\{+\} \times \{0\}) \\ &= \{(-,e),(+,o)\}\end{aligned}$$

□

Komponentenweise Kombinationen von Galois-Verbindungen

Oft ist die Zustandsmenge eines Programms als kartesiches Produkt vollständiger Verbände gegeben (verschiedene Variablen). Sind nun auf den Komponenten Galois-Verbindungen definiert, lass sich diese zu neuen Galois-Verbindungen für das kartesische Produkt verknüpfen.

Wir erhalten wieder zwei verschiedene Produkte.

a) Direktes Produkt Seien $(L_1, \leq_{L_2}) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (M_1, \leq_{M_1})$ und $(L_2, \leq_{L_2}) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (M_2, \leq_{M_2})$ zwei Galois-Verbindungen. Dann ist das *direkte Produkt*

$$(\alpha_1, \gamma_1) \hat{\times} (\alpha_2, \gamma_2) := (\alpha, \gamma)$$

mit

$$\begin{aligned}\alpha &: L_1 \times L_2 \to M_1 \times M_2 & \gamma &: M_1 \times M_2 \to L_1 \times L_2 \\ \alpha((l_1,l_2)) &:= (\alpha_1(l_1), \alpha_2(l_2)) & \gamma((m_1,m_2)) &:= (\gamma_1(m_1), \gamma_2(m_2))\end{aligned}$$

wieder eine Galois-Verbindung, und zwar zwischen $L_1 \times L_2$ und $M_1 \times M_2$.

b) TensorProdukt Seien $(\mathcal{P}(V_i), \subseteq) \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} (\mathcal{P}(D_i), \subseteq)$, $i = 1,2$, zwei Galois-Verbindungen. Dann ist auch das *Tensorprodukt*

$$(\alpha_1, \gamma_1) \hat{\otimes} (\alpha_2, \gamma_2) = (\alpha, \gamma)$$

mit

$$\begin{aligned}\alpha &: \mathcal{P}(V_1 \times V_2) \to \mathcal{P}(D_1 \times D_2) \\ \alpha(V') &:= \bigcup_{(v_1,v_2) \in V'} \alpha_1(\{v_1\}) \times \alpha_2(\{v_2\}) \\ \gamma &: \mathcal{P}(D_1 \times D_2) \to \mathcal{P}(V_1 \times V_2) \\ \gamma(D') &:= \{(v_1,v_2) \in V_1 \times V_2 \mid \alpha_1(\{v_1\}) \times \alpha_2(\{v_2\}) \subseteq D'\}\end{aligned}$$

eine Galois-Verbindung, und zwar zwischen $\mathcal{P}(V_1 \times V_2)$ und $\mathcal{P}(D_1 \times D_2)$.

5.3 Abstrakte Semantik

Ziel: Imitiere die konkrete Semantik auf einer abstrakten Datendomäne.

Definition 5.8 (Sichere Approximation von Funktionen)
Sei $(L, \leq_L) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \leq_M)$ eine Galois-Verbindung. Sei ferner $f : L \to L$ eine Funktion.

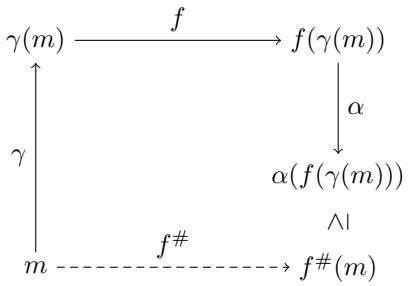
- Dann heißt $f^\# : M \to M$ *sichere Approximation von* f, falls

$$\alpha \circ f \circ \gamma \leq f^\#$$

  d.h. $\alpha(f(\gamma(m))) \leq_M f^\#(m)$ für alle $m \in M$.

- Funktion $f^\#$ heißt *genaueste* sichere Approximation von f, falls $\alpha \circ f \circ \gamma = f^\#$. □

Illustration:



Bemerkung
Oft sind f und $f^\#$ monoton. □

Lemma 5.9
Falls f und $f^\#$ monoton, dann

$$\alpha \circ f \circ \gamma \leq_M f^\# \quad \text{gdw.} \quad \alpha \circ f \leq f^\# \circ \alpha$$ □

Beispiel 5.10 (Sichere Approximation)
Betrachte die Vorzeichenabstraktion $(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha_{sign}}{\overset{\gamma_{sign}}{\leftrightarrows}} (\mathcal{P}(\{-, 0, +\}), \subseteq)$.

- Sei f_{-2} die Substraktion von 2:

$$f_{-2} : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$$
$$f_{-2}(Z) := \{ z - 2 \mid z \in Z \}$$

- Definiere eine sichere Approximation von f_{-2} mittels

$$f^\#_{-2} : \mathcal{P}(\{-, 0, +\}) \to \mathcal{P}(\{-, 0, +\})$$
$$f^\#_{-2}(A) := \begin{cases} \varnothing, & \text{falls } A = \varnothing \\ \{-, 0, +\}, & \text{falls } + \in A \\ \{-\}, & \text{sonst} \end{cases}$$

- Es ist zu zeigen, dass für alle $A \subseteq \{-, 0, +\}$ gilt:

$$\alpha(f_{-2}(\gamma(A))) \subseteq f^{\#}_{-2}(A)$$

  Zum Beispiel:

$$\begin{aligned}\alpha(f_{-2}(\gamma(\{0,+\}))) &= \alpha(f_{-2}(\{0,1,2,3,\ldots\})) \\ &= \alpha(\{-2,-1,0,1,\ldots\}) \\ &= \{-,0,+\} = f^{\#}_{-2}(\{0,+\})\end{aligned}$$

□

Als nächstes definieren wir nun die operationelle Semantik von While-Programmen auf einer abstrakten Datendomäne. Beachte dabei, dass die Transitionsrelation nicht-deterministisch wird:

$$(\texttt{if } x = 0 \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end}, \{\, (x = \text{even}) \,\})$$

Hier kann die Bedingung sowohl wahr als auch falsch sein.

Die abstrakte Semantik soll eine sichere Approximation der konkreten Semantik sein. Dazu müssen wir die konkrete Semantik als Funktion auf $\mathcal{P}(State)$ auffassen (also als *State Transformer*), wie folgt:

$$post_{c,c'}, post_c : \mathcal{P}(State) \to \mathcal{P}(State)$$

für alle $c, c' \in Prog$

$$\begin{aligned}post_{c,c'}(S) &:= \{\, \sigma' \in State \mid \exists\, \sigma \in S.\ (c,\sigma) \to (c',\sigma') \,\} \\ post_c(S) &:= \{\, \sigma' \in State \mid \exists\, \sigma \in S.\ (c,\sigma) \to \sigma' \,\}\end{aligned}$$

Definition 5.11 (Abstrakte Semantik)
Betrachte die Galois-Verbindung $(\mathcal{P}(State), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (M, \leq_M)$.

- Eine *abstrakte Semantik* ist gegeben durch eine Familie von Funktionen (für alle $c, c' \in Prog$):

$$post^{\#}_{c,c'}, post^{\#}_c : M \to M$$

  mit

$$\begin{aligned}\alpha \circ post_{c,c'} \circ \gamma &\leq_M post^{\#}_{c,c'} \\ \alpha \circ post_c \circ \gamma &\leq_M post^{\#}_c\end{aligned}$$

- Sind alle $post^{\#}_{c,c'}, post^{\#}_c$ genaueste sichere Approximationen (d.h. in der obigen Ungleichung gilt immer Gleichheit), spricht man von der *genauesten* abstrakten Semantik.
- Die abstrakte Semantik induziert die *abstrakte Transitionsrelation*:

$$\Rightarrow \subseteq (Prog \times M) \times (Prog \times M) \cup M)$$

  zwischen *abstrakten Konfigurationen* $(c, m) \in \mathcal{P}rog \times M$ mittels

$$\begin{aligned}(c,m) &\Rightarrow (c', post^{\#}_{c,c'}(m)) \\ (c,m) &\Rightarrow post^{\#}_c(m)\end{aligned}$$

□

Beispiel 5.12 (genaueste abstrakte Semantik)
Betrachte $(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha_{parity}}{\overset{\gamma_{parity}}{\leftrightarrows}} (\mathcal{P}(\{\, n = odd, n = even \,\}), \subseteq)$. Es gilt:

$$\begin{array}{llllll}
(& n := 3*n+1 & , & \{(n = \text{odd})\} &) & \Rightarrow \quad \{(n = even)\} \\
(& n := 3*n+1, \{(n = \text{odd}) & , & (n = \text{even})\} &) & \Rightarrow \quad \{(n = \text{odd}), (n = \text{even})\} \\
(& \texttt{while } n \neq 1 \texttt{ do } c \texttt{ end} & , & \{(n = \text{odd})\} &) & \Rightarrow \quad \{(n = \text{odd})\} \\
(& \texttt{while } n \neq 1 \texttt{ do } c \texttt{ end} & , & \{(n = \text{odd})\} &) & \Rightarrow \quad (c; \texttt{while } n \neq 1 \texttt{ do } c \texttt{ end}, \{(n = \text{odd})\}) \\
(& \texttt{while } n \neq 1 \texttt{ do } c \texttt{ end} & , & \{(n = \text{even})\} &) & \not\Rightarrow \quad \{(n = \text{even})\} \\
(& \texttt{while } n \neq 1 \texttt{ do } c \texttt{ end} & , & \{(n = \text{even})\} &) & \Rightarrow \quad (c; \texttt{while } n \neq 1 \texttt{ do } c \texttt{ end}, \{(n = \text{even})\})
\end{array}$$

□

Lemma 5.13

Die genaueste abstrakte Semantik ist im Allgemeinen *nicht* berechenbar. (Ungenauere abstrakte Semantiken lassen sich immer herleiten.) □

Beweis (Idee)

- Betrachte die Galois-Verbindung $(\mathcal{P}(State), \subseteq) \underset{\alpha_{sign}}{\overset{\gamma_{sign}}{\leftrightarrows}} (\mathcal{P}(\{-, 0, +\}), \subseteq)$
- Sei die abstrakte Konfiguration: $(\texttt{if } n > 2 \vee x^n + y^n = z^n \texttt{ then } n := 1 \texttt{ else } n := -1 \texttt{ end}, \{(n = +, x = +, y = +, z = +)\})$
- Um zu entscheiden, ob n auf $+$ oder $-$ gesetzt wird, muss man entscheiden, ob es Belegungen von n, x, y, z in $\mathbb{N}\backslash\{0\}$ gibt, die Bedingung erfüllt. (Letzter Satz von Fermat: nein.)

Allgemein: Es ist unentscheidbar, ob eine *Diophantische Gleichung*

$$p(x_1, \ldots, x_n) = 0$$

mit p einem Polynom mit Koeffizienten in $\mathbb{Z}$ eine Lösung in $\mathbb{Z}$ hat. (Hilberts 10. Problem 1900, Unentscheibar nach Matijassewitsch 1970.) ■