Bäume, Ordnungen und Anwendungen

Juniorprof. Dr. Roland Meyer
geTEXt von Jonathan Kolberg

5. Februar 2014

Inhaltsverzeichnis

1 Verbände und der Satz von Knaster und Tarski

1.1 Verbände in der Programmanalyse

Ziel: Ermittle Menge der Zustände, die an einem Programmpunkt eingenommen werden können (aufgrund von verschiedener Ausführungen)

Ansch: Vereiningung über alle Zustände, die von Ausführungen erreicht werden, die zu diesem Punkt führen

Beispiel 1.1.1.

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

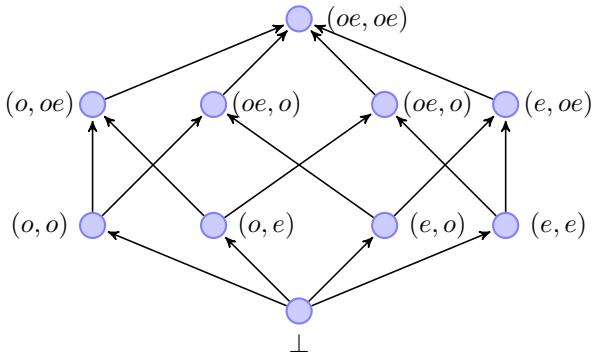
Es gibt nur eine Ausführung, die den Punkt 5 mehrfach erreicht und folgende Zustände erzeugt: $\{(6,2),(7,4),(8,6)\}$

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

Ansatz: Abstraktion

- Führe das Programm auf abstrakten Zuständen aus, *interpretiere* die Befehle in der abstrakten Domäne.
- Die konkreten Zustände an einem Punkt werden (neben anderen Zuständen) durch die abstrakten Zustände an diesem Punkt darstellt.
- 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-Verband

Beispiel 1.1.2. Vollständiger Verband der abstrakten Werte



(o,oe) repräsentiert *alle* konkreten Zustände mit

- p hat einen ungeraden Werte
- q hat irgendeinen Wert

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

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

Warum benötigen wir Fixpunkte?

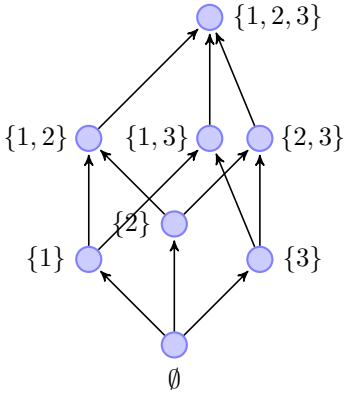
- Anstelle des JOP, berechne Fixpunkt von Funktionen auf dem Verband
- Unter weiteren Annahmen ist garantiert, dass der Fixpunkt JOP überapproximiert
- Satz von Knaster-Tarski sagt, wann Fixpunkte existieren und wie sie aussehen

1.2 Partielle Ordnungen und Verbände

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

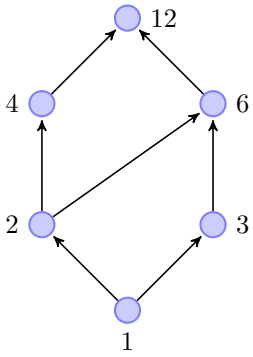
  Beispiel 1.2.1 (Teilmengen von $\{1, 2, 3\}$ & Teiler von 12)**.**

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



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

Teiler von 12



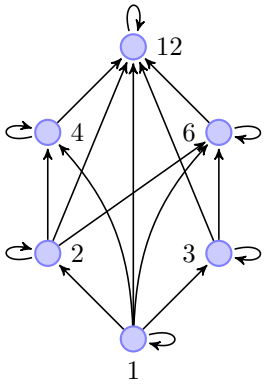
2 und 3 sind unvergleichbar.

Definition 1.2.2 (Partielle Ordnung)**.** Eine *partielle Ordnung* $(D, \leq)$besteht aus einer Menge $D \neq \emptyset$ und einer Relation $\leq \subseteq D \times D$ mit folgenden Eigenschaften

- reflexiv $(\forall d \in D : d \leq d)$
- transitiv $(\forall d, d', d'' \in D : d \leq d' \wedge d' \leq d'' \Rightarrow d \leq d'')$
- anti-symmetrisch $(\forall d, d' \in D : d \leq d' \wedge d' \leq d \Rightarrow d = d')$

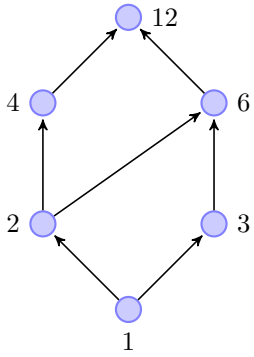
- Binäre Relationen lassen sich als *gerichtete Graphen* auffassen:
  $\{(a,a),(a,b),(b,c),(b,d),(d,c)\} =$
- Partielle Ordnungen liefern besondere Graphen
  Reflexivität = Schleifen an Knoten
  Anti-Symmetrie = keine nicht-trivialen Kreise
  Transitivität = Transitivität der Kanten

  Beispiel 1.2.3 (Teiler von 12)**.**



Hasse-Diagramm lässt

- Schleifen und
- induzierte Kanten weg



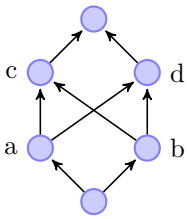
2 und 3 sind unvergleichbar.

Definition 1.2.4 (Join und Meet)**.**
Sei $(D, \leq)$ eine partielle Ordnung.

- Ein Element $o \in D$ heißt *obere Schranke* einer Menge $X \subseteq D$ falls $x \leq o$ für alle $x \in X$.
- Ein Element $o \in D$ heißt *kleinste obere Schranke* von $X \subseteq D$, auch *Join von* X genannt, falls
  - o ist obere Schranke und
  - $o \leq o'$ für alle oberen Schranken o' von X.

  Schreibe $o = \sqcup X$
- Analog ist $u = \sqcap X$ die *größte unter Schranke*, auch *Meet* von X genannt

Beispiel 1.2.5.



a und b haben

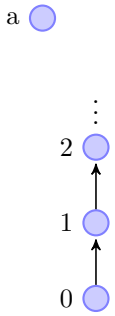
- c und d als obere Schranken
- aber keine kleinste obere Schranke

Definition 1.2.6 (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 $\sqcup\{a, b\}$.

– Ein Verband heißt *vollständig*, falls jede Teilmenge $X \subseteq D$ von Elementen Join und Meet hat.

Beispiel 1.2.7.



kein Verband

kein vollständiger Verband

Lemma 1.2.8.

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

$$\bot := \sqcup\emptyset = \sqcap D$$

(2) Ein vollständiger Verband hat ein eindeutiges größtes Element

$$\top := \sqcap\emptyset = \sqcup D$$

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

1.3 Monotone Funktionen und der Satz von Knaster und Tarski

Definition 1.3.1 (Monotone Funktionen und Fixpunkte)**.**

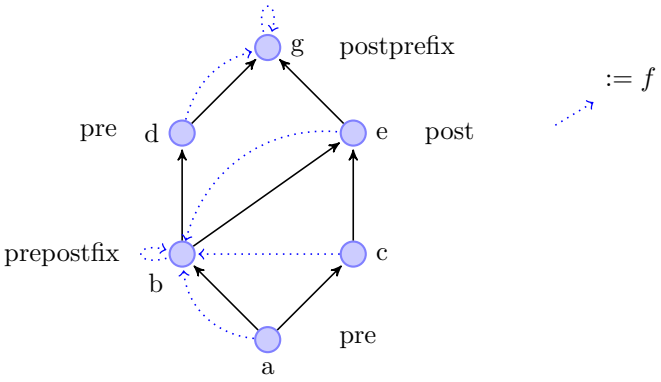
Sei $(D, \leq)$ eine partielle Ordnung.

- Eine Funktion $f : D \to D$ heißt *monoton*, falls

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

- Sei $f : D \to D$ eine Funktion auf einer partiellen Ordnung $(D, \leq)$
  - 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 1.3.2.



Satz 1.3.3 (Knaster und Tarski '55)**.**
Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.

(1) Dann besitzt f einen kleinsten *Fixpunkt, gegeben durch*

$$\mathrm{lfp}(f) := \sqcap \mathrm{Postfix}(f)$$

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

$$\mathrm{gfp}(f) := \sqcup \mathrm{Prefix}(f)$$

Beweis. Zeige die Behauptung für $\mathrm{lfp}(f)$.
Sei

$$l := \sqcap \mathrm{Postfix}(f)$$

Zeige zunächst

$$f(l) \leq l$$

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

$$f(l) \leq l$$

(*)
Zeige nun

$$l \leq f(l)$$

Mit (*) gilt:

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

Damit gilt

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

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

$$l = f(l)$$

□

1.4 Ketten

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

- Eine total geordnete Teilmenge $K \subseteq D$ heißt *Kette*

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

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

$$k_i \leq k_{i+1} \text{ f.a. } i \in \mathbb{N}$$

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

$$k_i \geq k_{i+1} \text{ f.a. } i \in \mathbb{N}$$

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

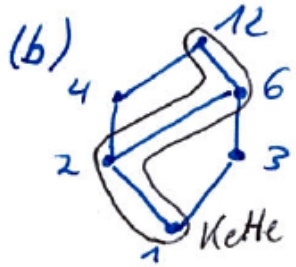
$$\exists n \in \mathbb{N} : \forall i \geq n : k_i = k_n$$

- $(D, \leq)$ 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 1.4.1.

(a) In $(\mathbb{N}, \leq)$ wird jede aufsteigende Kette stationär.

(b)



Definition 1.4.2 (Kettenbedingung)**.**
Eine partielle Ordnung $(D, \leq)$

- erfüllt die *aufsteigende Kettenbedingung (ACC)* (man sagt auch $(D, \leq)$ ist *Artisch*), falls jede aufsteigende Kette $k_0 \leq k_1 \leq \cdots$ stationär ist.
- erfüllt die *absteigende Kettenbedingung (DCC)* (man sagt auch $(D, \leq)$ ist *Noethersch*), falls jede aufsteigende Kette $k_0 \geq k_1 \geq \cdots$ stationär ist.

Beachte: ACC und DCC sind unabhängig von die Verbandsbedingungen.

Lemma 1.4.3. *Eine partielle Ordnung hat endliche Höhe gdw. (ACC) und (DCC) erfüllt sind*

Definition 1.4.4 (Stetigkeit)**.**
Sei $(D, \leq)$ ein vollständiger Verband. Eine Funktion $f : D \to D$ heißt

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

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

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

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

Satz 1.4.5 (Monotonie impliziert Stetigkeit)**.**
Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.

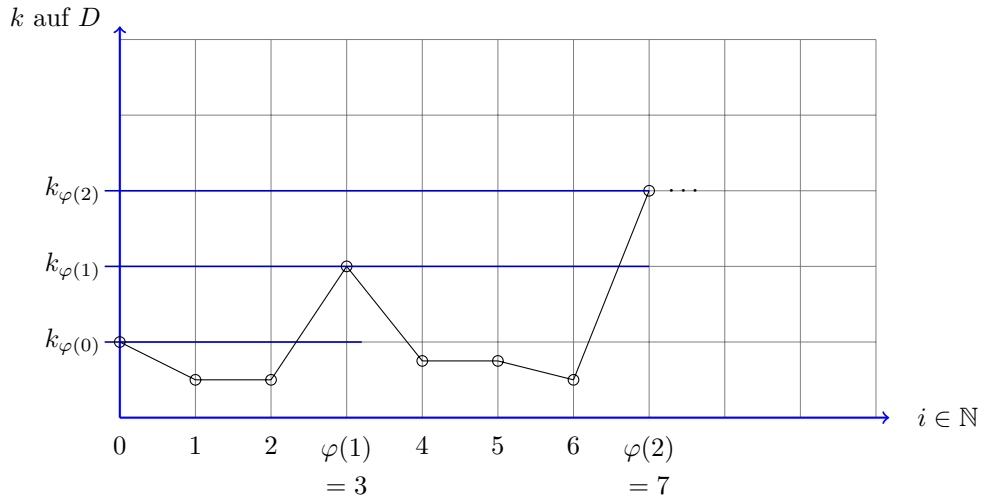
(i) Falls $(D, \leq)$ (ACC) erfüllt, dann ist f $\sqcup$-stetig.

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

Beweis. Sei K eine abzählbare Kette in D, die ohne Einschränkung als Folge dargestellt werden kann:

$$K = \{k_0, k_1, \cdots\} = (k_i)_{i \in \mathbb{N}}$$

Wir konstruieren eine aufsteigende Teilfolge $(k_{\varphi(i)})_{i \in \mathbb{N}}$ mit $k_{\varphi(0)} := k_0$ $k_{\varphi(i+1)} := k_j$ mit $j > \varphi(i)$ der kleinste Index, so dass $k_j \geq k_{\varphi(i)}$, falls es kein solches gibt wähle $k_{\varphi(i+1)} = k_{\varphi(i)}$.
Anschaulich:



- Es gilt $\sqcup K = \sqcup\{k_{\varphi(i)} | i \in \mathbb{N}\}$

  Für $\geq$,beachte $\{k_{\varphi(i)} | i \in \mathbb{N}\} \subseteq K$.

  Für $\leq$, beachte dass es für jedes Element $k \in K$ einen Index $i \in I$ gibt mit $k \leq k_{\varphi(i)}$.

- Mit (ACC) wird $(k_{\varphi(i)})_{i \in \mathbb{N}}$ stationär.

  Sei das entsprechende Element $k_{\varphi(n)}$

- Damit folgt

$$\begin{aligned} f(\sqcup K) &\overset{\text{oben}}{=} f(\sqcup\{k_{\varphi(i)} | i \in \mathbb{N}\}) \\ &\overset{\text{stationär}}{=} f(k_{\varphi(n)}) \\ &\overset{\text{Monotonie}}{=} \sqcup\{f(k_{\varphi(i)} | i \leq n\}) \\ &\overset{f(k_{\varphi(n+1)}) = f(k_{\varphi(n)})}{=} \sqcup\{f(k_{\varphi(i)} | i \leq \mathbb{N}\}) \\ &= \sqcup f(K) \end{aligned}$$

  Der letzte Schritt gilt, da es für jedes $k \in K$ ein $k_{\varphi(i)}$ gibt mit $k \leq k_{\varphi(i)}$.

  Per Monotonie folgt dann $f(k) \leq f(k_{\varphi(i)})$.

□

Lemma 1.4.6. *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. Zeige: $f^i(\bot) \leq f^{i+1}(\bot)$ f.a. $i \in \mathbb{N}$.
IA: $f^0(\bot) = \bot \leq f(\bot)$, da $\bot = \sqcap D$.
IS: Angenommen $f^i(\bot) \leq f^{i+1}(\bot)$, dann folgt

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

□

Satz 1.4.7 (Knaster, Tarski,Kleene)**.**
Sei $(D, \leq)$ ein vollständiger Verband und $f : D \to D$ monoton.

(i) Ist f $\sqcup$-vollständig, dann gilt

$$\text{lfp}(f) = \sqcup\{f^i(\bot) | i \in \mathbb{N}\}$$

(ii) Ist f $\sqcap$-vollständig, dann gilt

$$\text{gfp}(f) = \sqcap\{f^i(\top) | i \in \mathbb{N}\}$$

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

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

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

- Betrachte $d \in D$ mit $f(d) = d$ und zeige $\sqcup\{f^i(\bot) | i \in \mathbb{N}\}$ 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 $f^i(\bot) \leq d$
  Dann gilt:
  $$f^{i+1}(\bot) = f(f^i(\bot)) \overset{\text{IV+Mon.}}{\leq} f(d) = d$$
- Da $f^i(\bot) \leq d$ f.a. $i \in \mathbb{N}$ folgt
  $$\sqcup\{f^i(\bot) | i \in \mathbb{N}\} \leq d$$

□

Satz 1.4.8. *Sei $(D, \leq)$ ein vollständiger Verband mit (ACC) (DCC).*
Sei $f : D \to D$ monoton.
Dann ist

$$\begin{aligned} \mathrm{lfp}(f) &= \sqcup\{f^i(\bot) | i \in \mathbb{N}\} \\ &= f^n(\bot) \text{ mit } f^n(\bot) = f^{n+1}(\bot). \\ \mathrm{gfp}(f) &= \sqcap\{f^i(\top) | i \in \mathbb{N}\} \\ &= f^n(\top) \text{ mit } f^n(\top) = f^{n+1}(\top). \end{aligned}$$

Beweis. Aus Monotonie folgt Stetigkeit wegen (ACC) und (DCC).
Dann Knaster, Tarski und Kleene □

2 Datenflussanalyse

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

Ansatz: Fixpunktberechnung auf einer abstrakten Domäne

2.1 While-Programme

Definition 2.1.1 (Syntax beschrifteter While-Programme)**.**
Die *Syntax von beschrifteten While-Programmen* ist durch folgende BNF gegeben:

$$
\begin{aligned}
a ::= & k \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \\
& \text{// Arithmetische Ausdrücke} \\
b ::= & t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
& \text{// Boolsche Ausdrücke} \\
c ::= & [\text{skip}]^l \mid [x := a]^l \mid c_1; c_2 \\
& \mid \text{if } [b]^l \text{ then } c_1 \text{ else } c_2 \text{ fi} \\
& \mid \text{while } [b]^l \text{ do } c \text{ od} \\
& \text{// Programme}
\end{aligned}
$$

- Dabei sei $k \in \mathbb{Z}, t \in \mathbb{B}$ und $x \in \text{Var}$
- Ferner wird angenommen, dass alle Labels im Programm verschieden sind
- Beschriftete Befehle werden *Blöcke* genannt

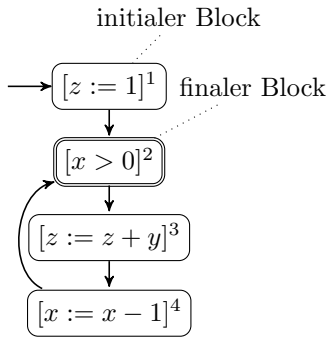
Programme lassen sich als *Kontrollflussgraphen* darstellen $G = (B, E, F)$, dabei ist

$$
\begin{aligned}
B &= \textit{Blöcke} \text{ im Programm} \\
E &= \text{Menge an } \textit{externalen Blöcken} \text{ (initial oder final)} \\
F &\subseteq B \times B = \text{Flussrelation}
\end{aligned}
$$

- Typischerweise repräsentieren Kontrollflussgraphen die Struktur eines Programms

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

gibt



- Es gibt jedoch Datenflussanalysen, die Programme entgegen der Befehlsfolge (rückwärts) analysieren (Live-Variables zum Beispiel). Daher werden wir bei einer Datenflussanalyse den zugrundeliegenden Kontrollflussgraphen genau festlegen.
- Für Kontrollflussgraphen wird 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 2.2.1 (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) oder (DCC))
- $i \in D$ ein *Anfangswert* für Extremalblöcke
- $f = \{f_b : D \to D | b \in D\}$ eine Familie von Funktionen, eine für jeden Block, die alle *monoton* sind.

Die Datenflussanalyse induziert ein *Gleichungssystem*

$$X_b = \begin{cases} i & \text{, falls } b \in E \\ \sqcup\{\varphi_{b'}(X_{b'}) | (b', b) \in F\} & \text{, sonst} \end{cases}$$

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

$$d'_b = \begin{cases} i & \text{, falls } b \in E \\ \sqcup\{\varphi_{b'}(d_{b'}) | (b', b) \in F\} & \text{, sonst} \end{cases}$$

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

$$\begin{aligned} g_s : D^{|B|} &\longrightarrow D^{|B|} \\ (d_1, \ldots, d_{|B|}) &\longmapsto (d'_1, \ldots, d'_{|B|}) \end{aligned}$$

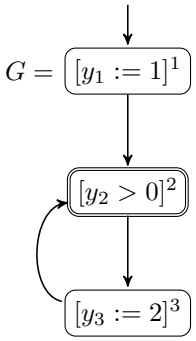
durch

$$d'_b = \begin{cases} i & \text{, falls } b \in E \\ \sqcup\{\varphi_{b'}(d_{b'}) | (b', b) \in F & \text{, sonst} \end{cases}$$

Satz 2.2.2. *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*

Beispiel 2.2.3. 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;
    od
```



Das zugehörige Datenflusssystem ist

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

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

$$\begin{aligned} X_1 &= \emptyset \\ X_2 &= \underbrace{X_1 \cup \{y_1\}}_{=\varphi_1(X_1)} \cup \underbrace{X_3 \cup \{y_3\}}_{\varphi_3(X_3)} \\ X_3 &= \underbrace{X_2}_{\varphi_2(X_2)} \end{aligned}$$

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

Beispiel 2.2.4. Weiter Beispiele zu Datenflusssystemen befinden sich in den Folien zur 3. Woche.

2.3 Join-over-all-paths

Bisher: Datenflussanalyse durch Lösung des Gleichungssystems, das von einem Datenflusssystem S induziert wird.

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 2.3.1. Sei $S = (G, (D, \leq), i, f)$ mit $G = (B, E, F)$ ein Datenflusssystem

- Für jeden Block $b \in B$ sei

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

Beispiel 2.3.2. $\text{paths}(b_3) = (b_1 b_2)^*$ 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

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

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

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

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

mit

$$X_b^{JOP} := \cup\{f_\pi(i) | \pi \in \mathrm{paths}(b)\}$$

Beispiel 2.3.3 (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;
    endif
    [z:=x+y]^6;
    [skip]^7
```

Führe 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:

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

Beachte:

- $\mathrm{paths}(b)$ ist typischerweise unendlich
- Es ist daher nich klar, ob X_b^{JOP} berechnet werden kann

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

Satz 2.3.4 (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 $\{1, \ldots, 9\}$

Frage:

- Gibt es eine Indexfolge $i_1 \dots i_k$ in $\{1, \dots, n\}$ mit $\underbrace{i_1 = 1}_{\text{macht das Problem nicht leichter}}$ so dass

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

Zur Reduktion sei $|u|$ für die Länge des Wortes u

Betrachte folgendes Programm:

```
x := u_1;
y := v_1;
while (...) do
  if (...) then
     x := x*10^|u_1| + u_1;
     y := y*10^|v_1| + v_1;
  else
    ....
  if (...) then
     x := x*10^|u_n| + u_n;
     y := y*10^|v_n| + v_n;
  else
     skip;
  endif
  ...
 endif
```

b = [z:=sign$((x-y)^2)$]^b

b$'$ = [skip]$'$

- 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 $X^{JOP}_{b'} = 1$ gdw. PCP hat keine Lösung □

Allerdings ist X^{LFP}_b immer eine sichere Approximation von X^{JOP}_b

Satz 2.3.5 (Zusammenhang zwischen LFP und JOP)**.**

Sei $S = (G, (D, \leq), i, f)$ ein Datenflusssystem mit $G = (B, E, F)$.
Sei $\mathrm{lfp}(g_S) = (X^{LFP}_{b_1}, \dots, X^{LFP}_{b_{|B|}})$ *die Fixpunktlösung und sei* $\mathrm{JOP}(S) = (X^{JOP}_{b_1}, \dots, X^{JOP}_{b_{|B|}})$ *die JOP-Lösung.*

a) Für alle $b \in B$ gilt $X^{JOP}_b \leq X^{LFP}_b$

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

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

Beweis. Zu (a): Definiere

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

Dann gilt:

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

Zeige nun:

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

Dann folgt

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

Induktion nach n

IA: Falls es einen Pfad der Länge 0 gibt, gibt $b \in$ Extremalknoten und so

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

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

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

□

3 Interprozedurale Datenflussanalyse

Ziel: Datenflussanalyse für rekursive Programmen

Problem: Berücksichtigung der Call-Return-Beziehung bei Prozeduraufrufen.

Idee: Berechne *JOVP-Lösung* (join-over-all-valid-paths)
Return zum richtigen Call-Block.

Zwei Tricks:

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 3.1.1 (Rekursives Programm)**.**

Ein *rekursives Programm* ist definiert als Folge von Prozeduren:

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

Keine Parameter, keine return-Werte.

Aber: alle Variablen in main() *global*, d.h. sichtbar innerhalb der Prozeduren.
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 $[\text{p}()]^{call}_{return}$ haben zwei Label.

Auch rekursive Programme werden als Kontrollflussgraph dargestellt:

- Gegeben Prozedur p in prog, sei

$$G_p := (B_p, E_p, F_p)$$

der Kontrollflussgraph, der wie bisher konstruiert wird.

- Dann ist

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

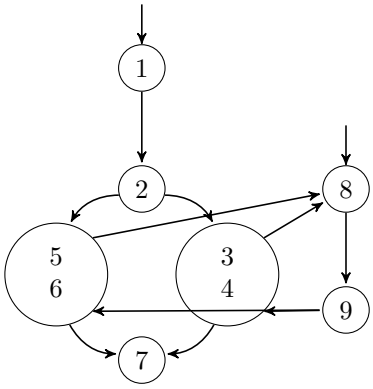
der *Kontrollflussgraph von prog.*

Dabei ist der *interprozeduale Flow* definiert als

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

Beispiel 3.1.2.

```
proc[main()]^1 begin
    if[(x)]^2 then
        [p()]^1_4;
    else
        [p()]^5_6;
    endif
[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 3.1.3 (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 3.1.4 (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

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

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

Korollar.

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

   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:

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

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

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

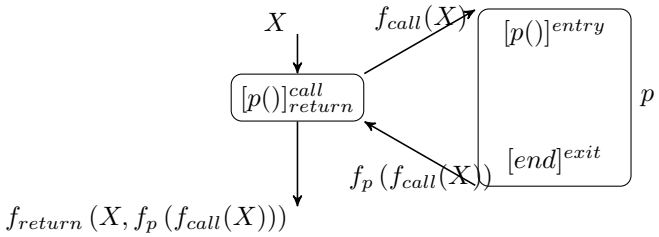
  darstellen.

  $$f_{call} : D \to D$$

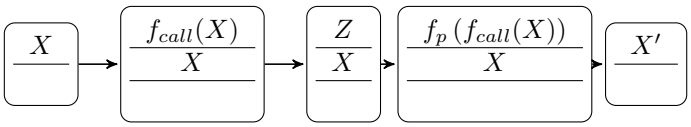
  - Initialisiere Datenflusswert bei Eintritt in die Prozedur
  - abhängig vom aktuellen Datenflusswert

    $$f_{return} : D \times D \to D$$

  - 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

  $$f_p \geq f_\pi \text{ für alle } \pi \in \text{validpaths}(entry, exit)$$

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

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

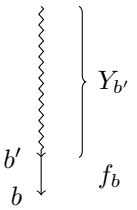
Definition 3.2.1. Sei $S = (G, (D, \leq), i, f)$ das Datenflusssystem eines rekursiven Programms.

Dann induziert S das *Summary-Gleichungssystem*:

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

Dabei ist

$$\begin{aligned} f_b &= \text{callret}(Y_q) : D \to D, \text{ falls } b = [q()]^{call}_{ret} \\ f_b &= \text{ wie in f angegeben, sonst} \end{aligned}$$



Dabei ist

$$\begin{aligned} \text{callret}(Y_q) &: D \to D \\ \text{callret}(Y_q)(d) &= f_{return}(d, Y_q(f_{call}(d))) \\ Y_p &= f_{exit} \circ Y_{exit} \end{aligned}$$

Satz 3.2.2. *Sei* $(Y_1, \ldots, Y_{|B|}) \in (\text{MonFun}(D \to D))^{|B|}$ *kleinste Lösung des Summary-Gleichungssystems.*

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

(b) $Y_p \geq f_\pi$ *für alle* $\pi \in \text{validpaths}(\text{entry}(p), \text{exit}(p))$

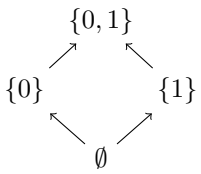
Problem: Monotone Funktionen müssen effektiv dargestellt werden

- Falls D endlich, nutze Wertetabelle
- Falls D unendlich, problematisch

Beispiel 3.2.3 (Realiable-Values-Analyes)**.**

Gegeben: Eine Boolesche Variable x

Berechne: Transferverhalten der Prozedur *pos*() auf der Wertemenge für x
Vollständiger Verband der Wertemengen:



Funktionen, die für Fixpunktberechnung relevant sind:

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

Summary-Gleichungssystem:

$$
\begin{aligned}
Y_2 &= f_{id} \\
Y_3 &= f_{id} \circ Y_2 \\
Y_4 &= f_0 \circ Y_3 \\
Y_5 &= f_{inv} \circ Y_4 \\
Y_6 &= Y_{pos} \circ Y_5 \\
Y_7 &= f_{id} \circ Y_2 \\
Y_8 &= f_{id} \circ Y_6 \sqcup f_1 \circ Y_7 \\
Y_{pos} &= f_{id} \circ Y_8
\end{aligned}
$$

Löse das Gleichungssystem durch Fixpunktiteration:

Iteration	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8	Y_{pos}
$g_S^0(\bot^9)$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$
1	f_{id}	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$
2	f_{id}	$f_\bot$	$f_\bot$	$f_\bot$	f_{id}	$f_\bot$	$f_\bot$	$f_\bot$
3	f_{id}	f_{id}	f_0	$f_\bot$	$f_\bot$	f_{id}	f_1[1]	$f_\bot$
4	f_{id}	f_{id}	f_0	$f_{inv(0)}$[2]	$f_\bot$	f_{id}	f_1	f_1
5	f_{id}	f_{id}	f_0	$f_{inv(0)}$	$f_{inv(0)}$	f_{id}	f_1	f_1
6	f_{id}	f_{id}	f_0	$f_{inv(0)}$	$f_{inv(0)}$	f_{id}	f_{make1}	f_1
7	f_{id}	f_{id}	f_0	$f_{inv(0)}$	$f_{inv(0)}$	f_{id}	f_{make1}	f_{make1}
8	f_{id}	f_iid	f_0	$f_{inv(0)}$	$f_{inv(0)}$	f_{id}	f_{make1}	f_{make1} $\surd$

[1] test1
[2] test

Sind die Transferfunktionen f_p berechnet, lässt sich auch für rekursive Datenflusssystem ein Gleichungssystem aufstellen

Betrachte $S = (G, (D, \leq), i, f)$ mit $G = (B, E, F)$ das *Gleichungssystem zur Datenflussanalyes* ist

$$X_b = \sqcup \{ f_{b'}(X_{b'}) \mid (b', b) \in F \}, \text{ falls } b \notin E$$

Dabei ist

- $f_{b'}(X_{b'})$ wie in f angegeben, falls b' gewöhlicher Block.
- $f_{b'}(X_{b'}) = f_{returnp}(X_{b'}, f_p(f_{callp}(X_{b'})))$, falls $b' = [p()]^{callp}_{returnp}$.

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

Satz 3.2.4 (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 $X_b^{JOVP} = X_b^{LFP}$ f.a $b \in B$

Korollar. *Das* Control-State-Reachability-Problem

Gegeben: *Rekursives, Boolesches Programm prog und ein Block b in prog.*

Frage: *Gibt es einen Ablauf, der zu b führt? (Ist entscheidbar!)*

3.3 Der Call-String-Ansatz

Kontext (in)sensitiv

- Der funktionale Ansatz berücksichtigt *keine* Information über den Kontext, in dem eine Prozedur aufgerufen wird.
- Falls $p()$ von $q()$ aus aufgerufen wird, könnte x immer 1 sein.
- Das macht die Analyes gegebenfalls unpräzise.

Ziel: Entwickle eine kontextsensitive interprozedurale Datenflussanalye.

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

- Sei $(D, \leq)$ der vollständige Verband der Datenflusswerte
- Sei Γ die Menge der Prozedurnamen im 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 also Call-Strings Datenflussinformationen zugewiesen.
Es lässt sich prüfen, dass $(\Gamma^* \to D, \leq^*)$ wieder vollständiger Verband ist.

Formal: Um ein gegebenes Datenflusssystem $S = (G, (D, \leq), i, f)$, unter Berücksichtigung von Call-Strings zu analysieren, modeliere

(1) den Initialwert und

(2) die Transferfunktionen

zu (1): Aus $i \in D$ wird

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

zu (2): Aus $f_b : D \to D$ wird

$\widetilde{f_b} : (\Gamma^* \to D) \to (\Gamma^* \to D)$ mit

$$\widetilde{f_b}(cs) := f_b \circ cs, \text{ falls } b \text{ gewöhnlicher Block}$$

$$\widetilde{f_{call_p}}(cs) := cs' \text{ mit} \qquad cs'(\gamma.p) := cs(\gamma)$$

$$cs'(\alpha) := \bot \text{ für } \gamma.p \neq \alpha \in \Gamma^*$$

Definition 3.3.1. 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 3.3.2. *Die Call-String-Lösung überapproximiert* JOVP(S).

Problem: Γ^* ist unendlich.

Ansätze:

Bounded Call-Strings:

- Stelle nur obersten n Elemente des Stacks dar, nutze also Call-Strings aus $\Gamma^{\leq n} := \cup_{i=0}^{n} \Gamma^i$
- In der Praxis 0 oder 1.
- 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.

Context-Information: Aus irgendeiner Menge I

Andere Sicht auf Call-Strings: Repliziere Prozedur p() für jeden Call-String.

4 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: (Patrick Cousot)

- *Führe* Programm auf abstrakten Werten *aus*
- Beispiele: $\mathcal{P}(\{-, 0, +\})$ anstatt $\mathbb{Z}$
- Berücksichtigte alle konkreten Eingaben
- *Ersetze* konkrete Operationen durch *abstrakte Operationen*

  Beispiel 4.0.3.

  $$\begin{aligned} op(x) &= x - 2 \\ op^{\#}(\{b\}) &= \{-, 0, +\} \\ op^{\#}(\{0\}) &= \{-\} = op^{\#}(\{-\}) \end{aligned}$$

  Es müssen alle Werte berücksichtigt werden, die durch den abstrakten Wert dargestellt sind.

Vorteile:

- *Mächtigkeit:*
  - Abstrakte Interpretation *unabhängig von Kontrollflussgraphen*
  - Funktioniert für viele Klassen von Programmen (if/while, parallel,Objektorientiert, Aktoren,funktional,...)
- *Korrektheit*
  - Durch Theorie garantiert
- Balance zwischen Präzesion und Komplexität
  - Wählbar durch Granularität der abstrakten Domäne.

Nachteile:

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

4.1 Galois-Verbindungen

Ziel:

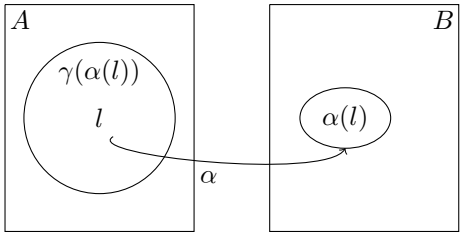
- Beschreibe geeignete *Abstraktionsfunktionen* $\alpha : L \to M$, die jedem *konkreten* Wert in L einen *abstrakten* Wert in M zuordnet.
- Definiere neben α auch *Konkretisierungsfunktion* $\gamma : M \to L$, die jedem abstrakten Wert alle konkreten Werte zuordnet, fzr die er steht.

Definition 4.1.1 (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

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

Anschaulich: Seien $L = \mathcal{P}(A)$ mit $A =$ Menge konkreter Werte und $M = \mathcal{P}(B)$ mit $B =$ Menge abstrakter Werte.



$(G1)$ $l \subseteq \gamma(\alpha(l))$
α erzeugt Überapproximation
Kein Präzisionsverlust durch Abstraktion nach Konkretisierung.
Typisch: $l \neq \gamma(\alpha(l))$, aber $m = \alpha(\gamma(m))$

Beispiel 4.1.2 (Intervallabstraktion)**.**
Sei $L = (\mathbb{Z}, \subseteq)$ die konkrete Domäne der Teilmengen von $\mathbb{Z}$.
Sei $M = ((\mathbb{Z} \cup \{-\infty\} x (\mathbb{Z} \cup \{+\infty\}) \cup \{\emptyset\}, \subseteq)$ die abstrakte Domäme der Intervalle.
Definiere $\alpha : L \to M$ mittels $\alpha(Z) := \begin{cases} \emptyset, \text{ falls } Z = \emptyset \\ [\sqcap Z, \sqcup Z], \text{ sonst} \end{cases}$ $\gamma : M \to L$ mittels

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

Zum Beispiel gilt

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

Satz 4.1.3 (Eigenschaften von Galois-Verbindungen I)**.**
Sei (α,γ) *eine Galois-Verbindung mit* $\alpha : L \to M, \gamma : M \to L$.
Seien ferner $l \in L, L' \subseteq L, m \in M, M' \subseteq M$.

(1.) $\alpha(l) \leq_M m$ *gdw.* $l \leq_L \gamma(m)$ *(daraus folgt wieder, Galois-Verbindungen)*

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

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

α *ist eindeutig durch* γ *bestimmt:*

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

(3.) $\alpha(\sqcup L') = \sqcup\alpha(L')$ α *ist* vollständig distributiv *(auch* vollständig additiv*)*

Beweis. (1) Gelte $\alpha(l) \leq_M m$

$\Rightarrow$ Es folgt

$$l \overset{(G1)}{\leq} \gamma(\alpha(l)) \overset{Mon\gamma}{\leq} \gamma(m)$$

$\Leftarrow$ ähnlich

(2) Zeige: $\gamma(m) = \sqcup\{l \in L \mid \alpha(l) \leq m\}$ für jede Galois-Verbindung.
Zeige dazu $\leq_M$ und $_M \geq$.
zu $\leq_M$:
Falls $\alpha(l) \leq m$, dann $l \leq_L \gamma(m)$ mit (1.)
Damit ist

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

zu $_M \geq$:
Da $\alpha(\gamma(m)) \leq_M m$, gilt

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

Also

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

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

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

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

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

□

4.2 Konstruktion von Galois-Verbindungen

4.2.1 Kongruenzabstraktion

- Rechne mit abstrakten Werten
- Vergiss absolute Größe

Sei $L = (\mathcal{P}(\mathbb{Z}), \subseteq)$ $M = (\mathcal{P}(\{0, \ldots, k-1\}), \subseteq)$ mit $k \in \mathbb{N}\backslash\{0\}$
Dann ist $\alpha : L \to M$ als

$$\alpha(Z) := \{z \mod k \mid z \in Z\}$$

(Beachte, $-3 \mod 5 = 2$)
$\gamma : M \to L$ mittels

$$\gamma(M) := \{z \in \mathbb{Z} \mid z \equiv m \mod k, \text{ für ein } m \in M\}$$

4.2.2 Galois-Verbindung aus Extraktionsfunktionen

Sei $\beta : V \to D$ eine Funktion.
Dann ist (α, γ) mit

$$\alpha : \mathcal{P}(V) \to \mathcal{P}(D) \text{ und } \gamma : \mathcal{P}(D) \to \mathcal{V}$$

eine *Galois-Verbindung* mit

$$\alpha(V') := \{\beta(v) \mid v \in V'\}$$
$$\gamma(D') := \{v \in V \mid \beta(v) \in D'\}$$

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

- Oft ist $\mathcal{P}(V) = \mathcal{P}(State)$ mit $State = B^{\text{Vars}}$ Ist nun $\beta : B \to D$ als Extraktionsfunktion bekannt, ergibt sich eine Abstraktionsfunktion für ganz $\mathcal{P}(State)$

Definition 4.2.1 (Liften von Extraktionsfunktionen)**.**
Sei $State = B^{\text{Vars}}$ die Menge der Variablenbelegungen σ und sei $\beta : B \to D$
Durch *Liften von β auf State* entsteht die Abstraktion

$$\alpha : \mathcal{P}(State) \to \mathcal{P}\left(D^{\text{Vars}}\right)$$

mit

$$\alpha(\Sigma) := \{\beta \circ \sigma \mid \sigma \in \Sigma\}$$

Es lässt sich zeigen, dass α vollständig additiv und damit Teil einer Galois-Verbindung ist.

4.2.3 Komposition von Galois-Verbindungen

Sequentielle Komposition:
Führe Abstraktionsfunktionen hintereinander aus
Seien (α_1, γ_1) mit $\alpha_1 : L_1 \to L_2$ und $\gamma_1 : L_2 \to L_1$
und (α_2, γ_2) mit $\alpha_2 : L_2 \to L_3$ und $\gamma_2 : L_3 \to L_2$
Galois-Verbindungen Dann ist $(\alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2)$ wieder Galois-Verbindung zwischen $(L_1, \leq_1)$ und $(L_3, \leq_3)$

Komponentenweise-Kombination:

Unabhänige Attribute (Direkte Produkt):
Seien (α_i, γ_i) mit $\alpha_i : L_i \to M_i$ und $\gamma_i : M_i \to L_i$ $i = 1, 2$ Galois-Verbindungen.

$$\begin{aligned} \text{Dann ist auch } (\alpha, \gamma) \text{ mit } \alpha :& L_1 \times L_2 \to M_1 \times M_2 \\ & (l_1, l_2) \mapsto (\alpha_1(l_1), \alpha_2(l_2)) \\ \text{und } \gamma :& M_1 \times M_2 \to L_1 \times L_2 \\ & (m_1, m_2) \mapsto (\gamma_1(m_1), \gamma_2(m_2)) \end{aligned}$$

wieder Galois-Verbindung (zwischen $L_1 \times L_2$ und $M_1 \times M_2$

Nachteil: Kein Zusammenhang zwischen Abstraktionen

Relationale Methode(Tensor-Produkt):
Seien (α_i, γ_i) mit $\alpha_i : \mathcal{P}(V_i) \to \mathcal{P}(D_i)$ und $\gamma_i : \mathcal{P}(D_i) \to \mathcal{P}(V_i), i = 1, 2$ Galois-Verbindungen.
Dann ist

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

wieder Galois-Verbindung.

Parallelkomposition von Galois-Verbindungen:

Unabhäniges (direktes) Produkt:
Entspricht unabhängiger Attributmethode. Seien (α_i, γ_i) mit $\alpha_i : L \to M_i$ und $\gamma_i : M_i \to L, i = 1, 2$ Galois-Verbindungen.

Dann ist auch

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

wieder Galois-Verbindung.

Abhängiges (Tensor) Produkt:

- Entspricht der relationalen Methode
- berücksichtigt das Zusammenspiel der Abstraktionsfunktionen

Seien (α_i, γ_i) mit $\alpha_i : \mathcal{P}(V) \to \mathcal{P}(D_i)$ und $\gamma_i : \mathcal{P}(D_i) \to \mathcal{P}(V), i = 1, 2$ Galois-Verbindungen.

Dann ist

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

4.3 Konkrete (strukturierte operationelle) Semantik von while-Programmen

Wiederholung (FGdP): Strukturierte operationelle Semantik (Plotkin,'81).

Ziel: Definiere *operationelle* Semantik (SoS) von while.

Idee von SoS:

- *Zustände* eines Programmes haben (syntaktische) Struktur. Komposition atomarer Elemente mittels Menge von Operatoren
- Damit lassen sich *Beweissysteme* (Kalküle) nutzen, um das Verhalten von Zuständen zu definieren.

  Transition existiert gdw. sie im Beweissystem herleitbar ist.
- 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.

Wiederholung (Syntax von while-Programmen):

$$
\begin{array}{l}
\mathrm{a} ::= \mathrm{k} \mid \mathrm{x} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \\
\mathrm{b} ::= \mathrm{t} \mid a_1 = a_2 \mid a_1 < a_2 \mid \neg b \mid b_1 \vee b_2 \\
\mathrm{c} ::= \mathrm{skip} \mid \mathrm{x} := \mathrm{a} \mid c_1; c_2 \\
\qquad \mid \mathrm{if\ b\ then}\ c_1\ \mathrm{else}\ c_2\ \mathrm{fi} \\
\qquad \mid \mathrm{while\ b\ do\ c\ od}
\end{array}
$$

Seien $Sig = (Funk, \text{Präd})$ die Signatur des Programmes:
$Funk$ = genutze Funktionssymbole = $\{+/_2, -/_2, */_2\} \cup \{k/0\}$
Präd = genutze Prädikatssymbole = $\{>/_2\}$

- Die Semantik ordnet jedem syntaktischen Ausdruck eine Bedeutung zu.

  Dabei ist eine Bedeutung ein Element eines semantischen Bereichs.

- Formel ist der semantische Bereich gegeben als (logische) *Sig-Struktur:*

  $\mathcal{S} = (\underbrace{D}_{\text{Domäne}} , \underbrace{\mathcal{I}}_{\text{Interpretation}})$

  mit D = Menge von Elementen,

  und $\mathcal{I}$ einer Menge von Abbildungen:

  $$\text{Sei } f/_n \in Funk : \mathcal{I}(f) : D^n \to D$$

  $$\text{Sei } p/_n \in Funk : \mathcal{I}(p) : D^n \to \mathbb{B}$$

  Schreibe auch $f_{\mathcal{I}}$ oder $p_{\mathcal{I}}$

  Hier: $D = \mathbb{Z}, \mathcal{I}$ wie erwartet.

- Das Verhalten eines Programmes in einem Zustant hängt von *der Belegung der Variablen ab:*

  $$\sigma : \text{Vars} \to \mathbb{Z}$$

- Die Semantik von *arithmetischen (a) und boolschen (b) Ausdrücken* ist $\mathcal{S}[\![a]\!](\sigma) \in \mathbb{Z}$, bzw. $\mathcal{S}[\![b]\!](\sigma) \in \mathbb{B}$, wie in Logik definiert (dort Terme und Formeln genannt)

Definition 4.3.1 (Transitionsrelation zwischen Konfigurationen)**.**
$\to \subseteq (\mathcal{P}rog \times State) \times ((\mathcal{P}rog \times State) \cup State)$ ist die kleinste Relation, die folgenden Regeln genu"gt:

$$(skip, \sigma) \to \sigma$$

$$(x := a, \sigma) \to \sigma[x := \mathcal{S}[\![a]\!]\sigma$$

$$\text{seq1}\ \frac{(c_1, \sigma) \to \sigma'}{(c_1; c_2, \sigma) \to (c_2, \sigma')}$$

$$\text{seq2}\ \frac{(c_1, \sigma) \to (c_1', \sigma')}{(c_1; c_2, \sigma) \to (c_1'; c_2, \sigma)}$$

, falls $\mathcal{S}[\![b]\!]\sigma = true \quad (if b then c_1 else c_2 fi, \sigma) \to (c_2, \sigma)$

(iffalse) a"hnlich

$$\text{, falls } \mathcal{S}[\![b]\!]\sigma = true\ ^{\text{whiletrue}}\ \frac{}{(while b do c od, \sigma) \to (c; while b do c od, \sigma)}$$

$$\text{,falls } \mathcal{S}[\![b]\!]\sigma = false \quad (while b do c od, \sigma) \to \sigma$$

Lemma 4.3.2. *Die Transitionsrelation ist deterministisch, d.h für alle* $(c, \sigma) \in \mathcal{P}rog \times State$ *gilt*

$$(c, \sigma) \to k_1 \text{ und } (c, \sigma) \to k_2 \text{ impliziert } k_1 = k_2$$

Um Galois-Verbindungen zur Abstraktion von Zustandsmengen nutzen zu können, definiere:

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

mit

$$post_{c,c'}(State') := \{\sigma' \in State \mid \exists \sigma \in State' : (c, \sigma) \to (c', \sigma')\}$$
$$post_c(State') := \{\sigma' \in State \mid \exists \sigma \in State' : (c, \sigma) \to \sigma'\}$$

Jeder Befehl wird als Transformer von Zustandsmengen aufgefasst (à la Dijkstra).

4.4 Abstrakte Semantik

Ziel: Imitiere die konkrete Semantik, genauer $post_{c,(c')}$ auf einer abstrakten Datendomäne

Definition 4.4.1 (Sichere Approximation von Funktionen)**.**
Sei $L \alpha \gamma M$ 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 $\alpha(f(\gamma(m))) \leq_M f^\#(m)$ fu"r alle $m \in M$.

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

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

Lemma 4.4.2. *Falls f und $f^\#$ monoton, dann*

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

Beispiel 4.4.3 (Sichere Approximation)**.** Betrachte $\mathcal{P}(\mathbb{Z})\, \alpha_{sign}, \beta\mathcal{P}(\{-,0,+\})$ Vorzeichenabstraktion.

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

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

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

$$\begin{aligned} f^\#_{-2} : \mathcal{P}(\{-,0,+\}) &\to \mathcal{P}(\{-,0,+\}) \\ f^\#_{-2}(A) &:= \{-\}, \text{ falls } A \neq \emptyset \\ &\quad \cup \{0,+\}, \text{ falls } + \in A \end{aligned}$$

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

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

  Zum Beispiel:

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

- Definiere nun operationelle Semantik auf einer abstrakten Datendoma"ne.
- Beachte, dass Transitionsrelation nicht-deterministisch wird.

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

  Bedingung kann wahr oder falsch sein.

Definition 4.4.4 (Abstrakte Semantik)**.**
Betrachte die Galois-Verbindung $\mathcal{P}(()\, State)\alpha, \beta M$

- Eine *abstrakte Semantik* ist gegeben durch eine Familie von Funktionen (für alle $c, c' \in \mathcal{P}rog$):

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

  mit

$$\alpha \circ post_{c,(c')} \circ \gamma \leq_M post^\#_{c,(c')}$$

- Sind alle $post^{\#}_{c,(c')}$ genaueste sichere Approximationen, nenne dies genaueste abstrakte Semantik
- Die abstrakte Semantik induziert die *abstrakte Transitionsrelation*: $\Rightarrow \subseteq (\mathcal{P} \times M) \times ((\mathcal{P}rog \times M) \cup M)$ zwischen *abstrakten Konfigurationen* $(c, m) \in \mathcal{P}rog \times M$ mittels

$$(c, m) \Rightarrow (c', post^{\#}_{c,c'}(m)$$
$$(c, m) post^{\#}_{c}(m)$$

Beispiel 4.4.5 (genaueste abstrakte Semantik)**.**

$$m := 3 * m + 1, \{(n = \text{odd})\}) \Rightarrow \{(n = even)\}$$
$$n := 3 * n + 1, \{(n = \text{odd}), (n = \text{even})\}) \Rightarrow \{(n = \text{odd}), (n = \text{even})\}$$
$$(whilen \neq 1docod, \{(n = \text{odd})\}) \Rightarrow \{(n = \text{odd})\}$$
$$(whilen \neq 1docod, \{(n = \text{odd})\}) \Rightarrow (c; whilen \neq 1docod, \{(n = \text{odd})\})$$
$$(whilen \neq 1docod, \{(n = \text{even})\}) \not\Rightarrow \{(n = \text{even})\}$$
$$(whilen \neq 1docod, \{(n = \text{even})\}) \Rightarrow (c; whilen \neq 1docod, \{(n = \text{even})\})$$

Lemma 4.4.6. *Die genaueste abstrakte Semantik ist im Allgemeinen* nicht *berechenbar.*

Ungenauere abstrakte Semantiken lassen sich immer herleiten.

Warum?

- *Betrachte die Galois-Verbindung* $\mathcal{P}(State), \alpha_{sign}, \beta, \mathcal{P}(\{-, 0, +\})$
- *Sei die abstrakte Konfiguration:* $(ifn > 2 \vee x^n + y^n = z^n thenn := 1elsen := -1fi, \{(n = +, x = +, y = +, z = +)\})$
- *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 erfu"llt.* $\Rightarrow$ *Letzter Satz von Fermat: nein.*

Allgemein: *Es ist unentscheidbar, ob* 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 Matiyosevič 1970)*

4.5 Herleitung einer abstrakten Semantik

Ziel: Berechne abstrakte Semantik für Galois-Verbindung $(\alpha_\beta, \gamma_\beta)$, die durch Liften einer Extraktionsfunktion $\beta : \mathbb{Z} \to D$ sind. Also $\mathcal{P}(State) = \mathcal{P}\left(\mathbb{Z}^{\text{Vars}}\right) \alpha, \gamma \mathcal{P}\left(D^{\text{Vars}}\right)$

Problem:

- Werte Boolesche Ausdrücke auf der abstrakten Domäne aus $\mathcal{P}\left(\left(\right) D\right)$
- Benötigt sichere Approximation von Prädikaten

Lösung: Werte Approximation in *3-wertiger Logik* aus:

$$\left(\mathcal{P}\left(\left(\right)\mathbb{B}\right)\backslash\{\emptyset\}, \wedge_3, \vee_3, \neg_3\right)$$

$\wedge_3$	0	1	1 / 2
0	0	0	0
1	0	1	1/2
1/2	0	1/2	1/2

$\vee_3$	0	1	1 / 2
0	0	1	1/2
1	1	1	1
1/2	1/2	1	1/2

$\wedge_3$	
0	1
1	0
1/2	1/2

Definition 4.5.1 (Sichere Approximation von Prädikaten)**.**
Sei $p : \mathbb{Z}^n \to \mathbb{B}$ ein n-stelliges Prädikat, das auch auf Mengen verstanden werden kann:

$$p : \mathcal{P}\left(\mathbb{Z}\right)^n \to \mathcal{P}\left(\mathbb{B}\right)\backslash\{\emptyset\}$$

Dann heißt $p^{\#} : \mathcal{P}\left(D\right)^n \to \mathcal{P}\left(\mathbb{B}\right)\backslash\{\emptyset\}$ *sichere Approximation von* p, falls

$$p \circ \gamma_\beta \leq p^{\#}$$

Definition 4.5.2. Sei $\mathcal{S} = (\mathbb{Z}, I)$ und $(\alpha_\beta, \gamma_\beta)$ die Galois-Verbindung $\mathcal{P}\left(\mathbb{Z}\right) \alpha\gamma \mathcal{P}\left(D\right)$
Dann heißt $\mathcal{S}_{Abs}(\mathcal{P}\left(D\right), I^{\#})$ *abstrakte Sig-Struktur*: $f_{\mathcal{I}}^{\#} : \mathcal{P}\left(D\right)^n \to \mathcal{P}\left(D\right)$ ist sichere Approximation von $f_{\mathcal{I}} : \mathbb{Z}^n \to \mathbb{Z}$.
$p_{\mathcal{I}}^{\#} : \mathcal{P}\left(D\right)^n \to \mathcal{P}\left(\mathbb{B}\right)$ ist sichere Approximation von $p_{\mathcal{I}} : \mathbb{Z}^n \to \mathbb{B}$
Die Semantik Boolescher Ausdrücke in 3-wertiger Logik ist dabei $\sigma : \text{Vars} \to \mathcal{P}\left(D\right)$

$$\mathcal{S}_{Abs}[\![p(a_1, \ldots, a_n)]\!](\sigma) := p_{\mathcal{I}}^{\#}(\mathcal{S}_{Abs}[\![a_1]\!](\sigma), \ldots, \mathcal{S}_{Abs}[\![a_n]\!](\sigma))$$

$$\mathcal{S}_{Abs}[\![b_1 \vee b_2]\!](\sigma) := \mathcal{S}_{Abs}[\![b_1]\!](\sigma) \vee_3 \mathcal{S}_{Abs}[\![b_2]\!](\sigma)$$

Lemma 4.5.3. *Es gilt:* $\beta(\mathcal{S}[\![a]\!](\sigma) \in \mathcal{S}_{Abs}[\![a]\!](\sigma')$ *mit* $\sigma'(x) := \beta(\sigma(x))$

$$\mathcal{S}[\![b]\!](\sigma) \in \mathcal{S}_{Abs}[\![b]\!](\sigma')$$

Ist eine eine Sig-Struktur gegeben, enthält

$$(x := a, Abs) \Rightarrow \{\tau[x := d] \mid \tau \in Abs, d \in \mathcal{S}_{Abs} \sim a(\tau) \text{ mit } \tau'(x) := \{\tau(x)\}\}$$

$$(\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi, Abs}) \Rightarrow (c_1, Abs \backslash \{\tau \mid \mathcal{S}_{Abs}[\![b]\!](\tau') = \{0\}\})$$

$$(\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi, Abs}) \Rightarrow (c_2, Abs \backslash \{\tau \mid \mathcal{S}_{Abs}[\![b]\!](\tau') = \{1\}\})$$

Beachte: Bei bedingten Anweisungen werden die abstrakten Zustände entfernt, die auf jeden Fall die andere Verzweigung ausgeführt hätten.

Definition 4.5.4. $post^{\#}_{c,c'}(Abs') := \begin{cases} Abs', \text{ falls } (c, Abs) \Rightarrow (c', Abs') \\ \emptyset, \text{ sonst} \end{cases}$

$post^{\#}_{c}(Abs) := \begin{cases} Abs', \text{ falls } (c, Abs) \Rightarrow Abs' \\ \emptyset, \text{ sonst} \end{cases}$

Satz 4.5.5. *Diese Familie von Funktionen* $post^{\#}_{c(c')}$ *ist eine abstrakte Semantik, also*

$$\alpha_\beta \circ post_{c(,c')} \circ \gamma_\beta \leq post^{\#}_{c(,c')}$$

5 Prädikatenabstraktion und Abstraktionsverfeinerung

Problem: Programmeigenschaft lässt sich mit aktueller Abstraktion *nicht* zeigen:

- Programm verletzt die Eigenschaft wirklich.
- Abstraktion zu grob

Ziel: Entwicklung eines abstraktionsbasierten Programmanalyseverfahrens, das

- die Verletzing der Eigenschaft aufzeigt oder
- die Abstraktion selbstständig verfeinert

Gegeben ein Programm c mit Eigenschaft φ. $\rightsquigarrow$ Nichterreichbarkeit (z.B. Wechselweiser Ausschluss)

Idee: Prädikatenabstraktion

- Nutze Prädikte $p_1, \ldots, p_n \in FO$ um Zustände σ auf Bitvektoren $(0/1, \ldots, 0/1$ zu abstraieren (direktes Produkt)
- *Verfeinerung* geschieht durch Hinzufügen von Prädikten
- *Wahl der Prädikate*
  - Initial aus dem Programm abgeleitet
  - Dann aus dem Gegenbeispiel
  - keine Interaktion des Nutzers notwendig

Konzeptionell:

- Die Behandlung von *Daten* ist ein Problem der *Logik*.
- Die Behandlung des *Kontrollflusses* ist ein Problem der *Automatentheorie*.

$\Rightarrow$ Prädikatenabstraktion ist eine geeignete Schnittstelle.

- Logik als *universelle Sprache* alle bisherigen Abstraktionen lassen sich als Prädikate über geeigneter Signatur auffassen

5.1 Prädikatenabstraktion

Idee:

- Seien Prädikate $p_1, \ldots, p_n$ über Vars gegeben. Abstrahiere Zustände $\sigma : \text{Vars} \to \mathbb{Z}$ auf Boolesche Kombinationen von $p_1, \ldots, p_n$
- Beliebige Boolesche Kombination skaliert nicht.
- Nutze *kartesische Abstraktion* auf Konjunktionen

Definition 5.1.1.
- Ein *Prädikat* ist ein Boolescher Ausdruck b // Siehe Syntax von Programmen b,a,c
- Ein Zustand $\sigma : \text{Vars} \to \mathbb{Z}$ *erfüllt* b, $\sigma \vDash b$, falls $\mathcal{S}[\![b]\!](\sigma) = true$
- Prädikat q *ist schwächer als* p, $p \vDash q$ falls $\forall \sigma \in State : \sigma \vDash p$ impliziert $\sigma \vDash q$
  Man sagt auch p *ist stärker als* q.
- Prädikate p und q heißen äquivalent, $p \dashv\vDash q$, falls $p \vDash q$ und $q \vDash p$
- Sei $P = \{p_1, \ldots, p_n\}$ eine endliche Menge von Prädikaten und $\neg P := \{\neg p_1, \ldots, \neg p_n\}$
  Der *Prädikatenabstraktionsverband* ist

$$Abs(P) := (\{\wedge Q \mid Q \subseteq P \cup \neg P\}, \vDash)$$

  Schreibe $true := \wedge\emptyset$, $false := \wedge\{p_i, \neg p_i \ldots\}$.
  Elemente in $Abs(P)$ heißen *Cubes*

Lemma 5.1.2. *$Abs(P)$ ist vollständiger Verband.*

Beweis.
- $\bot = false, \top = true$
- $q_1 \sqcap q_2 = q_1 \wedge q_2$
- $q_1 \sqcap q_2 = \overline{q_1 \vee q_2}$ Dabei ist Formel $\overline{b}$ die stärkste Formel in $Abs(P)$, die aus b folgt.

$$\begin{aligned} \overline{b} &\dashv\vDash \wedge \{q \in Abs(P) \mid b \vDash q\} \\ &\dashv\vDash \wedge \{l \in P \cup \neg P \mid b \vDash l\} \end{aligned}$$

□

Beispiel 5.1.3. Sei $P = \{p_1, p_2, p_3\}$.

1. Für $q_1 := p_1 \wedge \neg p_2$ und $q_2 := \neg p_2 \wedge p_3$ gilt:

- $$q_1 \sqcap q_2 = p_1 \wedge \neg p_2 \wedge \neg p_2 \wedge p_3$$
$$p_1 \wedge \neg p_2 \wedge p_3$$

- $$q_1 \sqcup q_2 = \overline{q_1 \vee q_v}$$
$$= \overline{(p_1 \wedge \neg p_2) \vee (\neg p_2 \wedge p_3)}$$

- Für $q_1 = p_1 \wedge p_2$ sowie $q_2 = p_1 \wedge \neg p_2$
  - $q_1 \sqcap q_2 = p_1 \wedge p_2 \wedge p_1 \wedge \neg p_2| = |false$
  - $$q_1 \sqcup q_2 = \overline{(p_1 \wedge p_2) \vee (p_1 \wedge \neg p_2)}$$
$$| = |\overline{p_1 \wedge (p_2 \vee \neg p_2)}$$
$$| = |\overline{p_1}$$
$$| = |p_1$$

Definition 5.1.4. Die Galoisverbindung zur Prädikatenabstraktion ist definiert durch

$$\alpha : \mathcal{P}(State) \to Abs(P) \text{ und } \gamma : Abs(P) \to \mathcal{P}(State)$$

mit

$$\gamma(q) := \{\sigma \in State \mid \sigma \vDash q\}$$

Was ist die *Abstraktionsfunktion?*
Gegeben $\sigma \in State$, definiere

$$q_\sigma := \wedge\{l \in P \cup \neg P \mid \sigma| = l\}$$

Dann gilt für $State' \subseteq State$:

$$\alpha(State') := \sqcup\{q_\sigma \mid \sigma \in State'\}$$

Beispiel 5.1.5. Betrachte $P = \{p_1, p_2, p_3\}$ mit

5.2 Abstrakte Semantik zur Prädikatenabstraktion

Ziel:

- Sei Abs(P) fest
- Bestimme $\text{post}^{\#}_{c(,c')}$ mit

$$\alpha(post(\gamma(q))) \leq \overset{\#}{\underset{c(,c')}{\text{post}}}(q)$$

f.a. $q \in$ Abs(P)

Idee:

- Sei $x := a$ der Befehl, der von c zu c' führt
- Die Menge
$$\underset{c,c'}{\text{post}}(\gamma(q))$$
ist gegeben durch die *stärkste Nachbedingung*
$$sp(q, x := a) :$$
$$\underset{c,c'}{\text{post}}(\gamma(q)) = \{\sigma \in \text{State} \mid \sigma \vDash sp(q, x := a)\}$$
Dann ist
$$\alpha(\underset{c,c'}{\text{post}}(\gamma(q)))$$
$$= \alpha(\{\sigma \in \text{State} \mid \sigma \vDash sp(q, x := a)\})$$
$$= \sqcup\{q_\sigma \mid \sigma \vDash sp(q, x := a)\}$$
$$\overset{!}{=\!=} \overline{sp(q, x := a)}$$

  Gut: Es ist also keine *Konkretisierung* über σ und q_σ notwendig.

  Schlecht: Stärkste Nachbedingung dennoch wegen *Quantoren* nachteilig.

Definition 5.2.1 (Hoare-Tripel, stärkste Nachbedingung, schwächste Vorbedingung)**.**

- Ein *Hoare-Tripel* $\{b\}x := a\{p\}$ besteht aus zwei Prädikaten $b, p \in BExp$ und einen Programm, hier $c = x := a$. Man nennt b die *Vorbedingung* und p die *Nachbedingung* des Tripels.
- Das Hoare-Tripel ist *gültig*, falls $\forall\sigma \in \text{State}$ mit $\sigma \vDash b$ und $\forall\sigma' \in \text{State}$ mit $(x := a, \sigma) \to \sigma'$ gilt $\sigma' \vDash p$

  Stärkste Nachbedingung $b \in BExp$ und $x := a$, bezeichnen wir mit $sp(b, x := a)$ die *stärkste (begl. $\vDash$) Formel* p, für die $\{b\}x := a\{p\}$ gültig ist.

- Gegeben $p \in BExp$ und $x := a$, bezeichnen wir mit $wp(x := a, p)$ die schächste Formel b, für die $\{b\}x := a\{p\}$ gültig ist, *schwächste Vorbedingung (zwecks precondition)*

Satz 5.2.2 (Dijkstra '76, aus FGdP bekannt)**.**

- $sp(b, x := a) \nparallel\!\!\nparallel \exists x' : (b\{x'/x\} \wedge x = a\{x/a\}$
- $wp(x := a, p) \nparallel\!\!\nparallel p\{a/x\}$

Beobachtung: Beide Formeln existieren und lassen sich berechnen.

- Stärkste Nachbedingungen benötigen Quantoren, die für Tools ($\underbrace{\text{Satisfiability modulo theories-solver}}_{\text{SMT}}$)

Beispiel 5.2.3. (1)

$$b = y > 5$$
$$x := y + 3$$

$$\begin{aligned} sp(y > 5, x := y + 3) &\nparallel\!\!\nparallel \exists x' : (y > 5\{x'/x\} \wedge x = (y + 3)\{x'/x\}) \\ &\nparallel\!\!\nparallel \exists x' : (y > 5 \wedge x = y + 3) \\ &\nparallel\!\!\nparallel y > 5 \wedge x = y + 3 \end{aligned}$$

(2)

$$\begin{aligned} sp(x > 5 \wedge y > 3, x := x + y) &\nparallel\!\!\nparallel \exists x' : ((x > 5 \wedge y > 3\{x'/x\}) \wedge x = (x + y\{x'/x\})) \\ &\nparallel\!\!\nparallel \exists x' : (x' > 5 \wedge y > 3 \wedge x = x' + y) \end{aligned}$$

(3)

$$\begin{aligned} wp(x := y + 7, x > 5) &\nparallel\!\!\nparallel x > 5\{y + 7/x\} \\ &\nparallel\!\!\nparallel y + 7 > 5 \\ &\nparallel\!\!\nparallel y > -2 \end{aligned}$$

Satz 5.2.4 (Dijkstra '76)**.**
$sp(b, x := a) \vDash p$ *gdw.* $b \vDash wp(x := a, p)$

Zurück zur abstrakten Transitionsrelation Definiere: $\Rightarrow \subseteq (Prog \times \text{Abs}(P)) \times ((Prog \times \text{Abs}(P) \cup \text{Abs}(P))$
mittels:

$$\text{(x:=a,q)} \Rightarrow \overline{sp(q, x := a)}$$

$$\text{(skip,q)} \Rightarrow \text{q}$$

$$\text{(if } b \text{ then } c_1 \text{ else } c_2 \text{ fi, q)} \Rightarrow (c_1, \overline{q \wedge b})$$

$$\text{(if } b \text{ then } c_1 \text{ else } c_2 \text{ fi, q)} \Rightarrow (c_1, \overline{q \wedge \neg b})$$

$$\text{(while } b \text{ do } c \text{ od,q)} \Rightarrow \text{(c;while } b \text{ do } c \text{ od, } \overline{q \wedge b})$$

$$\text{(while } b \text{ do } c \text{ od,q)} \Rightarrow \text{(c;while } b \text{ do } c \text{ od, } \overline{q \wedge \neg b})$$

Beobachtung:

- Es lässt sich zeigen, dass diese Transitionsrelation die genaueste abstrakte Semantik induziert.

$$\alpha \circ \underset{c,c'}{\text{post}} \circ \gamma \leq \overset{\#}{\underset{c,c'}{\text{post}}}$$

  Dazu wird benötigt: $\overline{sp(b, x := a)} \# \sqcup \{q_\sigma \mid \sigma \vDash sp(b, x := a)\}$

- Eigentlich sind $q \wedge b$ stärkste Nachbedingungen bei Conditionals.

  Insbesondere werden in diesen Fällen die Guards dem Prädikat hinzugefügt.

- Abstrakte Konfigurationen der Form $(c, false)$ repräsentieren *keine* erreichbaren Konfigurationen der konkreten Semantik, da $\sigma \vDash false$ für keine Belegung.

  Die abstrakte Konfiguration $(c, false)$ können daher weggelassen werden.

Beispiel 5.2.5.

```
if (@*$[x>y]^1$*@) then
 while (@*$[y\neq 0]^2$*@) do
   (@*$[x:=x-1]^3$*@);
   (@*$[y:=y-1]^4$*@);
 od
 if (@*$[x>y]^5$*@) then
  (@*$[skip]^6$*@);
 else
  (@*$[skip]^7$*@);
 fi
else
 (@*$[skip]^8$*@);
fi
```

Behauptung:
Block 7 wird nie betreten, da $x > y$ Invariante der while-Schleife ist.
Beweise das Verhalten automatisch mit Prädikatenabstraktion für $p_1 = x > y$ $p_2 = x \geq y$
(if $[x > y]^1$ then ..., true)

- Es ist abzulesen, dass Block 7 *nicht* erreichbar ist

Problem: Wie *berechnet* man $\overline{sp(q, x := a)}$?

Satz 5.2.6 (Graf & Saidi, '97)**.**
$\overline{sp(q, x := a)} \# \wedge \{l \in P \cup \neg P \mid\vDash p \rightarrow wp(x := a, l)\}$

Beweis. $sp(q, x := a)$

$\overset{\text{letzte VL}}{=\!\!=} \wedge\{l \in P \cup \neg P \mid sp(q, x := a) \vDash l\}$

$\overset{\text{Dijkstra}}{=\!\!=} \wedge\{l \in P \cup \neg P \mid q \vDash wp(x := a, l)\}$

$=\!\!= \wedge \{l \in P \cup \neg P \mid\!\vDash q \to wp(x := a, l)\}$ □

Um zu checken, ob $\vDash q \to wp(x := a, l)$ gilt nutze SMT-Solver.

- Damit Solver terminiert, sollte die Formel in einem entscheidbaren logischen Fragment liegen.

Beispiel 5.2.7.

Theorie	Beschreibung	voll	quantorenfrei
T_E	Gleichheit	x	√
T_{PA}	Pearo-Arth	x	x
$T_{\mathbb{N}}$	Presburger-Arith (nur +)	√	√
$T_{\mathbb{Z}}$	-	-	-
$T_{\mathbb{R}}$	Reals mit + und ·	√	√
azyklische rek. Datenstrukturen			
Arrays			

5.3 Abstraktionsverfeinerung

Ziel: CEGAR-Loop

Probleme:

- Wie prüft man, ob Gegenbeispiel echt ist?
- Wie extrahiert man neue Prädikate aus einem spurious Gegenbeispiel?

Typische Eigenschaften c:

- Keine Division durch 0
- x wird nie negativ
- Bei Terminierung ist y gerade
- Bestimmte Befehle sind *nicht* erreichbar (Dead-Code)

Gemeinsamkeit:
Eigenschaft lässt sich als das Vermeiden einer Konfiguration (c, σ) mit $c = c_{\text{bad}}$ einem unerwünschten Programm formulieren.

Definition 5.3.1 (Gegenbeispiel)**.**
Betrachte die abstrakte Semantik des Programms c unter Abs(P).
Sei c_{bad} das unerwünschte Programm.

- Ein *Gegenbeispiel* ist eine Folge abstrakter Transitionen.

  $(c, \text{true}) \Rightarrow (c_1, q_1) \Rightarrow \ldots \Rightarrow (c_k, q_k)$

  mit $c_k = c_{\text{bad}}$ und $q_i| = /|\,\text{false}$ für ale $1 \leq i \leq k$

- Das Gegenbeispiel heißt *echt*, falls es Zustände $\sigma_0, \ldots, \sigma_k \in \text{State}$ gibt mit $\sigma_i \vDash q_i$ und

$$(c, \sigma_0) \to (c_1, \sigma_1) \to \ldots \to (c_k, \sigma_k)$$

- sonst spurious

Lemma 5.3.2. *Das Gebenbeispiel* $(c, \text{true}) \Rightarrow \ldots \Rightarrow (c_k, q_k)$ *ist* spurious *gdw. es Prädikate* $p_0, \ldots, p_k$ *mit*

- $p_0 = \text{true}$ *und* $p_k = \text{false}$
- *für alle* $1 \leq i \leq k$ *und alle* $\sigma, \sigma' \in \text{State}$ *mit* $\sigma \vDash p_{i-1}$ *und* $(c_{i-1}, \sigma) \to (c_i, \sigma')$ *gilt* $\sigma' \vDash p_i$

  $\{\text{true}\}c_1\{p_1\}c_2\{p_2\}\ldots\ \{\text{true}\}r\{\text{false}\}$

Intuition:

- *Sei* r *das azyklische Programm, das den Befehlen der Folge entspricht (dabei werden* if *und while zu assertb*
- *Dann ist* r spurious *gdw.* $\{\text{true}\}r\{\text{false}\}$
- *Dann gilt, falls*

$$\text{true} \nVdash wp(r, false) \text{ oder } sp(true, r) \nVdash \text{false}$$

  ein gültiges Hoare-Tripel ist.

Beweis.

- Definiere p_i als stärkste Nachbedingung, angefangen bei $p_0 = true$.
- Es wird p_i abhängig von p_{i-1} und den Axiomen der Transitionsrelation definiert.

(skip) $p_i := p_{i-1}$

(assign x:= a) $p_i := \exists x' : (p_{i-1}\{x'/x\} \wedge x = (a\{x'/x\}))$

(while/if-true) $p_i := p_{i-1} \wedge b$

(while/if-false) $p_i := p_{i-1} \wedge \neg b$

□

Beispiel 5.3.3.

$[x := z]^0$;
$[z := z + 1]^1$;
$[y := z]^2$;
if $[y = x]^3$ then
  $[skip]^4$;
else
  $[skip]^5$;

Eigenschaft: Block 4 nicht erreichbar.

Initiale Abstraktion: $P = \emptyset$, also $\text{Abs}(P) = \{\text{false}, \text{true}\}$.

(Spurious Gegenbeispiel):

$$(0, \text{true}) \Rightarrow (1, \text{true}) \Rightarrow (2, \text{true}) \Rightarrow (3, \text{true}) \Rightarrow (4, \text{true})$$

Konstruktion der Prädikate im Beweis des obigen Lemmas:

$$\begin{aligned}
p_0 &:= && \text{true} \\
p_1 &:= && \exists x' : (p_0\{x'/x\} \wedge x = (z\{x'/x\})) \\
&\mathrel{\#\!\!\#} && \exists x' : (\text{true} \wedge x = z) \mathrel{\#\!\!\#} x = z \\
p_2 &:= && \exists z' : (p_1\{z'/z\} \wedge z = (z + 1\{z'/z\})) \\
&\mathrel{\#\!\!\#} && \exists z' : (x = z' \wedge z = z' + 1) \\
p_3 &:= && \exists y' : (p_2\{y'/y\} \wedge y = (z\{y'/y\})) \\
&\mathrel{\#\!\!\#} && \exists z' : (x = z' \wedge z = z' + 1) \wedge y = z \\
p_4 &:= && p_3 \wedge y = x \\
&\mathrel{\#\!\!\#} && \exists z' : (x = z' \wedge z = z' + 1) \wedge y = z \wedge y = x \\
&\vDash && z = x + 1 \wedge y = z \wedge y = x \\
&\vDash && y = x + 1 \wedge y = x \\
&\vDash && \text{false}
\end{aligned}$$

Abstraktionsverfeinerung:

- Seien $p_1, \ldots, p_{n-1}$ die neu bestimmten Prädikate aus obigem Lemma
- Definiere $P' := P \cup \{p_1, \ldots, p_{n-1}\}$

Lemma 5.3.4.
Berechnet man die abstrakte Semantik von Programm c mit $\text{Abs}(P')$*, dann gilt es keine Folge*

$$(c, \text{true}) \Rightarrow (c_1, q_1') \Rightarrow \ldots \Rightarrow (c_k, q_k')$$

mit

- c_i *die Programme des vorherigen Gegenbeispiels*
- $q'_k| = /|\,\text{false}$

Am Beispiel:
$p = \{\underbrace{x = z}_{=:p_1}, \underbrace{\exists z' : (x = z' \wedge z = z' + 1)}_{=:p_2}, \underbrace{\exists z' : (x = z' \wedge z = z' + x) \wedge y = z}_{=:p_3}\}$

Verfeinere abstrakte Transitionsrelation:

$$\begin{aligned}(0, \text{true}) &\Rightarrow (1, p_1 \wedge \neg p_2 \wedge \neg p_3) \\ &\Rightarrow (2, \neg p_1 \wedge p_2) \Rightarrow (3, \neg p_1 \wedge p_2 \wedge p_3) \\ &\Rightarrow (4, \underbrace{\overline{\neg p_1 \wedge p_2 \wedge p_3 \wedge x = y}}_{\nVdash \text{false}})\end{aligned}$$

Beweis.
- Angenommen es gibt eine Folge:

  $$(c, \text{true}) \Rightarrow (c_1, q'_1) \Rightarrow \ldots \Rightarrow (c_k, q'_k)$$

  mit $q'_k| = /|\,\text{false}$
- Die abstrakte Semantik ist über stärkste Nachbedingung definiert:

  $$q'_{i+1} := \overline{sp(q', c_{i+1}}$$

  wobei c_{i+1} als Befehl aufgefasst wird.
- Da q'_1 die stärkste Nachbedingung von true und c_1 ist, die in $\text{Abs}(P')$ ausgedrückt werden kann, und da auch p_1 eine solche Nachbedingung ist, folgt $q'_1 \vDash p_1$
- Allgemein gilt:
  - Wenn $q'_i \vDash q_i$ und man bereits aus p_i folgern kann, dass nach c_{i+1} p_{i+1} gilt, kann man das auch aus der stärksten Nachbedingung von q'_i folgern:

    $$sp(q'_i, c_{i+1}) \vDash p_{i+1}$$

    $q'_i \vDash p_i$, also $sp(q'_i, c_{i+1}) \vDash sp(p_i, c_{i+1}) \vDash p_{i+1}$
  - Da q'_{i+1} die stärkste Formel ist, die aus $sp(q', c_{i+1})$ folgt und in $\text{Abs}(P')$ liegt, und da $p_{i+1} \in \text{Abs}(P')$, folgt

    $$q'_{i+1} \vDash p_{i+1}$$

    $q'_i \vDash p_i$, also $sp(q'_i, c_{i+1}) \vDash sp(p_i, c_{i+1}) \vDash p_{i+1}$
- Also $q'_i \vDash p_i$ für alle $0 \leq i \leq k$. Da aber $p_k \nVdash \text{false}$, muss $q_k \nVdash \text{false}$ ↯

□

5.4 Optimierungen

(1) CEGAR schlägt manchmal fehl:

```
[x := a]^0 ;
[y := b]^1 ;
while [¬(x = 0)]^2 do
   [x := x − 1]^3 ;
   [y := y − 1]^4 ;
od ;
if [a = b ∧ ¬(y = 0)]^5 then
   [skip]^6 ;
else
   [skip]^7 ;
fi
```

- Block 6 wird nicht erreicht
- CEGAR liefert unendliche Folge von Gegenbeispielen mit Prädikaten

$$x = a - k, y = b - k \text{ für alle } k \in \mathbb{N}$$

- Benötigt: *Schleifeninvariante:* $a = b \to x = y$
- Wird nicht berechnet

⇒ Prädikate unnötigt komplex und beziehen sich auf irrelevante Variablen.

Lösung: *Craig-Interpolante*

Definition 5.4.1.
Seien $b, p \in \text{BExp}$ mit $b \vDash p$.
Eine Formel $v \in \text{BExp}$

- mit $b \vDash r$ und $r \vDash p$
- und $\text{Vars}(r) \subseteq \text{Vars}(b) \cap \text{Vars}(p)$

heißt *Craig-Interpolante.*

Satz 5.4.2 (Logik)**.**
Craig-Interpolanten existieren in Aussagenlogik und in Forst-Order.
Sie lassen sich aus Resolutionsbeweisen ablesen.

(2) Anstatt Prädikate uniform für alle Blöcke zu verwenden, generiere Prädikate geziehlt pro Block.
⇒ Lazy abstraction, Ranjit Jhala, UC San Diego
⇒ Auch über Craig-Interpolanten

6 Bisimulationsäquivalenz und Simulationsordnung

Ziel:

- Untersuche Beziehung zwischen dem konkreten und dem abstrakten Transitionssystem
- Wann ist der Schluss $c_A \vDash \varphi \Rightarrow c_C \vDash \varphi$ gültig?

Technisch: Zwei grundlegende Relationen zwischen Kripke-Strukturen (zustandsbeschriftete Transitionssysteme)

Bisimulationsäquivalenz ($K_C \approx K_A$) K_C und K_A zeigen dasselbe Transitionsverhalten (inkl. Branching)

Gut: $K_C \approx K_A$ gdw. $\forall \varphi \in CTL^* : K_C \vDash \varphi$ gdw. $K_A \vDash \varphi$

Schlecht:

- Mit diesem starken Zusammenhang enthält K_A ähnlich viel Information, wie K_C.
- Deshalb unterscheidet sich die Größe *kaum*.

Simulationsordnung($K_C \leq K_A$): Die abstrakte Kripke-Struktur K_A kann das Transitionsverhalten nachahmen.

Schlecht:

$$\forall \varphi \in ACTL^* : K_A \vDash \varphi \Rightarrow K_C \vDash \varphi$$
$$\forall \varphi \in ECTL^* : K_C \vDash \varphi \Rightarrow K_A \vDash \varphi$$

Gut: Mit diesem schwächeren Zusammenhang benötigt K_A oft deutlich weniger Zustände als K_C.

6.1 Bisimulationsäquivalenz

Ziel: Spielcharakterisierung. Entscheidbarkeit und Minimierung

Definition 6.1.1 (Kripke-Struktur)**.**
Eine *Kripke-Struktur* ist ein Tupel $K = (AP, S, S_0, \to, l)$

- AP Menge *atomarer Formeln*
- S Menge aller Zustände mit *initialem Zustand* S_0
- $\to \subseteq SxS$ *Transitionsrelation*
- $l : S \to \mathcal{P}(AP)$ *Beschriftung*
- Es wird *Deadlock-Freiheit* angenommen: $\forall s \in S : \exists s' \in S : s \to s'$
- Eine Kripke-Struktur heißt *endlich*, falls AP und S endlich sind.
- Die *Menge aller Kripke-Struktur* ist $\mathcal{K}$

Definition 6.1.2 (Bisimulation und Bisimulationsäquivalenz)**.**
Seien $K = (AP, S, S_0, \to, l)$ und $K' = (AP, S', S'_0, \to', l')$ Kripke-Strukturen über AP.
Eine Relation $R \subseteq S \times S'$ heißt *Bisimulation* zwischen K und K', falls für alle $(s, s') \in R$ gilt:

(1) $l(s) = l'(s')$

(2) $\forall t \in S : s \to t \Rightarrow \exists t' \in S' : s' \to t' \wedge (t, t') \in R$

(3) $\forall t' \in S' : s' \to t' \Rightarrow \exists t \in S : s \to t \wedge (t, t') \in R$

Die Kripke-Strukturen heißen *bisimulationsäquivalent*, $K \approx K'$, falls es eine Bisimulation $R \subseteq S \times S'$ gibt, die die initialen Zustände verbindet:

$$\forall s_0 \in S_0 \exists s'_0 \in S'_0 : (s_0, s'_0) \in R$$

$$\forall s'_0 \in S'_0 \exists s_0 \in S_0 : (s_0, s'_0) \in R$$

Lemma 6.1.3. *Bisimulationsäquivalenz* $\approx \subseteq \mathcal{K} \times \mathcal{K}$ *ist eine Äquivalenzrelation.*

Beweis. Seine $K^i = (AP, S^i, S^i_0, \to^i, l^i)$ mit $i = 1, 2, 3$

Transitivität: Gelte $K^1 \approx K^2$ mittels R und $K^2 \approx K^3$ mittels R' Dann folgt $K^1 \approx K^3$ mittels $R' \circ R := \{(s, s'') \in S^1 \times S^3, \exists s' \in S^2 : (s, s') \in R \text{ und } (s', s'') \in R'\}$
Reflexivität und Symmetrie sind Übungen □

6.1.1 Spielcharakterisierung

- Eine *Konfiguration* ist ein Paar $(s, s') \in S \times S'$
- Es gibt zwei Spieler: *Attacker* und *Defender*
- Das Spiel verläuft in Runden In jeder Runde:
  - Wählt *Attacker* die rechte oder linke Seite der Konfiguration (s, s') und macht dort eine Zug, sagen wir $s \to t$
  - Nun wählt *Defender* einen Zug auf der anderen Seite, sagen wir $s' \to t'$, so dass $l(t) = l'(t')$.
  - Die Konfiguration ihrer nächsten Runde lautet (t, t')
- Eine *Partie* ist eine maximale Folge von Konfigurationen.
  - Attacker *gewinnt die Partie*, falls Defender nicht mehr auf eine Transition reagieren kann.
  - Verläuft Partie unendlich lange, gewinnt Defender.

Partie wird immer von einem der Spieler gewonnen, nie von beiden.

Satz 6.1.4.
Zustände $s \in S$ und $s' \in S'$ sind bisimulationsäquivalent gdw. Defender eine Gewinnstrategie hat.

6.1.2 Entscheidbarkeit

Wie entscheidet man, ob $K \approx K'$?

Idee:

- Zunächst ist die volle Relation $S \times S'$ ein Kanidat für Bisimulation
- Dann werden Paare $(s, s') \in S \times S'$ entfernt, bei denen Beschriftung oder Nachfolge nicht passen.

Definition 6.1.5.
Seine $K = (AP, S, S_0, \to, l)$ und $K' = (AP, S', S'_0, \to', l')$
Die Funktion:

$$f_\approx : \mathcal{P}(S \times S') \to \mathcal{P}(S \times S')$$

ist definiert durch ($Q \subseteq S \times S'$):

$$\begin{aligned} f_\approx(Q) := &\{(s, s') \in Q \mid l(s) = l'(s')\} \\ &\cap \{(s, s') \in Q \mid \forall t \in S : s \to t \Rightarrow \exists t' \in S' : s' \to t' \wedge (t, t') \in Q\} \\ &\cap \{ \text{ dual } \} \end{aligned}$$

Da $f_\approx$ monoton, ist

$$S \times S' \subseteq f_\approx(S \times S') \subseteq \dots \text{ absteigend}$$

Konvergiert gegen größten Fixpunkt

$$\text{gfp}(f_\approx) = f_\approx^k(S \times S') \text{ mit } f_\approx^k(S \times S') = f_\approx^{k+1}(S \times S')$$

Lemma 6.1.6.
Jeder Fixpunkt von $f_\approx$ ist eine Bisimulation, insbesondere gfp($f_\approx$).

Idee: Um gfp($f_\approx$) zur Entscheidbarkeit zu nutzen, zeige, dass jede Bisimulation in gfp($f \approx$) enthalten.

Lemma 6.1.7. *Jede Bisimulation $R \subseteq S \times S'$ ist ein Fixpunkt von $f_\approx$*

Korollar.
gfp($f_\approx$) *ist größte Bisimulation.*

Beweis.
- Da mit 6.1.7 jede Bisimulation $R \subseteq S \times S'$ Fixpunkt von $f_\approx$, gilt mit Def. des größten Fixpunkts:
$$R \subseteq \text{gfp}(f_\approx)$$
- 6.1.6 zeigt, dass gfp($f_\approx$) wieder Bisimulation ist.

□

Satz 6.1.8.
$K \approx K'$ *gdw.* gfp($f_\approx$) *verbindet die Startzustände*

6.2 Berechnungsbaumlogik CTL

Ziel: Beschreibe temporale Eigenschaften von Kripke-Strukturen.

Alternativen: LTL = *Linear-time (temporal) logic* (Pnueli '77) interpretiert über Berechnungen (Worten, ohne Branching)
CTL = *Compuation-tree logic* interpretiert über Berechnungsbäumen (mit Branching)
CTL* = Verallgemeinerung von beiden.

Definition 6.2.1 (Syntax von CTL)**.** Sei AP eine Menge atomarer Formeln.
Die *Menge der CTL-Formeln* ist wie folgt definiert:

$$\varphi ::= \text{p} \mid \neg\text{p} \mid \varphi_1 \vee \varphi_2 \mid \text{EX } \varphi \mid \text{EG } \varphi \mid \text{E } (\varphi_1 \mathcal{U} \varphi_2)$$

Außerdem verwenden wir folgende Abkürzungen:
true $:= p \vee \neg p$ für $p \in \text{AP}$ $\varphi \to \psi := \neg\varphi \vee \psi$

$$\begin{aligned} AX\varphi &:= \neg EX\neg\varphi \\ EF\varphi &:= E(\text{true}\,\mathcal{U}\varphi) \\ AF\varphi &:= \neg EG\neg\varphi \\ AG\varphi &:= \neg EF\neg\varphi \\ A(\varphi_1\mathcal{U}\varphi_2) &:= (\neg EG\neg\varphi_2) \wedge \neg E(\neg\varphi_2\mathcal{U}(\neg\varphi_1 \wedge \neg\varphi_2)) \end{aligned}$$

Intuitive Bedeutung:
EF φ : *Es gibt* einen Pfad, auf dem schließlich φ gilt
AG φ : Auf *allen* Pfaden gilt immer φ
EG φ : *Es gibt* einen Pfad, auf dem immer φ gilt
AF φ : Auf *allen Pfaden* gilt schließlich φ
E($\varphi_1\mathcal{U}\varphi_2$) : *Es gibt* einen Pfad, auf dem φ_1 gilt, bis φ_2 eintritt.

Definition 6.2.2 (Semantik von CTL)**.**
Sei $K = (\text{AP}, S, S_0, \to, l)$ eine Kripke-Struktur.
Die *Erfülltheitsrelation* $\vDash$ für CTL ist induktiv über den Aufbau der Formeln und relativ zu einem Zustand von K definiert:

$K, s \vDash p$	, falls $p \in l(s)$
$K, s \vDash \neg p$	, falls nicht $K, s \vDash \varphi$
$K, s \vDash \varphi_1 \vee \varphi_2$	, falls $K, s \vDash \varphi_1$ oder $K, s \vDash \varphi_2$
$K, s \vDash EX\varphi$	, falls es einen (unendlichen) Pfad $\Pi = s_0 s_1 s_2 \ldots$ mit $s_0 = s$ und $K, s \vDash \varphi$
$K, s \vDash EG\varphi$	, falls es einen (unendlichen) Pfad $\Pi = s_0 s_1 s_2 \ldots$ mit $s_0 = s$ und $K, s_i \vDash \varphi \forall_i$
$K, s \vDash E(\varphi_1\mathcal{U}\varphi_2)$	, falls es einen (unendlichen) Pfad $\Pi = s_0 s_1 s_2 \ldots$ mit $s_0 = s$ und auf diesem Pfand ein $j \geq 0$ gibt mit $K, s_i \vDash \varphi_1$ für alle $0 \leq i < j$ und $K, s_j \vDash \varphi_2$

6.2.1 Model-Checking CTL nach Emerson & Clarke

Das *Model-Checking-Problem* für CTL ist wie folgt definiert:

Gegeben: Endliche Kripke-Struktur K und CTL-Formel φ_0

Frage: $K \vDash \varphi_0$?
Löse allgemeinere Aufgabe:
Bestimme für jede Teilformel φ von φ_0, die Menge $S(\varphi)$ der Zustände, in denen φ gilt:

$$S(\varphi) := \{s \in S \mid K, s \vDash \varphi\}$$

Damit ist das Model-Checking-Problem:

$$S_0 \subseteq S(\varphi_0)$$

Intuitiv werden in der i-ten Iteration die Teilformeln der Tiefe $i \in \mathbb{N}$ betrachtet: Dabei ist die Tiefe:

$$\begin{aligned} d(p) &:= 0 && \text{, für alle } p \in \text{AP} \\ d(\neg\varphi) &:= 1 + d(\varphi) \\ d(\varphi_1 \vee \varphi_2) &:= 1 + max\{d(\varphi_1), d(\varphi_2)\} \\ d(EX\varphi) &:= 1 + d(\varphi) \\ d(EG\varphi) &:= 1 + d(\varphi) \\ d(E(\varphi_1 \mathcal{U} \varphi_2)) &:= 1 + max\{d(\varphi_1), d(\varphi_2)\} \end{aligned}$$

Mit $cl(\varphi_0)$ berechnen wir die Menge der Teilformeln von φ_0

Algorithmus:

Input: $K = (\text{AP}, S, S_0, \rightarrow, l)$ und φ_0

Output: Mengen $S(\varphi)$ für all $\varphi \in cl(\varphi_0)$

```
begin:
    for i=0 to d(φ0) begin
        for all φ ∈ cl(φ0) mit d(φ) = i do
            check(φ);
        od
    od
end
```

Der Unteralgorithmus Check(φ) ist also abhängig von der Form von φ.

(1) Check(p) mit $p \in \text{AP}$

(2) Check($\neg\varphi$):
$S(\neg\varphi) := S \backslash S(\varphi)$

(3) Check($\varphi_1 \vee \varphi_2$):
$S(\varphi_1 \vee \varphi_2) := S(\varphi_1) \cup S(\varphi_2)$

(4) Check(EX φ):
$S(EX\varphi) := \{s \in S \mid \exists t \in S(\varphi) : s \rightarrow t\}$

(5) Check($E(\varphi_1 \mathcal{U} \varphi_2)$):

nutze die Äquivalenz:

$$E(\varphi_1 \mathcal{U} \varphi_2) \dashv\vdash \varphi_2 \vee (\varphi_1 \wedge EX(E(\varphi_1 \mathcal{U} \varphi_2)))$$

Implementierung:

```
Q := φ;
Q':= S(φ2);
while Q ≠ Q' do
      Q := Q'
      Q':= Q ∪  {s ∈ S | K, s ⊨ φ1 ∧ ∃t ∈ Q : s → t};
od
S(E(φ1 U φ2) := Q
// Starte in S(φ2)
// Füge in Breitensuche rückwärts Zustände hinzu die φ1 erfüllen
```

(6) Check($EG\varphi$):

Hausaufgabe

6.2.2 Satz von Hennessy & Milner

Ziel: $K \approx K'$ gdw $\forall \varphi \in CTL : K \vDash \varphi$ gdw. $K' \vDash \varphi$

Lemma 6.2.3.
Falls $K \approx K'$, dann $\forall l \in CTL : K \vDash \varphi$ gdw. $K' \vDash \varphi$

Beweis. Seien $K = (\mathrm{AP}, S, S_0, \to, l)$ und $K' = (\mathrm{AP}, S', S_0', \to', l')$ mit $K \approx K'$ mittels R

Zeige für alle Zustandspaare $(s, s') \in R$ und alle CTL-Formeln

$$K, s \vDash \varphi \text{ gdw. } K', s' \vDash \varphi$$

Der Beweis wir mittels Induktion über die Struktur von CTL-Formeln geführt

IA: Da $(s, s') \in R$, folgt $l(s) = l'(s')$. Also $p \in l(s)$ gdw. $p \in l'(s')$. das heißt: $K, s \vDash p$ gdw. $K', s' \vDash p$

IS: Angenommen die Aussage gilt für φ

EG φ $K, s \vDash EG\varphi$ gdw. Es gibt eine Pfad $\Pi = s_0 s_1 s_2 \ldots$ mit $s = s_0$ und $K, s_i \vDash \varphi$ für alle $i \in \mathbb{N}$ $\overset{\text{entsprechende Pfade}}{\text{gdw.}}$ □