Vorlesungsnotizen: Bäume, Ordnungen und Anwendungen / Programmanalyse

Prof. Dr. Roland Meyer

geTEXt von Jonathan Kolberg & Sebastian Muskalla

18. März 2016

Inhaltsverzeichnis

Diese Mitschrift wurde im Wintersemester 2013/14 von Jonathan Kolberg während der Vorlesung angefertigt und im Wintersemester 2015/16 von mir aktualisiert und korrigiert. Falls ihr irgendwelche Fehler findet, bitte ich euch, mir diese mitzuteilen: muskalla@cs.uni-kl.de

Sebastian Muskalla, 18. März 2016

Literatur

Die Vorlesung folgt keiner der folgenden Quellen streng:

- F. Nielson, H. R. Nielson, C. Hankin: *Principles of Program Analysis.* Springer-Verlag, 2005
- U. P. Khedker, A. Sanyal, B. Karkare: *Data Flow Analysis - Theory and Practice.* CRC Press, 2009
- H. Seidl, R. Wilhelm, S. Hack: *Übersetzerbau - Analyse und Transformation.* Springer-Verlag, 2010 *
- R. Berghammer: *Ordnungen, Verbände und Relationen mit Anwendungen.* Springer Verlag, 2012 *
- G. Grätzer: *General Lattice Theory.* Birkhäuser, 2003
- G. Birkhoff: *Lattice Theory.* Providence, RI,1967

* = erhältlich als E-Book auf den Seiten der Universitätsbibliothek

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 (auf Grund von verschiedenen Ausführungen)

Ansatz:

Vereinigung über alle Zustände, die von Ausführungen erreicht werden, die zu diesem Punkt führen

1.1.1 Beispiel

```
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).

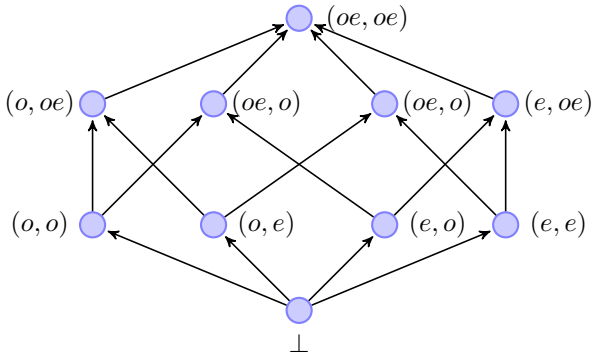
Ansatz:

Abstraktion

- Führe das Programm auf abstrakten Zuständen aus, *interpretiere* die Befehle in der abstrakten Domäne. Ziel ist es, das gewünschte Resultat in der abstrakten Domäne auszurechnen.
- 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.

1.1.2 Beispiel

Vollständiger Verband der abstrakten Werte



(o, oe) repräsentiert *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:

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

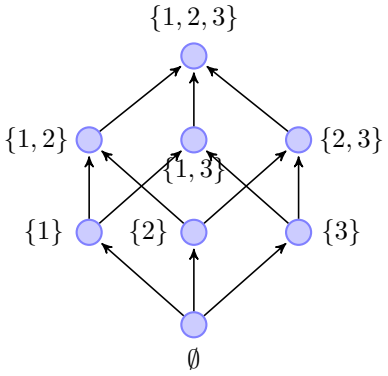
Warum benötigen wir Fixpunkte?

- 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 in diesem Fall können sie mit Kleene-Iteration berechnet werden

1.2 Partielle Ordnungen und Verbände

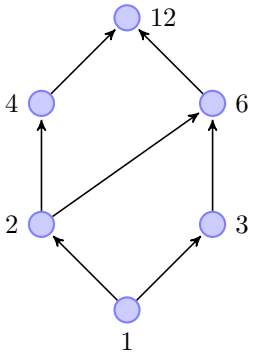
- $(\mathbb{N}, \leq)$ ist total geordnet: jeweils zwei Elemente sind in der Ordnung vergleichbar
- Einige Domänen sind nur partiell geordnet

1.2.1 Beispiel (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)



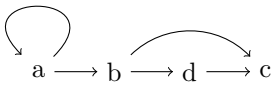
2 und 3 sind unvergleichbar.

1.2.2 Definition (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: $\forall d, d', d'' \in D : d \leq d' \wedge d' \leq d'' \Rightarrow d \leq d''$
- antisymmetrisch: $\forall d, d' \in D : d \leq d' \wedge d' \leq d \Rightarrow d = d'$

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

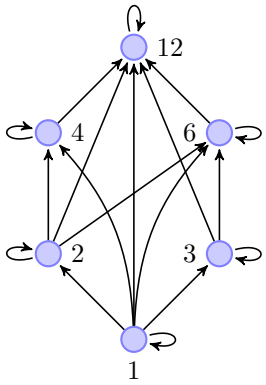
$$\{(a,a),(a,b),(b,c),(b,d),(d,c)\} =$$



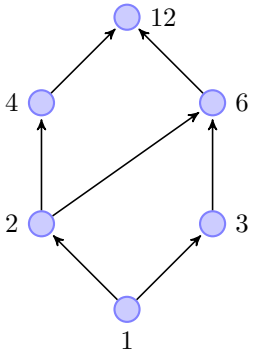
Partielle Ordnungen liefern besondere Graphen:

- Reflexivität = Schleifen an Knoten
- Antisymmetrie = keine nicht-trivialen Kreise
- Transitivität = Transitivität der Kanten

1.2.3 Beispiel (Teiler von 12)



Hasse-Diagramm lässt Schleifen und induzierte Kanten weg

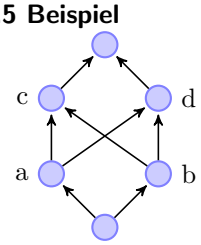


1.2.4 Definition (Join und Meet)
Sei $(D, \leq)$ eine partielle Ordnung und $X \subseteq D$.

- 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* von X (auch *Join von* X, Notation: $o = \sqcup X$), falls
  - o ist obere Schranke und
  - $o \leq o'$ für alle oberen Schranken o' von 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* von X (auch *Meet von* X, Notation: $u = \sqcap X$), falls
  - u ist obere Schranke und
  - $u' \leq u$ für alle unteren Schranken u' von 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 $o = o'$.

1.2.5 Beispiel



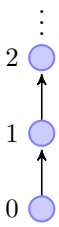
a und b haben
- c und d als obere Schranken
- aber keine kleinste obere Schranke

1.2.6 Definition (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 für jede Teilmenge $X \subseteq D$ von Elementen Join $\sqcup X$ und Meet $\sqcap X$ existieren.

1.2.7 Beispiel

kein Verband



kein vollständiger Verband

1.2.8 Lemma

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

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

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

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

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

1.3 Monotone Funktionen und der Satz von Knaster und Tarski

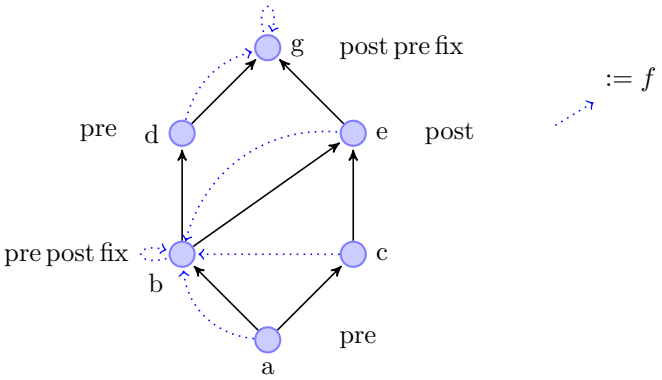
1.3.1 Definition (Monotone Funktionen und Fixpunkte)
Sei $(D, \leq)$ eine partielle Ordnung.

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

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

- 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 $f(x) \leq x$

1.3.2 Beispiel



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

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

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

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

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

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

Sei

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

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 $l = \sqcap \text{Postfix}(f)$

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

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

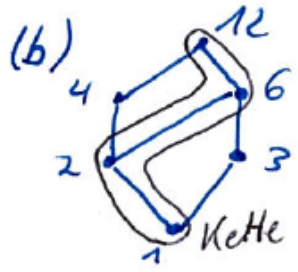
$$\exists n \in \mathbb{N} \colon \forall i \geq n \colon k_i = k_n$$

- $(D, \leq)$ 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.

1.4.1 Beispiel

(1)
In $(\mathbb{N}, \leq)$ wird jede absteigende Kette stationär.

(2)



1.4.2 Definition (Kettenbedingung)

Eine partielle Ordnung $(D, \leq)$

- erfüllt die *aufsteigende Kettenbedingung (ACC - ascending chain condition)*, falls jede aufsteigende Kette $k_0 \leq k_1 \leq \cdots$ stationär ist. (Man sagt auch $(D, \leq)$ ist *Artinsch*, nach Emil Artin.)
- erfüllt die *absteigende Kettenbedingung (DCC - descending chain condition)* falls jede absteigende Kette $k_0 \geq k_1 \geq \cdots$ stationär ist. (Man sagt auch $(D, \leq)$ ist *Noethersch*, nach Emmy Noether.)

Beachte: ACC und DCC sind unabhängig von den Verbandsbedingungen.

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

1.4.4 Definition (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

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

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

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

1.4.5 Satz (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:

$$f(\sqcup K) = \sqcup f(K).$$

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

" $\geq$ " 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

$$f(\sqcup K) = f(k') \leq \sqcup f(K).$$

□

1.4.6 Lemma
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:
$$\begin{array}{lcl} f^{i+1}(\bot) & & f(f^i(\bot)) \\ & \underset{\text{IV + Monotonie}}{=} & \\ & \leq & f(f^{i+1}(\bot)) = f^{i+2}(\bot) \end{array}$$

□

1.4.7 Satz (Knaster, Tarski, 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 von (1):
Zeige: $\sqcup\{f^i(\bot) | i \in \mathbb{N}\}$ ist Fixpunkt.

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

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 $f^i(\bot) \leq d$ 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

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

Der Beweis der zweiten Aussage funktioniert analog. □

1.4.8 Satz
Sei $(D, \leq)$ ein vollständiger Verband mit (ACC) und (DCC).

Sei $f : D \to D$ monoton.

Dann ist

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

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

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

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

$$
\begin{aligned}
a &::= k \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \\
&\quad // \text{ Arithmetische Ausdrücke, repräsentieren ganze Zahlen} \\
b &::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
&\quad // \text{ Boolsche Ausdrücke} \\
c &::= [\text{skip}]^l \mid [x := a]^l \mid c_1; c_2 \\
&\quad \mid \text{if } [b]^l \text{ then } c_1 \text{ else } c_2 \text{ end} \\
&\quad \mid \text{while } [b]^l \text{ do } c \text{ end} \\
&\quad // \text{ Programme, jeder Befehl hat ein Label } l
\end{aligned}
$$

- Dabei sei $k \in \mathbb{Z}, t \in \mathbb{B} = \{0,1\} = \{\mathit{false}, \mathit{true}\}$ und $x \in \text{Var}$
- Ferner wird angenommen, dass alle Labels im Programm verschieden sind
- Beschriftete Befehle werden *Blöcke* genannt

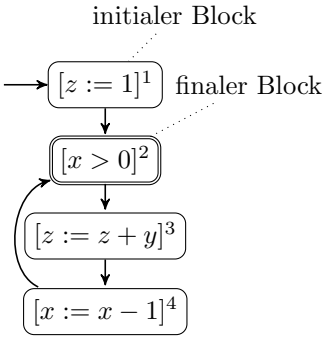
Programme lassen sich als *Kontrollflussgraphen* $G = (B, E, F)$ darstellen, dabei ist

$B =$ *Blöcke* im Programm
$E =$ Menge an *externalen Blöcken* (initial oder final)
$F \subseteq B \times B =$ Flussrelation

- 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



- 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.

2.2.1 Definition (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
- $f = \{f_b : D \to D | b \in D\}$ eine Familie von Funktionen, eine für jeden Block, die alle *monoton* sind.

Hinweis:

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*

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

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

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

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

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

durch

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

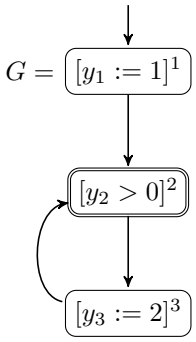
2.2.2 Satz
Vektor $\bar{d} = (d_1, \ldots, d_{|B|}) \in D^{|B|}$ löst das Gleichungssystem von S gdw. $g_s(\bar{d}) = \bar{d}$, d.h. $\bar{d}$ ist Fixpunkt von g_s

Beachte: Mittels Iteration kann der Kleinste Fixpunkt gefunden werden. Dieser liefert die präziseste Information.

2.2.3 Beispiel

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

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

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

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

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

2.2.1 Beispiele zu intraprozeduraler Datenflussanalyse

Klassifikation von Datenflussanalysen

Datenflussanalysen lassen sich anhand von vier Parametern klassifizieren:

Richtung der Analyse:

Vorwärts Berechne Information über die Vergangenheit von Daten.

Rückwärts Berechne Information über das zukünftige Verhalten von Daten.

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.

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.

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 ($\bot$)	$\emptyset$	AExp	$\emptyset$	AExp
Anfangsw. (i)	$\{(x, ?) \mid x \in \mathit{Vars}\}$	$\emptyset$	Vars	$\emptyset$
Transferf. (f)	$f_b(X) := (X \setminus \mathit{kill}(b)) \cup \mathit{gen}(b)$			

Reaching-Definitions-Analyse

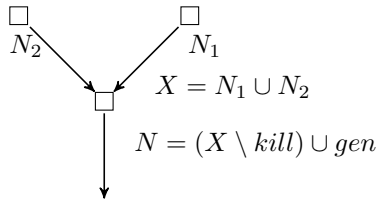
Ziel: Berechne für jeden Block die Zuweisungen, die es gegegeben haben könnte (nicht überschrieben), wenn eine Ausführung den Block erreicht.-

Klassifikation:

Vorwärtsanalyse, die Information über die Vergangenheit von Daten berechnet.

May-Analyse, die das Verhalten aller einzelnen Ausführungen überapproximiert.
Das heißt, das Verhalten jeder Ausführung ist sicher in der Information enthalten.

Idee:



Anwendungen: 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*.

2.2.4 Beispiel

Betrachte ein Programm mit Variablen *Vars* und Blöcken *Blocks*.

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

Kontrollflussgraph $G = (B, E, F)$:
$B = \mathit{Blocks}$, $E =$ initialer Block, $F =$ Kontrollfluss in Programmordnung.

Verband $(D, \preceq)$:
$(D, \preceq) = (\mathbb{P}(\mathit{Vars} \times (\mathit{Blocks} \cup \{?\})), \subseteq)$.
Es handelt sich um einen (Potenzmengen)Verband.
(ACC) gilt, da der Verband endlich ist.
Die Bedeutung der Elemente in $\mathit{Vars} \times (\mathit{Blocks} \cup \{?\})$ ist wie folgt:

$(x, ?)$ = 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 $f_b : D \to D$:

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

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

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

//Zuweisungen, die von Block b überschrieben werden.

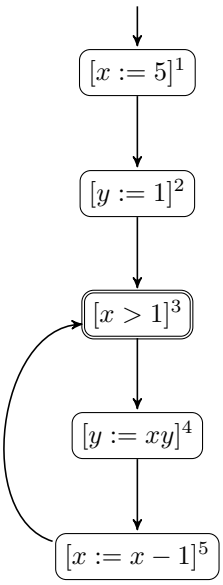
$$\mathit{gen}(b) := \begin{cases} \{(x, b)\}, & \text{falls } b = [x := a]^b \\ \emptyset, & \text{sonst.} \end{cases}$$

//Zuweisungen, die von Block b generiert werden.

Die Transferfunktionen sind monoton.

Betrachte das Beispielprogramm

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



Die Transferfunktionen sind

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

$$X_1 = \underbrace{\{(x,?),(y,?)\}}_{=i}$$

$$X_2 = \underbrace{(X_1 \setminus \{(x,?),(x,1),(x,5)\}) \cup \{(x,1)\}}_{=f_1(X_1)}$$

$$X_3 = \underbrace{((X_2 \setminus \{(y,?),(y,2),(y,4)\}) \cup \{(y,2)\})}_{=f_2(X_2)} \cup \underbrace{((X_5 \setminus \{(x,?),(x,1),(x,5)\}) \cup \{(x,5)\})}_{=f_5(X_5)}$$

$$X_4 = X_3$$

$$X_5 = (X_4 \setminus \{(y,?),(y,2),(y,4)\}) \cup \{(y,4)\}$$

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

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	$d_5)$
$g_S^0(\bot)$	$(\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset)$
$g_S^1(\bot)$	$(\{(x,?),(y,?)\}$	$\{(x,1)\}$	$\{(y,2),(x,5)\}$	$\emptyset$	$\{(y,4)\})$
$g_S^2(\bot)$	$(\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(y,2),(x,5)\}$	$\{(y,4)\})$
$g_S^3(\bot)$	$(\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,5)(y,4)\})$
$g_S^4(\bot)$	$(\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(x,5),(y,4)\})$
$g_S^5(\bot)$	$(\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(x,5),(y,4)\})$

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

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

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.

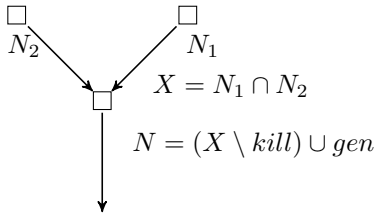
Available-Expressions-Analyse

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

Klassifikation:

Vorwärtsanalyse, die Information über die Vergangenheit von Daten berechnet.
Must-Analyse, die das gemeinsame Verhalten aller Ausführungen unterapproximiert.
Das heißt, die berechnete Information gilt definitiv für alle Ausführungen.

Idee:



Anwendungen: Vermeide erneute Berechnung bekannter Werte.

2.2.5 Beispiel

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 Datenflussystem $S = (G, (D, \preceq), i, \{f_b : D \to D \mid b \in Blocks\})$.

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 Potenzmengen)verband.
(ACC) gilt, da der Verband endlich ist.

Anfangswert i:
$\emptyset$.
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

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

//Teilausdrücke, die x enthalten und daher von Block b geändert werden.

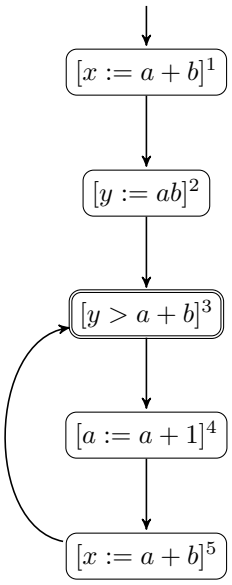
$$gen(b) := \begin{cases} \{a' \in AExp(a) \mid x \notin Vars(a')\}, & \text{falls } b = [x := a]^b \\ AExp(cond), & \text{falls } b = [cond]^b \\ \emptyset, & \text{sonst.} \end{cases}$$

//Teilausdrücke, die von Block b genutzt werden.

Beachte, dass bei einer Zuweisung, deren rechte Seite den zugewiesenen Wert beinhaltet, die entsprechenden Ausdrückte 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	$kill(b)$	$gen(b)$	$f_b(X)$
$[x := a + b]^1$	$\emptyset$	$\{a + b\}$	$X \cup \{a + b\}$
$[y := ab]^2$	$\emptyset$	$\{ab\}$	$X \cup \{ab\}$
$[y > a + b]^3$	$\emptyset$	$\{a + b\}$	$X \cup \{a + b\}$
$[a := a + 1]^4$	$\{a + b, ab, a + 1\}$	$\emptyset$	$X \setminus \{a + b, ab, a + 1\}$
$[x := a + b]^5$	$\emptyset$	$\{a + b\}$	$X \cup \{a + b\}$

Das vom Datenflusssystem induzierte Gleichungssystem ist

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

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

Berechne eine Lösung des Gleichungssystems durch Iteration von

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

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

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

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

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

Die kleinste Lösung des Gleichungssystems ist

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

Die kleinste Lösung ist die gewünschte Information.

Größere (bzgl. $\supseteq$) Must-Information bedeutet Informationsverlust.

Bemerkung: Wir haben hier den größten Fixpunkt auf dem Potenzmengenverband $(\mathbb{P}(\mathit{AExp}), \subseteq)$ berechnet.

Durch Dualisierung des Verbandes zu $(\mathbb{P}(\mathit{AExp}), \supseteq)$ konnten wir eine kleinste Fixpunktberechnung und so unser Framework mit (ACC) nutzen.

Live-Variables-Analyse

Definition: Eine Variable heißt *lebendig* 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.

Am Ende des Programms sind alle Variablen lebendig.

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

Klassifikation:

Rückwärtsanalyse, die Information über die Zukunft von Daten berechnet.

May-Analyse, die das Verhalten aller einzelnen Ausführungen überapproximiert.
Das heißt, das Verhalten jeder Ausführung ist sicher in der Information enthalten.

Anwendungen:

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.

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.

2.2.6 Beispiel

Betrachte ein Programm mit Variablen Blöcken *Blocks* und Variablen *Vars*.
Ferner sei $\mathit{Vars}(a)$ die Menge der Variablen in einem Ausdruck a.

Definiere das Datenflussystem $S = (G, (D, \preceq), i, \{f_b : D \to D \mid b \in \mathit{Blocks}\})$,
Kontrollflussgraph $G = (B, E, F)$:

$B = \mathit{Blocks}$, $E =$ finale Blöcke, $F =$ Kontrollfluss gegen die Programmordnung.

Verband $(D, \preceq)$:
$(D, \preceq) = (\mathbb{P}(\mathit{Vars}), \subseteq)$.

Es handelt sich um einen (Potenzmengen)verband.

(ACC) gilt, da der Verband endlich ist.

Anfangswert i:
Vars (am Ende des Programms sind per Definition alle Variablen lebendig).
Transferfunktionen $f_b : D \to D$:

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

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

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

//Variablen, die von Block b überschrieben werden.

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

//Variablen, die von Block b genutzt werden.

Die Transferfunktionen sind monoton.

Betrachte das Beispielprogramm c an der Tafel. (*Fehlt hier leider.*)

Die Transferfunktionen sind

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

Das vom Datenflusssystem induzierte Gleichungssystem ist

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

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

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	$d_7)$
$g_S^0(\bot)$	$(\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset)$
$g_S^1(\bot)$	$(\emptyset$	$\emptyset$	$\{y\}$	$\{y, x\}$	$\{z\}$	$\{z\}$	$\{x, y, z\})$
$g_S^2(\bot)$	$(\emptyset$	$\{y\}$	$\{y, x\}$	$\{y, x\}$	$\{y, z\}$	$\{y, z\}$	$\{x, y, z\})$
$g_S^3(\bot)$	$(\emptyset$	$\{y\}$	$\{y, x\}$	$\{y, x\}$	$\{y, z\}$	$\{y, z\}$	$\{x, y, z\})$

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

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

Die kleinste Lösung des Gleichungssystems ist

$$
\begin{aligned}
X_1 &= \emptyset & X_2 &= \{y\} \\
X_3 &= \{y, x\} = X_4 & X_5 &= \{y, z\} = X_6 \\
X_7 &= \{x, y, z\}.
\end{aligned}
$$

Die kleinste Lösung ist die gewünschte Information.

Größere May-Information bedeutet Informationsverlust.

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

Very-Busy-Expressions-Analyse

Definition: 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, die Information über die Zukunft von Daten berechnet.

Must-Analyse, die das gemeinsame Verhalten aller Ausführungen unterapproximiert.

Das heißt, die berechnete Information gilt definitiv für alle Ausführungen.

Anwendungen:

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.

2.2.7 Beispiel

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 Datenflussystem $S = (G, (D, \preceq), i, \{f_b : D \to D \mid b \in Blocks\})$.

Kontrollflussgraph $G = (B, E, F)$:
$B = Blocks$, $E =$ finale Blöcke, $F =$ Kontrollfluss gegen die Programmordnung.

Verband $(D, \preceq)$:
$(D, \preceq) = (\mathbb{P}(AExp), \supseteq)$.

Es handelt sich um einen (dualen Potenzmengen)verband.

(ACC) gilt, da der Verband endlich ist.

Anfangswert i:
$\emptyset$.

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

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

//Teilausdrücke, die x enthalten und daher von Block b geändert werden.

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

//Teilausdrücke, die von Block b genutzt werden.

Die Transferfunktionen sind monoton.

Betrachte das Beispielprogramm c an der Tafel. (*Fehlt hier leider.*

Die Transferfunktionen sind

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

Das vom Datenflusssystem induzierte Gleichungssystem ist

$$X_1 = \underbrace{(X_2 \cup \{b - a\})}_{=f_2(X_2)} \cap \underbrace{(X_4 \cup \{b - a\})}_{=f_4(X_4)}$$
$$X_2 = X_3 \cup \{a - b\}$$
$$X_3 = \underbrace{\emptyset}_{=i}$$
$$X_4 = X_5 \cup \{a - b\}$$
$$X_5 = \underbrace{\emptyset}_{=i}$$

$$X_1 = (X_2 \cup \{b - a\}) \cap (X_4 \cup \{b - a\})$$
$$X_2 = X_3 \cup \{a - b\}$$
$$X_3 = \emptyset$$
$$X_4 = X_5 \cup \{a - b\}$$
$$X_5 = \emptyset$$

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	$d_5)$
$g_S^0(\bot)$	$(\{a-b, b-a\}$	$\{a-b, b-a\}$	$\{a-b, b-a\}$	$\{a-b, b-a\}$	$\{a-b, b-a\})$
$g_S^1(\bot)$	$(\{a-b, b-a\}$	$\{a-b, b-a\}$	$\emptyset$	$\{a-b, b-a\}$	$\emptyset)$
$g_S^2(\bot)$	$(\{a-b, b-a\}$	$\{a-b\}$	$\emptyset$	$\{a-b\}$	$\emptyset)$
$g_S^3(\bot)$	$(\{a-b, b-a\}$	$\{a-b\}$	$\emptyset$	$\{a-b\}$	$\emptyset)$

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

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

Die kleinste Lösung des Gleichungssystems ist

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

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.

Distributive Frameworks

Eine Funktion f auf einem endlichen 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.)

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.

2.2.8 Satz

Bitvektor-Frameworks sind distributive Frameworks.

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.

2.2.9 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 Worklist, initial $W = \varepsilon$
for all $(b, b') \in F$ **do** $W := W.(b, b')$ **endfor**
for all $b \in E$ **do** $X_b := i$ **endfor**
while $W \neq \varepsilon$ **do**
  pop (b, b') **from** W;
  if $f_b(X_b) \not\preceq X_{b'}$ **then**
    $X_{b'} := X_{b'} \sqcup f_b(X_b)$;
    for all $(b', b'') \in F$ **do**
      if $(b', b'') \notin W$ **then** $W := W.(b', b'')$ **endif**
    endfor
  endif
endwhile

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

2.2.10 Satz
Sei das Datenflusssystem S die Eingabe für obigen Algorithmus. Der Algorithmus terminiert und berechnet $lfp(g_S)$.

2.2.11 Beispiel
Available-Expressions-Analyse am Beispielprogramm mittels Worklist:

Nach Initialisierung:

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

$$\begin{aligned} X_1 &= \emptyset \\ X_2 &= AExp \\ X_3 &= AExp \\ X_4 &= AExp \\ X_5 &= AExp \end{aligned}$$

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

Nach Iteration 1:

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

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

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

Nach Iteration 2:

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

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

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

Nach Iteration 3:

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

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

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

Nach Iteration 4:

$$W = (5,3)$$

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

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:

$$W = (3,4)$$

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

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:

$$W = (4, 5)$$

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

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

Damit terminiert der Algorithmus.

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.

2.3.1 Definition

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

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

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

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

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

mit

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

2.3.2 Beispiel (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:

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

Beachte:

- 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.

2.3.3 Satz (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}.

2.3.4 Satz (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 $X_b^{JOP,n} = \sqcup\emptyset = \bot \leq X_b^{LFP}$.

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

Ziel: Datenflussanalyse für rekursive Programmen

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

Idee: Berechne *JOVP-Lösung* (join-over-all-valid-paths)

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

3.1.1 Definition (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. (Damit lassen sich Parameter und return-Werte nachbilden.)
- 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:

- Für jede 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.*

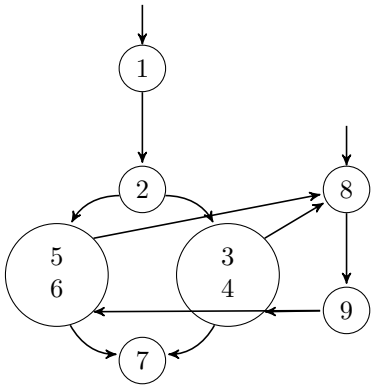
Dabei ist der *interprozeduale Flow* IF definiert als

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

3.1.2 Beispiel

Warum brauchen wir die 4-Tupel?

```
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:

3.1.3 Definition (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

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

3.1.4 Definition (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

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:

- 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

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

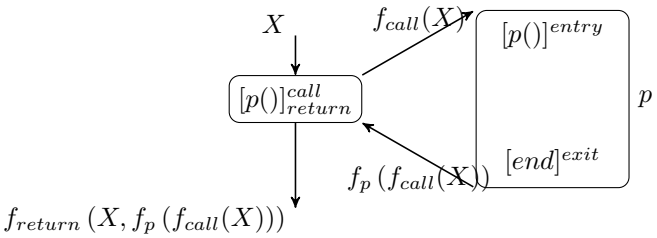
  darstellen. Sowohl f_{call} als auch f_{return} sind als Teil des Datenflusssystems gegeben.

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

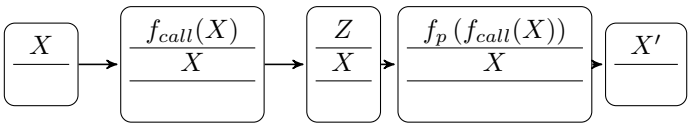
  - Initialisiere Datenflusswert bei Eintritt in die Prozedur
  - abhängig vom aktuellen Datenflusswert

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

  - 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 $D \to D$

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

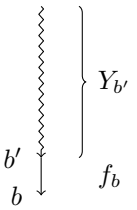
3.2.1 Definition

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

Dann induziert S das *Summary-Gleichungssystem*:

Eine Variable Y_b pro Block und eine Variable Z_p pro Prozedur.

Idee: Y_b fasst den Effekt der Prozedur, in der b vorkommt von *entry* bis b zusammen.



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

Dabei ist

$$f_{b'} = \text{callret}(Z_q) : D \to D, \text{ falls } b' = [q()]^{call}_{ret}$$
$$f_{b'} = \text{ wie in f angegeben, sonst}$$

Dabei ist

$$\begin{array}{lrcl} \text{callret}(Z_q): & D & \to & D \\ \text{callret}(Z_q) & (d) & = & f_{return}(d, Z_q(f_{call}(d))) \\ & Z_p & = & f_{exit} \circ Y_{exit} \end{array}$$

3.2.2 Satz

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) $Y_p \geq f_\pi$ 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

3.2.3 Beispiel (Reliable-Values-Analyes)

Gegeben das Program

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

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

Erste Modellierungsmöglichkeit:

Das Gleichungssystem hat für einen Block eine Variable Y_b, die wir mit dem call-Label identizifieren.

Im Beispiel: $Y_b = Y_3$ und $Y_5 = f_3 \circ Y_3 = callret(Z_q) \circ Y_3$, mit $f_3 = callret(Z_q)$.

Alternativ (und äquivalent):

Zwei Variablen werden zu Blöcken, die Funktionsaufrufe darstellen, assoziiert.

Y_3 erhält die Transferfunktion

$$f_3 = Z_q \circ f_call.$$

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 er"halt man die folgende Gleichung für Block 5:

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

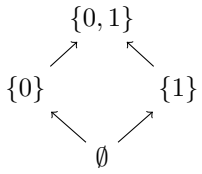
und damit das selbe Ergebnis wie im anderen Fall.

3.2.4 **Beispiel** (Reachable-Values-Analyse)

Gegeben: Eine Boolesche Variable x

Berechne: Transferverhalten der Prozedur $pos()$ auf der Wertemenge für x.

Vollständiger Verband der Wertemengen:




```
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
[end]^8;
```

Wir verwenden die Transferfunktionen

$$\begin{aligned} f_{get1} = f_2 = f_{call(pos)} = f_8 &= id \\ f_3 &= f_{get0} \\ f_7 &= f_{get1} \\ f_4 &= f_{invert} \\ f_{return(pos)}(d_1, d_2) &= d_2 \end{aligned}$$

Dabei ist f_{geti} eine Funktion, die ausdrückt, dass wir ein $assert(x = i)$ nur dann nehmen können, wenn der Wert $x = i$ auch tatsächlich möglich ist. Die Funktion f_{invert} kehrt den Wert um.

Wir definieren außerdem die Hilfsfunktionen $f_{make1}, f_{invert0}$ und $f_{\perp}$.

Die Werte all dieser Funktionen sind durch die folgende Tabelle gegeben

X	$id(X)$	$f_{\perp}(X)$	$f_{get0}(X)$	$f_{get1}(X)$	$f_{invert}(X)$	$f_{make1}(X)$	$f_{invert0}(X)$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{0\}$	$\{0\}$	$\emptyset$	$\{0\}$	$\emptyset$	$\{1\}$	$\{1\}$	$\{1\}$
$\{1\}$	$\{1\}$	$\emptyset$	$\emptyset$	$\{1\}$	$\{0\}$	$\{1\}$	$\emptyset$
$\{0,1\}$	$\{0,1\}$	$\emptyset$	$\{0\}$	$\{1\}$	$\{0,1\}$	$\{1\}$	$\{1\}$

Beachte, dass die folgenden Beziehungen gelten:

$$\begin{aligned} f_{invert} &\circ f_{get0} &&= f_{invert0} \\ f_{get1} &\circ f_{invert0} &&= f_{invert0} \\ f_{make1} &\circ f_{invert0} &&= f_{invert0} \\ f_{invert0} &\sqcup f_{get1} &&= f_{make1} \end{aligned}$$

Summary-Gleichungssystem:

$$\begin{aligned} Y_2 &= id \\ Y_3 &= id \circ Y_2 \\ Y_4 &= f_{get0} \circ Y_3 \\ Y_5 &= f_{inv} \circ Y_4 \\ Y_6 &= Y_{pos} \circ Y_5 \\ Y_7 &= id \circ Y_2 \\ Y_8 &= id \circ Y_6 \sqcup f_{get1} \circ Y_7 \\ Y_{pos} &= id \circ Y_8 \end{aligned}$$

Löse das Gleichungssystem durch Fixpunktiteration:

[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] $f_{get1} \circ f_{invert0} \sqcup f_{get1} \circ id = f_{invert0} \sqcup f_{get1} = f_{make1}$
[4] $f_{make1} \circ f_{invert0} \sqcup f_{get1} = f_{invert0} \sqcup f_{get1} = f_{make1}$

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	id	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$	$f_\bot$
2	id	$f_\bot$	$f_\bot$	$f_\bot$	id	$f_\bot$	$f_\bot$	$f_\bot$
3	id	id	f_{get0}	$f_\bot$	$f_\bot$	id	f_{get1}	$f_\bot$
4	id	id	f_{get0}	$f_{invert0}$	$f_\bot$	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_{make1}	$f_{get1}^{\ 2}$
7	id	id	f_{get0}	$f_{invert0}$	$f_{invert0}$	id	f_{make1}	$f_{make1}^{\ 3}$
8	id	id	f_{get0}	$f_{invert0}$	$f_{invert0}$	id	f_{make1}	$f_{make1}^{\ 4}$ $\surd$

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

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

Dabei ist

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

3.2.5 Satz (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 $X_b^{JOVP} = X_b^{LFP}$ $\forall b \in B$

Korollar

Das *Control-State-Reachability-Problem* ist entscheidbar.

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

Frage: Gibt es einen Ablauf, der zu b führt?

Beweis:
Die Transferfunktionen, die für die Reachable-Values-Analyse genutzt werden, sind distributiv. Damit ist Fixpunktlösung auch die JOVP-Lösung. Ein Block b ist erreichbar, wenn in der JOVP-Lösung $Y_b \neq \emptyset$ gilt. □

3.3 Der Call-String-Ansatz

- Der funktionale Ansatz ist kontextinsenstitiv, d.h. er berücksichtigt *keine* Information über den Kontext, in dem eine Prozedur aufgerufen wird.
- z.B.: 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

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

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

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

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

Dann ist das induzierte *Call-String-Gleichungssystem*

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

3.3.2 Satz
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 $\Gamma^{\leq n} := \cup_{i=0}^{n} \Gamma^i$
- 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.

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 $\mathbb{Z}$
- Berücksichtigt alle konkreten Eingaben
- *Ersetze* konkrete Operationen durch *abstrakte Operationen*. Dabei müssen alle konkreten Werte berücksichtigt werden, die durch den abstrakten Wert dargestellt sind.

4.0.3 Beispiel

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

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 Präzision 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, für die er steht.

4.1.1 Definition (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

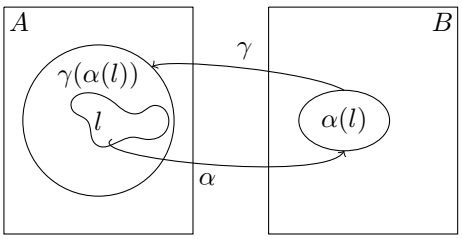
$$\begin{array}{lllll} \text{(G1)} & \forall l \in L: & l & \leq_L & \gamma(\alpha(l)) \\ \text{(G2)} & \forall m \in M: & \alpha(\gamma(m)) & \leq_M & m \end{array}$$

Notation: $L \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} M$

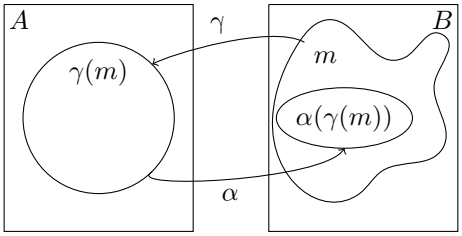
Typisch:

- $l \neq \gamma(\alpha(l))$, 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*)

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



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

4.1.2 Satz (Eigenschaften von Galois-Verbindungen)
Sei $L \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} M$ eine Galois-Verbindung ($\alpha : L \to M, \gamma : M \to L$).

Seien ferner $l \in L, m \in M$.

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

(2a) Die Konkretisierung γ ist *eindeutig* durch α bestimmt:
$$\gamma(m) = \sqcup\{l \in L \mid \alpha(l) \leq_M m\}.$$

(2b) α ist eindeutig durch γ bestimmt:
$$\alpha(l) = \sqcap\{m \in M \mid l \leq_L \gamma(m)\}.$$

(3a) Sei $L' \subseteq L$. α ist *vollständig distributiv*, d.h.
$$\alpha(\sqcup L') = \sqcup\alpha(L').$$
(Diese Eigenschaft wird auch *vollständig additiv* genannt.)

(3b) Sei $M' \subseteq M$. γ ist *vollständig multiplikativ*, d.h.
$$\gamma(\sqcap M') = \sqcap\gamma(M').$$

(4a) Zu jeder vollständig distributiven Funktion $\alpha' : L \to M$ gibt es ein $\gamma' : M \to L$ (wie in (2a)), so dass $L \underset{\gamma'}{\overset{\alpha'}{\leftrightarrows}} M$ eine Galois-Verbindung ist.

(4b) Zu jeder vollständig distributiven Funktion $\gamma' : M \to L$ gibt es ein $\alpha' : L \to M$ (wie in (2b)), so dass $L \underset{\gamma'}{\overset{\alpha'}{\leftrightarrows}} M$ eine Galois-Verbindung ist.

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

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

Zeige dazu $\leq_L$ und ${}_L \geq$.

zu $\leq_L$:

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 ${}_L \geq$:

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) 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 (Übungsaufgabe!)

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

Damit gilt dann

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

(3b) analog

(4) Übungsaugabe

□

4.2 Konstruktion von Galois-Verbindungen

4.2.1 Zwei konkrete Galois-Verbindungen

Intervallabstraktion

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

Sei

$$M = ((\mathbb{Z} \cup \{-\infty\} \times (\mathbb{Z} \cup \{+\infty\}), \subseteq)$$

die abstrakte Domäme der Intervalle. (Dabei kann das leere Intervall $\emptyset$ durch Intervalle der Form $[u, o]$ mit $o < u$ dargestellt werden, z.B. durch $[1, -1]$.)

Wir definieren eine Galois-Verbinung $L \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} M$ durch
$\alpha : L \to M$ mit

$$\alpha(Z) := \begin{cases} \emptyset, \text{ falls } Z = \emptyset \\ [\min Z, \max Z], \text{ sonst} \end{cases}$$

und $\gamma : M \to L$ mit

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

Zum Beispiel gilt

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

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

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

verwendet, d.h. 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.

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

Definiere $\alpha : L \to M$ durch

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

und $\gamma : M \to L$ mittels

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

(Beachte, $-3 \mod 5 = 2$)

4.2.2 Galois-Verbindung aus Extraktionsfunktionen

Sei $\beta : V \to D$ eine Funktion.

Dann ist $\mathcal{P}(V) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \mathcal{P}(D)$ mit

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

eine *Galois-Verbindung* mit

$$\begin{array}{lll} \alpha(V') & := \beta(V) & = \{\beta(v) \mid v \in V'\} \\ \gamma(D') & := \beta^{-1}(D') & = \{v \in V \mid \beta(v) \in D'\} \end{array}$$

4.2.2 Beispiel
Die oben definierte Kongruenzabstraktion ergibt sich aus der Extraktionsfunktion

$$\begin{array}{rccl} \mod k : & \mathbb{Z} & \to & \{0, ..., k-1\} \\ & z & \mapsto & z \mod k \end{array}$$

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

Beachte dass sich eine Belegung boolscher Variablen wahlweise als Vektor in $\mathbb{B}^{\text{Vars}}$ oder als Funktion Vars $\to \mathbb{B}$ auffassen lässt.

4.2.3 Definition (Liften von Extraktionsfunktionen)
Sei $State = \mathbb{B}^{\text{Vars}}$ die Menge der Variablenbelegungen σ und sei $\beta : \mathbb{B} \to D$

Durch *Liften von β auf State* entsteht die Abstraktion

$$\begin{array}{rccl} \alpha: & \mathcal{P}(State) & \to & \mathcal{P}\left(D^{\text{Vars}}\right) \\ & \Sigma \subseteq State & \mapsto & \alpha(\Sigma) := \{\beta \circ \sigma \mid \sigma \in \Sigma\} \end{array}$$

Es lässt sich zeigen, dass α vollständig additiv und damit eine Galois-Verbinung definiert (siehe Satz oben).

4.2.3 Komposition von Galois-Verbindungen

Sequentielle Komposition:

Führe Abstraktionsfunktionen hintereinander aus

Seien $L_1 \underset{\gamma_1}{\overset{\alpha_1}{\leftrightarrows}} L_2$ und $L_2 \underset{\gamma_2}{\overset{\alpha_2}{\leftrightarrows}} L_3$ Galois-Verbindungen,
(d.h. $\alpha_1 : L_1 \to L_2$, $\gamma_1 : L_2 \to L_1$ $\alpha_2 : L_2 \to L_3$, $\gamma_2 : L_3 \to L_2$).

Dann ist $L_1 \underset{\gamma_1 \circ \gamma_2}{\overset{\alpha_2 \circ \alpha_1}{\longleftrightarrows}} L_3$ eine Galois-Verbindung zwischen $(L_1, \leq_1)$ und $(L_3, \leq_3)$

Parallelkomposition von Galois-Verbindungen:

Unabhängiges (direktes) Produkt / Unabhängige Attribute:
Seien $L \underset{\gamma_1}{\overset{\alpha_1}{\leftrightarrows}} M_1$ und $L \underset{\gamma_2}{\overset{\alpha_2}{\leftrightarrows}} M_2$ Galois-Verbindungen.

Dann ist auch das *direkte Produk*

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

mit

$$\begin{array}{rccl} \alpha: & L & \to & M_1 \times M_2 \\ & l & \mapsto & \alpha(l) := (\alpha_1(l), \alpha_2(l)) \end{array}$$

$$\begin{array}{rccl} \gamma: & M_1 \times M_2 & \to & L \\ & (m_1, m_2) & \mapsto & \gamma(m_1, m_2) := \gamma_1(m_1) \sqcap \gamma_2(m_2) \end{array}$$

eine Galois-Verbindung $L \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} M_1 \times M_2$. Sie führt die beiden gegebenen Galois-Verbinungen unabhängig voneinander aus.

Vorteil: Man kann schnell damit rechnen.
Nachteil: Unpräzise.

Abhängiges (Tensor-)Produkt / Relationale Methode: Wir möchten nun eine Galois-Verbingung definieren, die das Zusammenspiel der Abstraktionsfunktionen berücksichtigt. Dazu wenden wir die Abstraktionen auf die Mengen der Form $\{v\}$ an und vereinigen die Ergebnisse.

Seien $\mathcal{P}(V) \underset{\gamma_1}{\overset{\alpha_1}{\rightleftarrows}} \mathcal{P}(D_1)$ und $\mathcal{P}(V) \underset{\gamma_2}{\overset{\alpha_2}{\rightleftarrows}} \mathcal{P}(D_2)$ Galois-Verbindungen.

Dann ist auch das *Tensorprodukt*

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

mit

$$\begin{array}{rrcl} \alpha: & \mathcal{P}(V) & \to & \mathcal{P}(D_1 \times D_2) \\ & V' \subseteq V & \mapsto & \alpha(V') := \bigcup\limits_{v \in V'} \alpha_1(\{v\}) \times \alpha_2(\{v\}) \end{array}$$

$$\begin{array}{rrcl} \gamma: & \mathcal{P}(D_1 \times D_2) & \to & \mathcal{P}(V) \\ & D' \subseteq D_1 \times D_2 & \mapsto & \gamma(D') := \{v \in V \mid \alpha_1(\{v\}) \times \alpha_2(\{v\}) \subseteq D'\} \end{array}$$

eine Galois-Verbindung $\mathcal{P}(V) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \mathcal{P}(D_1 \times D_2)$.

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

$$\begin{array}{rrcl} sign: & \mathbb{Z} & \to & \{-, 0, +\} \\ & x & \mapsto & sign(x) = \begin{cases} +, & x > 0, \\ 0, & x = 0, \\ -, & x < 0, \end{cases} \end{array}$$

$$\begin{array}{rrcl} parity: & \mathbb{Z} & \to & \{e, o\} = \{even, odd\} \\ & x & \mapsto & parity(x) = \begin{cases} e, & x = 0 \mod 2 \\ o, & x = 1 \mod 2 \end{cases} \end{array}$$

Diese Funktionen induzieren Galois-Verbindungen

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

Wir erhalten die neuen Galois-Verbindungen

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

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

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

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

Komponentenweise Kombinationen:

Man kann die Konstruktionen auch analog für zwei Galois-Verbinungen, die unterschiedliche Definitionsbereiche haben, durchführen.

Unabhängige Attribute / Direkte Produkt:
Seien $L_1 \underset{\gamma_1}{\overset{\alpha_1}{\rightleftarrows}} M_1$ und $L_2 \underset{\gamma_2}{\overset{\alpha_2}{\rightleftarrows}} M_2$ Galois-Verbindungen.

Dann ist auch das *direkte Produk*

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

mit

$$
\begin{array}{rrcl}
\alpha: & L_1 \times L_2 & \to & M_1 \times M_2 \\
& (l_1, l_2) & \mapsto & \alpha(l_1, l_2) := (\alpha_1(l_1), \alpha_2(l_2))
\end{array}
$$

$$
\begin{array}{rrcl}
\gamma: & M_1 \times M_2 & \to & L_1 \times l_2 \\
& (m_1, m_2) & \mapsto & \gamma(m_1, m_2) := (\gamma_1(m_1), \gamma_2(m_2))
\end{array}
$$

eine Galois-Verbindung $L_1 \times L_2 \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} M_1 \times M_2$. Sie führt die beiden gegebenen Galois-Verbinungen unabhängig voneinander aus.

Relationale Methode / Tensor-Produkt: Seien $\mathcal{P}(V_1) \underset{\gamma_1}{\overset{\alpha_1}{\rightleftarrows}} \mathcal{P}(D_1)$ und $\mathcal{P}(V_2) \underset{\gamma_2}{\overset{\alpha_2}{\rightleftarrows}} \mathcal{P}(D_2)$ Galois-Verbindungen.

Dann ist auch das *Tensorprodukt*

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

mit

$$
\begin{array}{rrcl}
\alpha: & \mathcal{P}(V_1 \times V_2) & \to & \mathcal{P}(D_1 \times D_2) \\
& V' \subseteq V_1 \times V_2 & \mapsto & \alpha(V') := \bigcup\limits_{(v_1, v_2) \in V'} \alpha_1(\{v_1\}) \times \alpha_2(\{v_2\})
\end{array}
$$

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

eine Galois-Verbindung $\mathcal{P}(V_1 \times V_2) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \mathcal{P}(D_1 \times D_2)$.

4.3 Konkrete (strukturierte operationelle) Semantik von while-Programmen

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

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

Idee von SoS:

- *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.

  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):

a ::= k | x | $a_1 + a_2$ | $a_1 - a_2$ | $a_1 * a_2$
b ::= t | $a_1 = a_2$ | $a_1 < a_2$ | $\neg b_1$ | $b_1 \vee b_2$ | $b_1 \wedge b_2$
c ::= skip | x := a | $c_1; c_2$
| if b then c_1 else c_2 end
| while b do c_1 end

Dabei sind `a` arithmetische Ausdrücke und repräsentieren Werte in $\mathbb{Z}$ (insbesondere: Konstanten `k` sind in $\mathbb{Z}$ und die Variablen `x` repräsentierten Werte in $\mathbb{Z}$), `b` boolsche Ausdrücke (insbesondere: Konstanten `t` sind `true` oder `false`). Sei $Prog$ die Menge der Programme, gegeben durch die EBNF für `c`.

Seien $Sig = (Funk, \text{Präd})$ die Signatur des Programmes:

$Funk$ = genutze Funktionssymbole = $\{+/_2, -/_2, */_2\} \cup \{k/0 \mid k \in \mathbb{Z}\}$

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:

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

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

  Schreibe auch $f_\mathcal{I}$ bzw. $p_\mathcal{I}$ statt $\mathcal{I}(f)$ bzw. $\mathcal{I}(p)$.

  Hier: $D = \mathbb{Z}, \mathcal{I}$ natürliche Interpretation der Operatoren und Konstanten.

- Das Verhalten eines Programmes in einem Zustand hängt von *der Belegung der Variablen ab*, also einem $\sigma \in$ State, wobei State $= \mathbb{Z}^{\text{Vars}}$, d.h.

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

- 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)

4.3.1 Definition (Transitionsrelation zwischen Konfigurationen)
Eine Konfiguration ist ein Paar $(c, \sigma) \in \mathcal{P}rog \times$ State. Dabei ist c das noch auszuführende Programm und σ die aktuelle Variablenbelegung.

Die Transitionsrelation zwischen Konfigurationen

$$\to \quad \subseteq (\times State) \times (\underbrace{(\mathcal{P}rog \times State)}_{\text{Programm terminiert noch nicht}} \cup \underbrace{State}_{\text{Programm hat terminiert}})$$

ist die kleinste Relation, die folgenden Regeln genügt:

$$(skip)\quad \frac{}{(skip,\sigma) \;\to\; \sigma}$$

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

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

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

$$(iftrue)\quad \frac{}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end},\sigma) \;\to\; (c_1,\sigma)} \text{, falls } \mathcal{S}[\![b]\!]\sigma = true$$

$$(iffalse)\quad \frac{}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end},\sigma) \;\to\; (c_2,\sigma)} \text{, falls } \mathcal{S}[\![b]\!]\sigma = false$$

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

$$(whilefalse)\quad \frac{}{(\texttt{while } b \texttt{ do } c \texttt{ end },\sigma) \;\to\; \sigma} \text{, falls } \mathcal{S}[\![b]\!]\sigma = false$$

4.3.2 Beobachtung

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

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

Der Zustandsraum State $= \mathbb{Z}^{\text{Vars}}$ ist kein vollständiger Verband. Um Galois-Verbindungen zur Abstraktion von Zustandsmengen nutzen zu können, definieren wir die Funktionen

$$\begin{array}{rrcl} post_{c,c'}: & \mathcal{P}\,(\text{State}) & \to & \mathcal{P}\,(\text{State}) \\ post_c: & \mathcal{P}\,(\text{State}) & \to & \mathcal{P}\,(\text{State}) \end{array}$$

mit

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

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

4.4.1 Definition (Sichere Approximation von Funktionen)
Sei $L \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} M$ Galois-Verbindung.

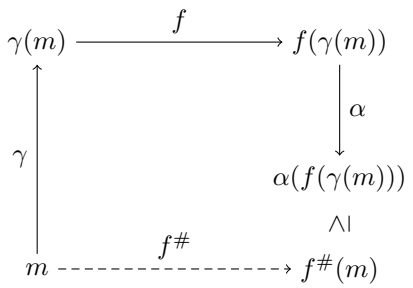
Sei ferner $f : L \to L$ eine Funktion.

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

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

  d.h $\alpha(f(\gamma(m))) \leq_M f^\#(m)$ 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.

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

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

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

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

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

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

$$f^{\#}_{-2}: \quad \mathcal{P}(\{-,0,+\}) \quad \to \quad \mathcal{P}(\{-,0,+\})$$

$$f^{\#}_{-2}(A): \quad := \quad \begin{cases} \emptyset & , A = \emptyset, \\ \{-,0,+\} & , + \in A, \\ \{-\} & , \text{ sonst.} \end{cases}$$

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

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

Zum Beispiel:

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

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

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

Bedingung kann wahr oder falsch sein.

4.4.4 **Definition** (Abstrakte Semantik)

Betrachte die Galois-Verbindung $\mathcal{P}(State) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} M$

- 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 (d.h. in der obigen Ungleichung gilt immer Gleichheit), nenne dies genaueste abstrakte Semantik
- Die abstrakte Semantik induziert die *abstrakte Transitionsrelation*:

  $$\Rightarrow \subseteq (\mathcal{P}rog \times M) \times (\mathcal{P}rog \times M) \cup M)$$

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

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

4.4.5 Beispiel (genaueste abstrakte Semantik)

Betrachte $\mathcal{P}(() \mathbb{Z}) \underset{\gamma_{parity}}{\overset{\alpha_{parity}}{\rightleftarrows}} \mathcal{P}(() \{n = odd, n = even\})$. Es gilt:

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

4.4.6 Lemma

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

Ungenauere abstrakte Semantiken lassen sich immer herleiten.

Warum?

- Betrachte die Galois-Verbindung $\mathcal{P}(State) \underset{\gamma_{sign}}{\overset{\alpha_{sign}}{\rightleftarrows}} \mathcal{P}(\{-, 0, +\})$
- Sei die abstrakte Konfiguration: (if $n > 2 \vee x^n + y^n = z^n$ then $n := 1$ else $n := -1$ fi , $\{(n = +, x = +, y = +, z = +)\}$)
- 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 *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)

4.5 Herleitung einer abstrakten Semantik

Ziel:

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

Problem:

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

Lösung:

Werte Approximation in *3-wertiger Logik* aus:

$$(\mathcal{P}(\mathbb{B}) \setminus \{\emptyset\}, \wedge_3, \vee_3, \neg_3)$$

Dabei identifizieren wir

$$\begin{array}{rcll} \{0\} & = & 0 = false & \text{(definitiv 0)} \\ \{1\} & = & 1 = true & \text{(definitiv 1)} \\ \{0,1\} & = & 1/2 & \text{(wir kennen den genauen Wert nicht)} \end{array}$$

Die logischen Operatoren sind wie folgt definiert:

$\wedge_3$	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

$\neg_3$	
0	1
1	0
1/2	1/2

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

$$p : \mathcal{P}(\mathbb{Z})^n \to \mathcal{P}(\mathbb{B}) \setminus$$

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

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

4.5.2 Definition
Sei $\mathcal{S} = (\mathbb{Z}, I)$ und $(\alpha_\beta, \gamma_\beta)$ die Galois-Verbindung $\mathcal{P}(\mathbb{Z}) \underset{\gamma_\beta}{\overset{\alpha_\beta}{\leftrightarrows}} \mathcal{P}(D)$, die als Lift von $\beta : \mathbb{Z} \to D$ definiert ist.

Dann heißt eine Struktur $\mathcal{S}_{Abs}(\mathcal{P}(D), I^\#)$ *abstrakte Sig-Struktur* falls

- $f_\mathcal{I}^\# : \mathcal{P}(D)^n \to \mathcal{P}(D)$ ist sichere Approximation von $f_\mathcal{I} : \mathbb{Z}^n \to \mathbb{Z}$ für alle Funktionssymbole f und
- $p_\mathcal{I}^\# : \mathcal{P}(D)^n \to \mathcal{P}(\mathbb{B})$ ist sichere Approximation von $p_\mathcal{I} : \mathbb{Z}^n \to \mathbb{B}$ für alle Prädikate p.

Die Semantik Bool'scher Ausdrücke in 3-wertiger Logik ist dabei wie folgt gegeben. Es sei $\sigma : \text{Vars} \to \mathcal{P}(D)$.

$$
\begin{array}{rcl}
\mathcal{S}_{Abs}[\![p(a_1,\ldots,a_n)]\!](\sigma) & := & p_{\mathcal{I}}^{\#}(\mathcal{S}_{Abs}[\![a_1]\!](\sigma),\ldots,\mathcal{S}_{Abs}[\![a_n]\!](\sigma)) \\
\mathcal{S}_{Abs}[\![b_1 \vee b_2]\!](\sigma) & := & \mathcal{S}_{Abs}[\![b_1]\!](\sigma) \vee_3 \mathcal{S}_{Abs}[\![b_2]\!](\sigma) \\
\mathcal{S}_{Abs}[\![b_1 \wedge b_2]\!](\sigma) & := & \mathcal{S}_{Abs}[\![b_1]\!](\sigma) \wedge_3 \mathcal{S}_{Abs}[\![b_2]\!](\sigma) \\
\mathcal{S}_{Abs}[\![\neg b]\!](\sigma) & := & \neg_3 \mathcal{S}_{Abs}[\![b]\!](\sigma)
\end{array}
$$

Eine mögliche Definition für $f_{\mathcal{I}}^{\#}$ ist dabei:

$$f_{\mathcal{I}}^{\#}(D_1,\ldots D_n) = \alpha_\beta\left(f_{\mathcal{I}}\left(\gamma_\beta(D_1),\ldots,\gamma_\beta(D_n)\right)\right)$$

4.5.3 Lemma

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

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

Ist eine abstrakte Sig-Struktur gegeben, *erhält* man das abstrakte Transitionssystem

$$\Rightarrow \quad \subseteq \left(\mathcal{P}rog \times \mathcal{P}\left(D^{\mathrm{Vars}}\right)\right) \times \left(\left(\mathcal{P}rog \times \mathcal{P}\left(D^{\mathrm{Vars}}\right)\right) \cup \mathcal{P}\left(D^{\mathrm{Vars}}\right)\right)$$

mittels folgender Regeln:

$$\frac{}{(skip, Abs) \Rightarrow Abs}$$

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

$$\frac{(c_1, Abs) \Rightarrow Abs;}{(c_1; c_2, Abs) \Rightarrow (c_2, Abs')}$$

$$\frac{(c_1, Abs) \Rightarrow (c_1', Abs')}{(c_1; c_2, Abs) \Rightarrow (c_1'; c_2, Abs')}$$

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

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

$$\frac{}{(\texttt{while } b \texttt{ do } c \texttt{ end }, Abs) \Rightarrow (c; \texttt{while } b \texttt{ do } c \texttt{ end }, Abs \backslash \{\tau \mid \mathcal{S}_{Abs}[\![b]\!](\tau') = \{0\}\})}$$

$$\frac{}{(\texttt{while } b \texttt{ do } c \texttt{ end }, Abs) \Rightarrow Abs \backslash \{\tau \mid \mathcal{S}_{Abs}[\![b]\!](\tau') = \{1\}\}}$$

Beachte, dass $\mathcal{S}_{Abs}[\![b]\!]$ eine Belegung in $\mathcal{P}(D)^{\mathrm{Vars}}$, d.h. vom Typ $\mathrm{Vars} \to \mathcal{P}(D)$ erwartet, wir jedoch $\tau \in Abs \subseteq \mathcal{P}\left(D^{\mathrm{Vars}}\right)$ gegeben haben, d.h. $\tau : Vars \to D$. Wir konstruieren daher $\tau' : \mathrm{Vars} \to \mathcal{P}(D)$ via

$$\tau'(x) := \{\tau(x)\}.$$

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

4.5.4 Definition

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

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

4.5.5 Satz

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:

- Entweder: Programm verletzt die Eigenschaft wirklich.
- Oder: Abstraktion ist bloß zu grob

Ziel:

Entwicklung eines abstraktionsbasierten Programmanalyseverfahrens,

- das die Verletzung der Eigenschaft aufzeigt,
- oder die Abstraktion selbstständig verfeinert.

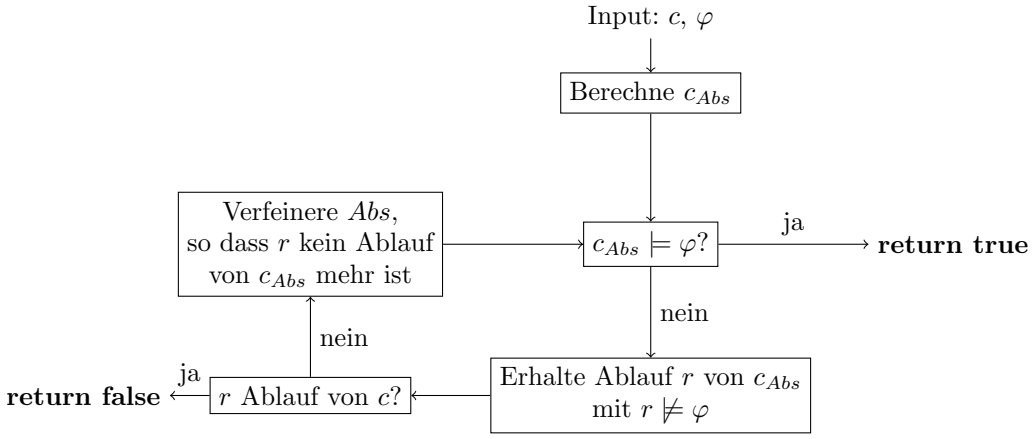
CEGAR

Counter example-guided abstraction refinement
Gegeben ein Programm c und eine Eigenschaft φ.

1. Teste sichere Abstraktion c_{Abs} von c auf φ
2. Falls φ in c_{Abs} erfüllt ist, ist φ auch in c erfüllt (**return true**).
3. Falls φ nicht erfüllt ist, liefert das Verfahren ein Gegenbeispiel, also einen Ablauf a in der abstrakten Semantik, der φ verletzt
4. Falls dieser Ablauf auch in der konkreten Semantik ausführbar ist, ist φ nicht erfüllt (**return false**)
5. Falls dieser Ablauf in der konkreten Semantik nicht ausführbar ist, verfeinere die Abstraktion so, dass er auch in der abstrakten Semantik nicht mehr ausführbar ist, und gehe zurück zu 1.

Dieses Verfahren ist ein Semi-Entscheidungsverfahren, d.h. es terminiert eventuell nicht.

Illustration:



Idee: Prädikatenabstraktion

- Nutze Prädikte wie in First-Order-Logik $p_1, \ldots, p_n \in FO$ um Zustände $\sigma : \text{Vars} \to \mathbb{Z}$ auf Bitvektoren $(0/1, \ldots, 0/1) \in \{0,1\}^n$ zu abstrahieren (direktes Produkt)
- *Verfeinerung* geschieht durch Hinzufügen von Prädikten
- *Wahl der Prädikate:*
  - Initial aus dem Programm abgeleitet (Bedingungen, in *if* und *while* verwendet werden)
  - 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 $p_1, \ldots, p_n$
- Verwendung beliebiger Boolescher Kombinationen $(\vee, \wedge, \neg)$ skaliert nicht.
- Nutze *kartesische Abstraktion*, d.h. nur Konjunktionen $(\wedge)$

5.1.1 Definition

- Ein *Prädikat* ist ein Boolescher Ausdruck b // Siehe: Syntax von Programmen
- Ein Zustand $\sigma : \text{Vars} \to \mathbb{Z}$ *erfüllt* b, falls $\mathcal{S}[\![b]\!](\sigma) = true$. Schreibe: $\sigma \vDash b$.
- Pädikat q *ist schwächer als* p falls $\forall \sigma \in State : \sigma \vDash p$ impliziert $\sigma \vDash q$

  Wir schreiben $p \vDash q$ und sagen auch: p *ist stärker als* q.
- Prädikate p und q heißen *(logisch) äquivalent*, falls $p \vDash q$ und $q \vDash p$.
  Schreibe: $p \models\!\!\models q$.
- $\models$ definiert eine Quasiordnung auf der Menge der Booleschen Ausdrücke. Problem: Diese Ordnung ist reflexiv und transitiv, aber nicht antisymmetrisch, da es logisch äquivalente Formeln gibt. Wir lösen dieses Problem, in dem wir logisch äquivalente Formeln mit einander identifizieren.
- 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

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

  (Modulo logischer Äquivalenz.)

  Schreibe $true := \wedge\emptyset, false := \wedge\{p_i, \neg p_i \ldots\}$.

  Elemente in $Abs(P)$ heißen *Cubes*

5.1.2 Lemma

$Abs(P)$ ist vollständiger Verband.

Beweis:

- $\bot = false, \top = true$
- $q_1 \sqcap q_2 = q_1 \wedge q_2$

- $q_1 \sqcup q_2 = \overline{q_1 \vee q_2}$
  Dabei ist zu einer Formel b (die eventuell nicht in $Abs(P)$ ist) $\overline{b}$ die stärkste Formel in $Abs(P)$, die aus b folgt.

  Sie ist eindeutig bestimmt und wir können sie wie folgt berechnen:

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

□

5.1.3 Beispiel

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

1. Für $q_1 = p_1 \wedge p_2$ sowie $q_2 = p_1 \wedge \neg p_2$

$$\begin{array}{lcl} q_1 \sqcap q_2 & = & p_1 \wedge p_2 \wedge p_1 \wedge \neg p_2 \\ & \models\models & false \end{array}$$

$$\begin{array}{lcl} q_1 \sqcup q_2 & = & \overline{(p_1 \wedge p_2) \vee (p_1 \wedge \neg p_2)} \\ & \models\models & \overline{p_1 \wedge (p_2 \vee \neg p_2)} \\ & \models\models & \overline{p_1} \\ & \models\models & p_1 \end{array}$$

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

$$\begin{array}{lcl} q_1 \sqcap q_2 & = & p_1 \wedge \neg p_2 \wedge \neg p_2 \wedge p_3 \\ & = & p_1 \wedge \neg p_2 \wedge p_3 \end{array}$$

$$\begin{array}{lcl} q_1 \sqcup q_2 & = & \overline{q_1 \vee q_v} \\ & = & \overline{(p_1 \wedge \neg p_2) \vee (\neg p_2 \wedge p_3)} \end{array}$$

   Angenommen $p_1 = (x > 4)$ und $p_3 = (x > 6)$.
   Dann: $p_1 \vee p_3 \models p_1$ und damit $q_1 \sqcup q_2 \models\models \neg p_2 \wedge p_1$.

5.1.4 Definition

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

$$q_\sigma := \wedge\{l \in (P \cup \neg P) \mid \sigma \vDash l\}$$

Dann gilt für $State' \subseteq State$:

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

5.1.5 Beispiel

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

$$p_1 = (x \leq y), \quad p_2 = (x = y), \quad p_3 = (x > y),$$

$$\sigma_1 = (x \mapsto 1, y \mapsto 2), \quad \sigma_2 = (x \mapsto 2, y \mapsto 2).$$

Dann ist

$$\begin{array}{lcl} \alpha(\{\sigma,\sigma_2\}) & = & q_{\sigma_1} \sqcup q_{\sigma_2} \\ & = & \underline{(p_1 \wedge \neg p_2 \wedge \neg p_3) \sqcup (p_1 \wedge p_2 \wedge \neg p_3)} \\ & = & (p_1 \wedge \neg p_2 \wedge \neg p_3) \vee (p_1 \wedge p_2 \wedge \neg p_3) \\ & = & p_1 \wedge p_3 \end{array}$$

5.2 Abstrakte Semantik zur Prädikatenabstraktion

Ziel:

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

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

  für alle $q \in \text{Abs}(P)$

Idee:

- Sei $x := a$ der Befehl, der von c zu c' führt
- Die Menge

  $$post_{c,c'}(\gamma(q))$$

  ist gegeben durch die *stärkste Nachbedingung (strongest postcondition)* sp(q,x:=a):

  $$post_{c,c'}(\gamma(q)) = \{\sigma \in \text{State} \mid \sigma \vDash sp(q, x := a)\}$$

Trick:

Dann ist

$$\begin{array}{lcl} \alpha(\text{post}_{c,c'}(\gamma(q))) & = & \alpha(\{\sigma \in \text{State} \mid \sigma \vDash sp(q, x := a)\}) \\ & = & \sqcup\{q_\sigma \mid \sigma \vDash sp(q, x := a)\} \\ & \models\!\models & \overline{sp(q, x := a)} \end{array}$$

Gut:

Mit dem Trick ist keine *Konkretisierung* über σ und q_σ notwendig.

Schlecht:

Stärkste Nachbedingung dennoch wegen *Quantoren* nachteilig.

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

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

  Das heißt, wenn wir c in einem Zustand ausführen, der b erfüllt, erfüllen alle möglichen Folgezustände nach dem Ausführen von c Prädikat p.
  (Partielle Korrektheit: muss nur für terminierende Ausführungen von c erfüllt sein.)
- Für ein Prädikat $b \in BExp$ und ein Programm c bezeichnen wir mit $sp(b, c)$ die *stärkste (begl. $\vDash$) Formel* p, für die $\{b\}c\{p\}$ gültig ist. Diese Formel heißt *Stärkste Nachbedingung (strongest postcondition).*
- Gegeben $p \in BExp$ und ein Programm c, bezeichnen wir mit $wp(x := a, p)$ die schwächste Formel b, für die $\{b\}c\{p\}$ gültig ist. Diese Formel heißt *schwächste Vorbedingung (weakest precondition).*

5.2.2 Satz (Dijkstra '76, aus FGdP bekannt)

$$\begin{array}{rcl} sp(b, x := a) & \models\!\!\models & \exists x' \colon ((b\{x := x\}) \wedge x = (a\{x := x'\}) \\ wp(x := a, p) & \models\!\!\models & p\{x := a\} \end{array}$$

Hierbei ist für Ausdrücke exp, x, y der Ausdruck $exp\{x := y\}$ als der Ausdruck, der entsteht, wenn wir alle Vorkommen von x in exp durch y ersetzen, definiert. Unter anderem ist auch die Notation $exp\{y/x\}$ dafür üblich.

Die Intuition hinter der Formel für $sp(b, x := a)$ ist, dass x' den alten Wert von x zwischenspeichert.

Beobachtung:

Beide Formeln existieren und lassen sich berechnen. Aber: Stärkste Nachbedingungen benötigen Quantoren, die für Tools (Satisfiability modulo theories-solver) schwer zu handhaben sind.

SMT

5.2.3 Beispiel

$$
\begin{array}{rcl}
sp(y > 5, x := y + 3) & \models\!\models & \exists x' \colon (y > 5\{x := x'\} \land x = (y + 3)\{x := x'\}) \\
& \models\!\models & \exists x' \colon (y > 5 \land x = y + 3) \\
& \models\!\models & y > 5 \land x = y + 3 \\
\\
sp(x > 5 \land y > 3, x := x + y) & \models\!\models & \exists x' : ((x > 5 \land y > 3\{x := x'\}) \land x = (x + y\{x := x'\})) \\
& \models\!\models & \exists x' : (x' > 5 \land y > 3 \land x = x' + y) \\
\\
wp(x := y + 7, x > 5) & \models\!\models & x > 5\{x := y + 7\} \\
& \models\!\models & y + 7 > 5 \\
& \models\!\models & y > -2
\end{array}
$$

5.2.4 Satz (Dijkstra '76)

$sp(b, x := a) \vDash p$ gdw. $b \vDash wp(x := a, p)$

5.2.5 Definition

Wir haben in der Vergangenheit bereits abstrakte Transitionsrelationen für Galois-Verbindungen etc. definiert.

Nun definieren wir die abstrakte Transitionsrelation zur Prädikatenabstraktionen

$$\Rightarrow \subseteq (\mathcal{P}rog \times \mathrm{Abs}(P)) \times ((\mathcal{P}rog \times \mathrm{Abs}(P)) \cup \mathrm{Abs}(P))$$

mittels folgender Regeln:

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

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

$$\frac{(c_1, q) \Rightarrow q;}{(c_1; c_2, q) \Rightarrow (c_2, q)}$$

$$\frac{(c_1, q) \Rightarrow (c_1', q')}{(c_1; c_2, q) \Rightarrow (c_1'; c_2, q')}$$

$$\frac{}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end}, q) \quad \Rightarrow \quad (c_1, \overline{q \wedge b})}$$

$$\frac{}{(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end}, q) \quad \Rightarrow \quad (c_2, \overline{q \wedge \neg b})}$$

$$\frac{}{(\texttt{while } b \texttt{ do } c \texttt{ end }, q) \quad \Rightarrow \quad (c; \texttt{while } b \texttt{ do } c \texttt{ end }, \overline{q \wedge b})}$$

$$\frac{}{(\texttt{while } b \texttt{ do } c \texttt{ end }, q) \quad \Rightarrow \quad \overline{q \wedge \neg b}}$$

Beobachtung:

- 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.

5.2.6 Definition

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

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

Beachte: Dadurch dass c' fixiert ist, ist $post^{\#}_{c,c'}$ eine Funktion.

5.2.7 Satz

Die Familie von Funktionen $post^{\#}_{c(,c')}$ ist die genaueste abstrakte Semantik

Beweis:
Nach der Definition ist zu zeigen, dass

$$\alpha\left(post_{c(,c')}\left(\gamma(q)\right)\right) \models\!\!=\!\!\models post^{\#}_{c(,c')}(q)$$

für alle $q \in \mathrm{Abs}(P)$ und alle c, c' gilt.

Alle Fälle außer der folgende sind leicht zu zeigen. Wir beschränken uns also auf den Fall $c :=$ `x := a`, müssen also zeigen

$$\alpha\left(post_{x:=a}\left(\gamma(q)\right)\right) \models\!\!=\!\!\models post^{\#}_{x:=a}(q).$$

Mit Hilfe der Hilfsaussagen, die wir im Anschluss zeigen werden, lässt sich beweisen:

$$\begin{array}{ll} & \alpha\left(post_{x:=a}\left(\gamma(q)\right)\right) \\ \overset{\text{(Lemma 1)}}{=} & \alpha\left(\{\sigma \in \text{State} \mid \sigma \models sp(q, x := a)\}\right) \\ \overset{\text{(Def } \alpha\text{)}}{=} & \bigsqcup\{q_\sigma \mid \sigma \models sp(q, x := a)\} \\ \overset{\text{(Satz 2)}}{\models\!\!=\!\!\models} & \overline{sp(q, x := a)} \\ \overset{\text{(Def } \Rightarrow\text{)}}{\models\!\!=\!\!\models} & post^{\#}_{x:=a}(q) \end{array}$$

□

5.2.8 Lemma (Lemma 1)
Es gilt:

$$post_{x:=a}\left(\gamma(q)\right) = \{\sigma \in \text{State} \mid \sigma \models sp(q, x := a)\}$$

Beweis:
Die folgende Kette von Äquivalenzen beweist die Aussage:

$$\begin{array}{ll} & \sigma \in post_{x:=a}\left(\gamma(q)\right) \\ \text{gdw.} & \exists \sigma' \in State: \quad \sigma' \in \gamma(q) \text{ und } (x := a, \sigma') \to \sigma \\ \text{gdw.} & \exists \sigma' \in State: \quad \sigma' \models q \text{ und } (x := a, \sigma') \to \sigma \\ \text{gdw.} & \sigma \models sp(q, x := a) \end{array} \quad .$$

Die letzte Äquivalenz ist nicht-trivial und wird nun bewiesen.

" $\Rightarrow$ " $\{q\}x := a\{sp(q, x := a)\}$ ist ein gültiges Hoare-Tripel. Nach der Definition gilt nun zusammen mit der Voraussetzung also $\sigma \models sp(q, x := a)$.

" $\Leftarrow$ " Angenommen $\sigma \models sp(q, x := a)$. Nach Dijkstra gilt

$$sp(b, x := a) \models\!\!=\!\!\models \exists x' \colon \left((b\{x := x\}) \wedge x = (a\{x := x'\})\right),$$

d.h. es gibt ein $d \in \mathbb{Z}$ mit:

$$\begin{array}{lll} \sigma[x' := d] & \models & q'\{x = x'\} \qquad (*) \\ \sigma[x' := d] & \models & x = a\{x := x'\} \qquad (**) \end{array}$$

Aus (**) folgt:

$$\begin{array}{lll} \sigma(x) & = & (\sigma[x' := d])(x) \\ & = & \mathcal{S}[\![ax := x']\!](\sigma[x' := d]) \\ & = & \mathcal{S}[\![a]\!](\sigma[x' := d]) \qquad (***) \end{array}$$

wobei die letzte Gleichheit aus dem Substitutionslemma (Logik) folgt, da in a die "frische"Variable x' nicht vorkommt und in $ax := x'$ die ersetzte Variable x nicht mehr vorkommt.

Wir zeigen nun die gewünschten Eigenschaften

1) $\sigma[x' := d] \models q$
2) $(x := a, \sigma[x' := d]) \to \sigma$

Daraus folgt, dass $\sigma[x' := d]$ die Bedingungen, die wir an σ' gestellt haben erfüllt und damit die Existenz eines solchen σ' bewiesen ist.

Beweis von 1) Da mit (*) $\sigma[x' := d] \models q\{x := x'\}$ gilt, folgt $\sigma[x := d] \models q$.

Beweis von 2) Es gilt

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

Für die Belegung auf der rechten Seite gilt:

$$\begin{array}{ll} & (\sigma[x' := d])[x := \mathcal{S}[\![a]\!](\sigma[x' := d])] \\ = & \sigma[x := \mathcal{S}[\![a]\!](\sigma[x' := d])] \\ \overset{(***)}{=} & \sigma[x := \sigma(x)] = \sigma \end{array}$$

□

5.2.9 Satz (Satz 2)
Es gilt:

$$\overline{sp(q, x := a)} \models\!\models \bigsqcup \{q_\sigma \mid \sigma \models sp(q, x := a)\}$$

Der Beweis des Satzes benötigt die folgenden Lemmata.

5.2.10 Lemma (Hilfslemma)
(HL1) $b_1 \models b_2$ impliziert $\overline{b_1} \models \overline{b_2}$.

(HL2) $\overline{\overline{b}} \models\!\models b$

(HL3) $\sigma \models q_\sigma$

Beweis:
Übungsaufgabe. □

5.2.11 Lemma (Lemma 3)
(1) $sp(b, x := a) \models \bigvee\{q_\sigma \mid \sigma \models sp(b, x := a)\}$

(2) $\bigvee\{q_\sigma \mid \sigma \models sp(b, x := a)\} \models \overline{sp(b, x := a)}$

Beweis:
Zu (1): Gelte $\sigma' \models sp(b, x := a)$. Da $\sigma' \models q'_\sigma$ (HL3) und $q_{\sigma'}$ eines der q_σ in der Disjunktion ist, folgt die gesamte Disjunktion.

Zu (2): Gelte $\sigma' \models \bigvee\{q_\sigma \mid \sigma \models sp(b, x := a)\}$.

Es gilt

$$\overline{sp(b, x := a)} \models\!\models \bigwedge \{l \in (P \cup \neg P) \mid sp(b, x := a) \models l\}$$

Um $\overline{sp(b, x := a)}$ aus der Disjunktion zu folgern, reicht es also, jedes $l \in (P \cup \neg P)$ zu folgern, für das $sp(b, x := a) \models l$ gilt.

Sei so ein l gegeben. Da $\sigma' \models \bigvee\{q_\sigma \mid \sigma \models sp(b, x := a)\}$ gilt, folgt $\sigma \models q_{\sigma'}$ für ein $\sigma' \in$ State mit $\sigma' \models sp(b, x := a)$.

Da $\sigma' \models sp(b, x := a)$ und $sp(b, x := a) \models l$, folgt mit der Transitivität von " $\models$ " nun $\sigma' \models l$.

Damit ist $l \in \{l' \in (P \cup \neg P) \mid sp(b, x := a) \models l'\}$. Diese Menge definiert jedoch $q_{\sigma'}$:

$$q_{\sigma'} = \bigwedge \{l' \in (P \cup \neg P) \mid \sigma' \models l'\}\,.$$

Da $\sigma \models q_{\sigma'}$ gilt, folgt nun wie gewünscht $\sigma \models l$.

□

Beweis von Satz 2:
Zu zeigen war:

$$\overline{sp(q, x := a)} \models\!\models \bigsqcup \{q_\sigma \mid \sigma \models sp(q, x := a)\}\,.$$

" $\models$ " Mit Lemma 3, (1) gilt:

$$sp(q, x := a) \quad \models \quad \bigvee\{q_\sigma \mid \sigma \models sp(b, x := a)\}$$

Mit (HL1) gilt:

$$\overline{sp(q, x := a)} \quad \models \quad \overline{\bigvee\{q_\sigma \mid \sigma \models sp(b, x := a)\}}$$

$$\overset{\text{Def } \sqcup}{\models\!\models} \quad \{q_\sigma \mid \sigma \models sp(q, x := a)\}$$

" $=\!\!|$" Lemma 3, (2) liefert:

$$\bigvee\{q_\sigma \mid \sigma \models sp(b, x := a)\} \quad \models \quad \overline{sp(q, x := a)}$$

Mit (HL1) gilt:

$$\overline{\underbrace{\bigvee\{q_\sigma \mid \sigma \models sp(b, x := a)\}}_{(lhs)}} \quad \models \quad \overline{\underbrace{\overline{sp(q, x := a)}}_{(rhs)}}$$

Nach der Definition von $\sqcup$ gilt $(lhs) \models\!\models \bigsqcup\{q_\sigma \mid \sigma \models sp(q, x := a)\}$ und mit (HL2) folgt $(rhs) \models\!\models \overline{sp(q, x := a)}$.

□

5.2.12 Beispiel

```
if [x > y]^1 then
    while [y ≠ 0]^2 do
        [x := x − 1]^3;
        [y := y − 1]^4;
    end
    if [x > y]^5 then
        [skip]^6;
    else
        [skip]^7;
    end
else
    [skip]^8;
end
```

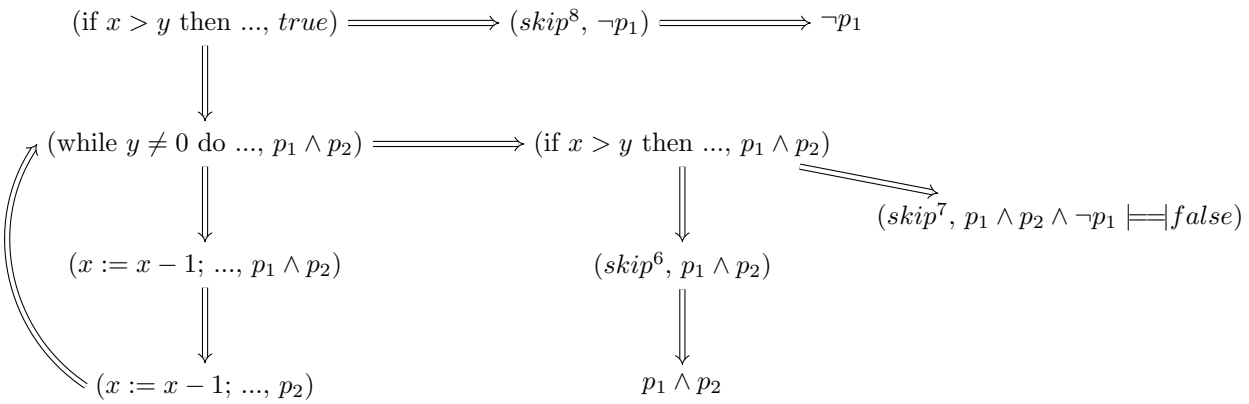
Behauptung:

Block 7 wird nie betreten, da $x > y$ Invariante der while-Schleife ist.

Wir beweisen dies, indem wir das abstrakte Transitionssystem bezüglich der Prädikatenabstraktion mit den Prädikaten

$$p_1 = x > y \qquad \text{und} \qquad p_2 = x \geq y$$

aufstellen.



Es ist abzulesen, dass Block 7 *nicht* erreichbar ist, da er nur mit dem Cube $false$ auftritt.

Problem:

Um die Prädikatenabstraktion zu implementieren, müssen wir $\overline{sp(q, x := a)}$ berechnen.

Der folgende Satz reduziert das Problem auf die Prüfung von Implikation (auf Gültigkeit). Zu beachten ist, dass der Satz von Dijkstra für wp (im Gegensatz zu sp) eine quantorenfreie Darstellung liefert.

5.2.13 Satz (Graf & Saidi, '97)

$$\overline{sp(q, x := a)} \models\!\models \bigwedge\{l \in P \cup \neg P \mid \vDash p \to wp(x := a, l)\}$$

Beweis:

$$\begin{array}{ll} & sp(q, x := a) \\ \models\!\models & \wedge\{l \in P \cup \neg P \mid sp(q, x := a) \vDash l\} \\ \overset{\text{Dijkstra}}{\models\!\models} & \bigwedge\{l \in P \cup \neg P \mid q \vDash wp(x := a, l)\} \\ \models\!\models & \bigwedge\{l \in P \cup \neg P \mid\vDash q \to wp(x := a, l)\} \end{array}$$

□

Um zu überprüfen, ob $\vDash q \to wp(x := a, l)$ gilt nutze SMT-Solver. Allgemeingültigkeit prädikatenlogischer Formeln ist im Allgemeinen nicht entscheidbar. Damit Solver terminiert, sollte die Formel in einem entscheidbaren Fragment der Prädikatenlogik liegen.

5.2.14 Beispiel
In welchen Theorien ist Allgemeingültigkeit entscheidbar?

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

5.3 Abstraktionsverfeinerung

Ziel:

CEGAR-Loop (siehe Illustration)

Probleme:

- Wie prüft man, ob Gegenbeispiel *echt* oder *spurious* (unecht), ist, d.h. ob es wirklich oder nur wegen der zu groben Abstraktion existiert?
- Wie extrahiert man neue Prädikate aus einem spurious Gegenbeispiel, so dass dieses Gegenbeispiel in der Verfeinerung nicht mehr auftreten kann?

Typische Eigenschaften φ:

- 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_{bad}, σ) formulieren, wobei c_{bad} ein unerwünschtes Programm ist (*Safety-Verifikation*).

5.3.1 Definition (Gegenbeispiel)
Betrachte die abstrakte Semantik des Programms c unter Abs(P).

Sei c_{bad} das unerwünschte Programm.

- Ein *Gegenbeispiel* ist eine Folge abstrakter Transitionen.

  $$(c_0, \text{true}) \Rightarrow (c_1, q_1) \Rightarrow \ldots \Rightarrow (c_k, q_k)$$

  mit $c_0 = c, c_k = c_{\text{bad}}$ und $q_i \not\models\!\!\models \text{false}$ für alle $1 \leq i \leq k$
- Das Gegenbeispiel heißt *echt*, falls es Zustände $\sigma_0, \ldots, \sigma_k \in \text{State}$ gibt mit $\sigma_i \vDash q_i$ und

  $$(c, \sigma_0) \to (c_1, \sigma_1) \to \ldots \to (c_{bad}, \sigma_k),$$

  ansonsten *spurious*.

- Im folgenden identifizieren wir eine Sequenz von Programmen

  $$(c_0, ...) \to (c_1, ...) \to \ldots (c_k, ...)$$

  mit dem azyklischem Programm $r = r_1; ...; r_k$, dass der Folge von Befehlen entspricht, die beim Auswerten der Sequenz abgearbeitet werden.
- Dabei werden if und while zu assume b bzw. assume $\neg b$, je nachdem welcher Fall ausgewertet wird.

Die Sequenz

$$\begin{array}{c} \text{x := x + 1; if } y \neq 0 \text{ then x := x - y else y := 2; skip; end;} \\ \downarrow \\ \text{if } y \neq 0 \text{ then x := x - y else y := 2; skip; end;} \\ \downarrow \\ \text{y := 2; skip;} \\ \downarrow \\ \text{skip;} \end{array}$$

wird beispielsweise mit der Befehlsfolge

$$r \;=\; x := x + 1; \text{assume } \neg(y \neq 0); y := 2;$$

identifiziert.

5.3.2 Lemma
Das Gegenbeispiel

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

ist *spurious* gdw. es Prädikate $p_0, \ldots, p_k$ mit

- $p_0 = \text{true}$ und $p_k = \text{false}$
- für alle $1 \leq i \leq k$ ist $\{p_{i-1}\}r_i\{p_i\}$ ein gültiges Hoare-Tripel,
  d.h. für alle $\sigma, \sigma' \in \text{State}$ mit $\sigma \vDash p_{i-1}$ und $(c_{i-1}, \sigma) \to (c_i, \sigma')$ gilt $\sigma' \vDash p_i$.

Beweis:

- Definiere p_0 als $true$.
- Für alle $i > 0$, definiere p_i als stärkste Nachbedingung von r_i und p_{i-1}.
- Dadurch gilt insbesondere:

(skip)	$p_i := p_{i-1}$
(x:= a)	$p_i := \exists x' : (p_{i-1}\{x := x'\} \wedge x = (a\{x := x'\}))$
(assume b)	$p_i := p_{i-1} \wedge b$

□

Korollar
Ein Gegenspiel ist also spurious gdw. $\{\text{true}\}r\{\text{false}\}$ ein gültiges Hoare-Tripel ist. Dies gilt, falls

$$\text{true} \models\!\!\!\models wp(r, false) \qquad \text{oder äquivalent} \qquad sp(true, r) \models\!\!\!\models \text{false}\,.$$

5.3.3 Beispiel

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

Eigenschaft:

Block 4 nicht erreichbar.

Initiale Abstraktion:

$P = \emptyset$, also $\mathrm{Abs}(P) = \{\mathrm{false}, \mathrm{true}\}$.

Spurious Gegenbeispiel:

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

(Statt den kompletten Programmen sind hier nur die Zeilennummern angegeben, ab denen das Programm ausgeführt wird.)

Konstruktion der Prädikate wie im Beweis des obigen Lemmas:

$$\begin{array}{lcll}
p_0 & = & & \mathrm{true} \\
p_1 & = & & \exists x' : (p_0\{x'/x\} \wedge x = (z\{x'/x\})) \\
 & & \models\!\models & \exists x' : (\mathrm{true} \wedge x = z) \models\!\models x = z \\
 & & & \\
p_2 & = & & \exists z' : (p_1\{z'/z\} \wedge z = (z + 1\{z'/z\})) \\
 & & \models\!\models & \exists z' : (x = z' \wedge z = z' + 1) \\
 & & & \\
p_3 & = & & \exists y' : (p_2\{y'/y\} \wedge y = (z\{y'/y\})) \\
 & & \models\!\models & \exists z' : (x = z' \wedge z = z' + 1) \wedge y = z \\
 & & & \\
p_4 & = & & p_3 \wedge y = x \\
 & & \models\!\models & \exists z' : (x = z' \wedge z = z' + 1) \wedge y = z \wedge y = x \\
 & & \models & z = x + 1 \wedge y = z \wedge y = x \\
 & & \models & y = x + 1 \wedge y = x \\
 & & \models & \mathrm{false}
\end{array}$$

Abstraktionsverfeinerung:

- Seien $p_1, \ldots, p_{n-1}$ die neu bestimmten Prädikate wie im Beweis des Lemmas.
- Definiere $P' := P \cup \{p_1, \ldots, p_{n-1}\}$

5.3.4 Lemma

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
- $q_k' \not\models \text{false}$

mehr.

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:

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

Beweis des Lemmas:

- Angenommen es gibt eine Folge:

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

  mit $q_k' \not\models \text{false}$
- Die abstrakte Semantik ist über stärkste Nachbedingung definiert:

  $$q_{i+1}' := \overline{sp(q', r_{i+1})}$$

- Da q_1' die stärkste Nachbedingung von true und r_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:

– Angenommen $q_i \models p_i$. Dann folgt durch die Monotonie (bezüglich $\models$) von $sp(\cdot, r_{i+1})$:

$$sp(q_i', r_{i+1}) \models sp(p_i, r_{i+1}) \models p_{i+1}$$

– Mit (HL1) gilt dann auch

$$\overline{sp(q_i', r_{i+1})} \models \overline{p_{i+1}}.$$

– Da aber $\overline{sp(q_i', r_{i+1})} \models q_{i+1}'$ und $\overline{p_{i+1}} \models\!\models p_{i+1}$, gilt $q_{i+1}' \models p_{i+1}$.

– Also mit Induktion: $q_i' \models p_i$ für alle $0 \leq i \leq k$.

– Da $p_k \models\!\models \text{false}$, folgt insbesondere $q_k' \models \text{false}$ ↯

□

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 ;
end ;
if [a = b ∧ ¬(y = 0)]^5 then
      [skip]^6 ;
else
      [skip]^7 ;
end
```

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

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

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

⇒ Prädikate unnötigt komplex und beziehen sich auf irrelevante Variablen. Füge besser möglichst allgemeine Prädikate hinzu, die sich auf möglichst viele Zustände auswirken.

Lösung:

Craig-Interpolante

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

Eine Formel $r \in \text{BExp}$

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

heißt *Craig-Interpolante*.

5.4.2 Satz (Logik)
Craig-Interpolanten existieren in Aussagenlogik und in First-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

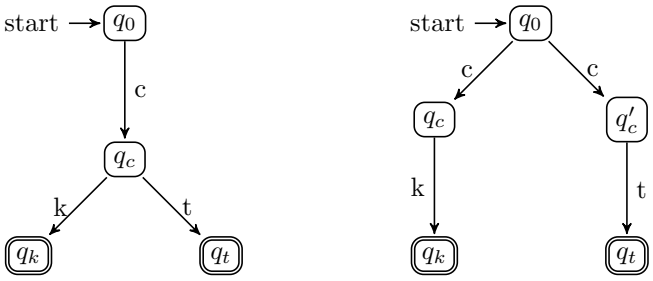
⇒ Auch über Craig-Interpolanten

6 Bisimulationsäquivalenz und Simulationsordnung

Motivation:

Wir betrachten zwei Getränkeautomaten, endliche Automaten über dem Alphabet

c // wirf Münze (coin) in den Automaten
t // drücke Taste für Tee
k // drücke Taste für Kaffee



Beide Automaten akzeptieren die gleiche Sprache $\{ck, ct\}$, aber ihr Verhalten unterscheidet sich: In den linken Automaten wirft man eine Münze und kann dann entscheiden, ob man Kaffee oder Tee möchte (gewünschtes Verhalten). Der rechte Automat entscheidet nichtdeterministisch, in welchen Zustand er geht, nachdem eine Münze eingeworfen wurde, und je nachdem welchen Zustand er wählt, kann man hinterher entweder nur Tee oder nur Kaffee bekommen.

Es macht also Sinn, zwischen diesen Automaten zu unterscheiden, obwohl sie sprachäquivalent sind. Wir werden sehen, dass diese Automaten nicht bisimulationsäquivalent sind.

Wir betrachten nun statt endlichen Automaten ein allgemeineres Setting: Kripke-Strukturen, d.h. zustandsbeschriftete Transitionssysteme.

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

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

Gut: Der Satz von Hennessy–Milner sagt, dass die beiden Systeme bezügliche bestimmer logischer Formeln gleich verhalten:

$$K_C \approx K_A \quad \text{gdw.} \quad \forall \varphi \in CTL^* \colon K_C \vDash \varphi \text{ gdw. } K_A \vDash \varphi$$

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 von KC nachahmen.

Schlecht: Nur die folgenden schwächeren Aussagen gelten:

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

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

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

6.1 Bisimulationsäquivalenz

Ziel:

Spielcharakterisierung. Entscheidbarkeit und Minimierung

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

- $(S, S_0, \to)$ ist ein Transitionssystem, d.h.
  - S ist eine Menge von Zuständen

  - $S_0 \subseteq S$ ist eine Menge von initialen Zuständen
  - $\rightarrow \subseteq S \times S$ ist eine *Transitionsrelation*
- AP ist die Menge der *atomarer Formeln*
- $l : S \rightarrow \mathcal{P}(AP)$ ist eine *Beschriftung* der Zustände

Wir nehmen dabei immer *Deadlock-Freiheit* an, d.h. jeder Zustand hat einen Nachfolger: $\forall s \in S : \exists s' \in S : s \rightarrow s'$.

Eine Kripke-Struktur heißt *endlich*, falls AP und S endlich sind.

Die *Klasse aller Kripke-Struktur* ist $\mathcal{K}$

6.1.2 Definition (Bisimulation und Bisimulationsäquivalenz)
Seien $K = (AP, S, S_0, \rightarrow, l)$ und $K' = (AP, S', S'_0, \rightarrow', l')$ 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) Die Beschriftung sind gleich, d.h. $l(s) = l'(s')$

(2) s' kann die von s ausgehenden Transitionen simulieren, d.h.

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

(3) s kann die von s' ausgehenden Transitionen simulieren, d.h.

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

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

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

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

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

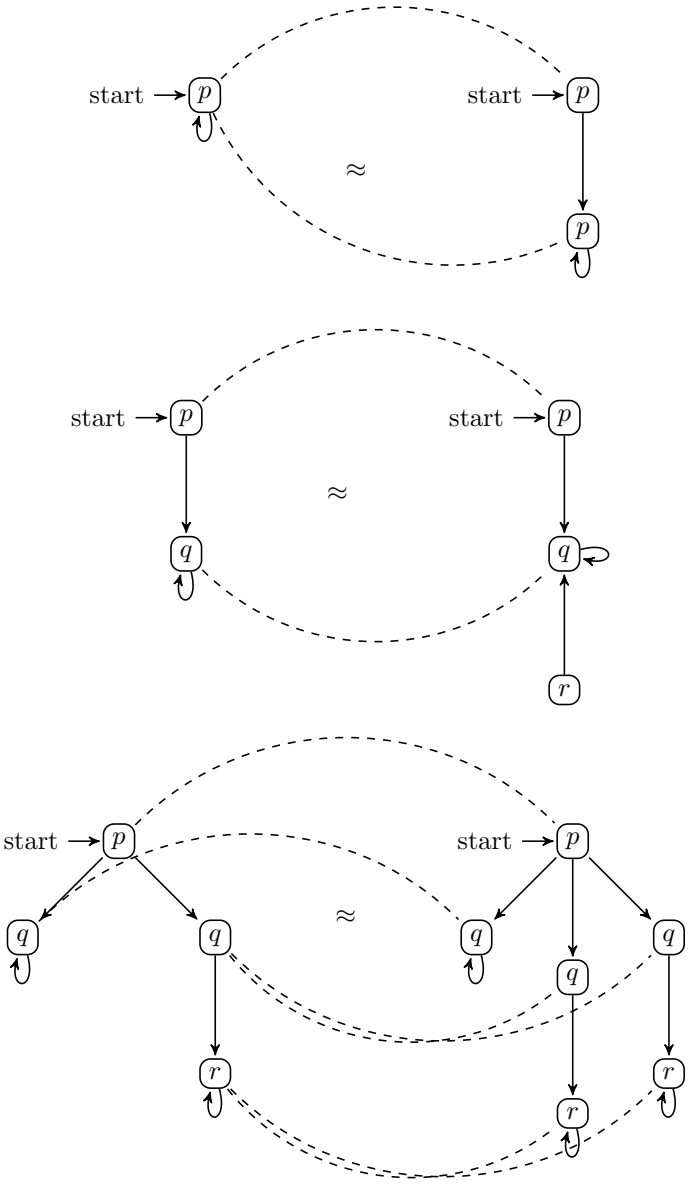
Beweis:
Seine $K^i = (AP, S^i, S^i_0, \rightarrow^i, l^i)$ mit $i = 1, 2, 3$

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 $R' \circ R := \{(s, s'') \in S^1 \times S^3, \exists s' \in S^2 : (s, s') \in R \text{ und } (s', s'') \in R'\}$

Der Beweis von Reflexivität und Symmetrie ist eine Übungsaufgabe. □

6.1.4 Beispiel

Die folgenden Paare von Kripke-Strukturen sind Bimulationsäquivalent. Die gestrichelten Linien geben dabei eine Bisimulationsäquivalenz an. Wie man am zweiten Beispiel sieht, spielen unerreichbare Teile keine Rolle.



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 der 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.

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

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

6.1.2 Entscheidbarkeit

Wie entscheidet man für endliche Kripke-Strukturen K, K' über der selben Menge AP, ob $K \approx K'$ gilt?
Bemerkung. Wenn R und R' zwei Bisimulationsäquivalenzen zwischen K und K' sind, dann ist auch $R \cup R'$ eine Bisimulationsäquivalenz. Es gibt also insbesondere eine größte Bisimulationsäquivalenz zwischen K und K'.

6.1.6 Lemma
$K \approx K'$ gdw. die größte Bisimulationsrelation R die Startzustände verbindet.

Dabei ist

$$R = \bigcup_{\substack{R' \subseteq S \times S \\ R' \text{ ist Bisim.}}} R' \ .$$

Wie berechnet man die größte Bisimulationsrelation?

Idee:

- Zunächst ist die volle Relation $S \times S'$ ein Kandidat für die Bisimulation.
- Dann werden iterativ Paare $(s, s') \in S \times S'$ entfernt, bei denen Beschriftung oder Nachfolger nicht passen $\Rightarrow$ Fixpunkt!

6.1.7 Definition

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 für $Q \subseteq S \times S'$ definiert durch:

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

Sie entfernt alle Paare aus Q, bei denen Beschriftung oder Nachfolger nicht passen. Da durch das Entfernen von Paaren eventuell bei anderen Paaren nun die Nachfolger nicht mehr passen, muss die Funktion mehrfach angewandt werden.

6.1.8 Lemma

$f_{\approx}$ ist monoton.

Da $f_{\approx}$ monoton, ist

$$S \times S' \supseteq f_{\approx}(S \times S') \supseteq \ldots$$

eine absteigende Kette, die gegen größten Fixpunkt konvergiert:

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

6.1.9 Lemma

(1) Jeder Fixpunkt von $f_{\approx}$ ist eine Bisimulation, insbesondere $\text{gfp}(f_{\approx})$.

(2) Jede Bisimulation $R \subseteq S \times S'$ ist ein Fixpunkt von $f_{\approx}$

6.1.10 Satz

$\text{gfp}(f_{\approx})$ ist größte Bisimulation.

Beweis:

- Da mit (1) jede Bisimulation, und damit insbesondere die größte R, Fixpunkt von $f_{\approx}$, gilt mit Def. des größten Fixpunkts:

$$R \subseteq \text{gfp}(f_{\approx}) \; .$$

- (2) zeigt, dass gfp($f_{\approx}$) wieder Bisimulation ist, daher muss Gleichheit gelten.

□

Korollar
$K \approx K'$ gdw. gfp($f_{\approx}$) verbindet die Startzustände

Bemerkung. Analog zur Minimierung von endlichen Automaten kann man die größte Bisimulation zwischen einer Kripke-Struktur und einer Kopie von ihr nutzen, um die minimale zu ihr bisimulationsäquivalente Struktur zu finden.

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)
- LTL und CTL sind unvergleichbar
- CTL* = Verallgemeinerung von beiden.

Hier: CTL

6.2.1 Definition (Syntax von CTL)
Sei AP eine Menge atomarer Formeln.

Die *Menge der CTL-Formeln* ist wie folgt definiert:

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

(mit $p \in AP$)

Außerdem verwenden wir folgende Abkürzungen:

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

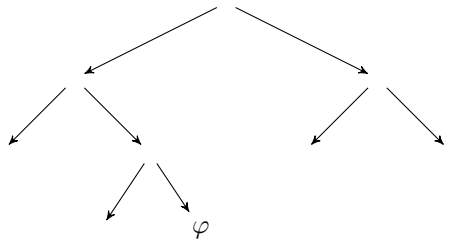
Bedeutung:

- E: exists, es gibt eine Pfad, auf dem gilt...
- A: all, für alle Pfade gilt ...

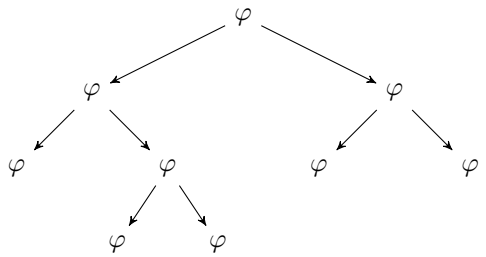
- X: next, für einem Nachfolgezustand gilt...
- F: finally, schließlich (für einen Zustand im Pfad) gilt ...
- G: globally, für alle Knoten im Pfad gilt...

Intuition:

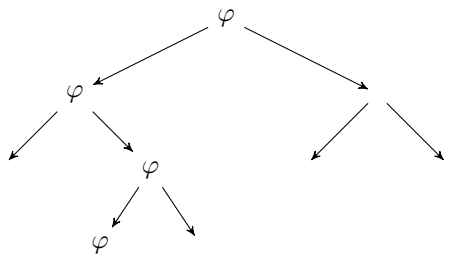
- EF φ : *Es gibt* einen Pfad, auf dem schließlich φ gilt



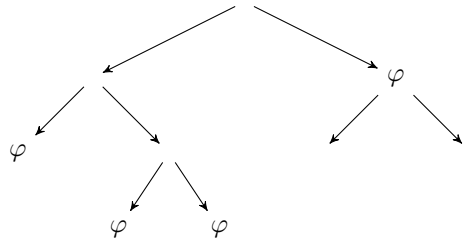
- AG φ : Auf *allen* Pfaden gilt immer φ



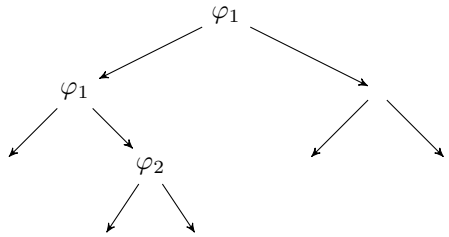
- EG φ : *Es gibt* einen Pfad, auf dem immer φ gilt



- 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.



Berechnungslogiken wie CTL machen von einem gegebenen Punkt in einer Struktur (Zeitpunkt) aus Aussagen über die Nachfolger des Punktes (Zukunft). Dementsprechend ist auch die Erfülltheitsrelation definiert.

6.2.2 Definition (Semantik von CTL)
Sei $K = (\mathrm{AP}, S, S_0, \rightarrow, l)$ 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	$\vDash \neg\varphi$	, 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 $K, s_1 \vDash \varphi$ gibt
K, s	$\vDash EG\varphi$	, falls es einen (unendlichen) Pfad $\Pi = s_0 s_1 s_2 \ldots$ mit $s_0 = s$ und $K, s_i \vDash \varphi\ \forall_i$ gibt
K, s	$\vDash E(\varphi_1 \mathcal{U} \varphi_2)$	, falls es einen (unendlichen) Pfad $\Pi = s_0 s_1 s_2 \ldots$ mit $s_0 = s$ gibt und es auf diesem Pfad ein $j \geq 0$ gibt mit $K, s_i \vDash \varphi_1$ für alle $0 \leq i < j$ und $K, s_j \vDash \varphi_2$

Wir schreiben $K \models \varphi$, wenn für alle Startzustände $s_0 \in S_0$ gilt: $K, s_0 \models \varphi$.

Bemerkung. Wir fordern für $E(\varphi_1 \mathcal{U} \varphi_2)$, dass es einen (unendlichen) Pfad gibt, auf dem auf einem endlichen Präfix φ_1 gilt und dann zu einem (endlichen) Zeitpunkt φ_2 gilt. Ein unedlicher Pfad, auf dem auf jedem Knoten φ_1 gilt, erfüllt diese Eigenschaft *nicht*!

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:

Gilt $K \vDash \varphi_0$?

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 kann man insbesondere das Model-Checking-Problem lösen, indem man die Inklusion

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

überprüft.

Hierzu iterieren wir über die Teilformeln und berechnen die Mengen $S(\varphi)$, wobei in der i-ten Iteration die Teilformeln der Tiefe $i \in \mathbb{N}$ betrachtet.

Dabei ist die Tiefe wie folgt definiert:

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

Mit $cl(\varphi_0)$ bezeichnen wir die Menge der Teilformeln von φ_0 (*closure*, Fisher-Ladner-Abschluss).

Algorithmus:

Input: $K = (\mathrm{AP}, S, S_0, \rightarrow, l)$ 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);
        end
    end
end
```

Der Unteralgorithmus Check(φ) befüllt die Menge $S(\varphi)$ und ist abhängig von der Form von φ.

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

$S(p) = \{s \in S \mid p \in l(s)\}$

(2) Check($\neg\varphi$):

$S(\neg\varphi) := S \setminus S(\varphi)$

(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:

$$E(\varphi_1 \mathcal{U} \varphi_2) \models\!\!\!\models \varphi_2 \vee (\varphi_1 \wedge EX(E(\varphi_1 \mathcal{U} \varphi_2)))$$

Implementierung:

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

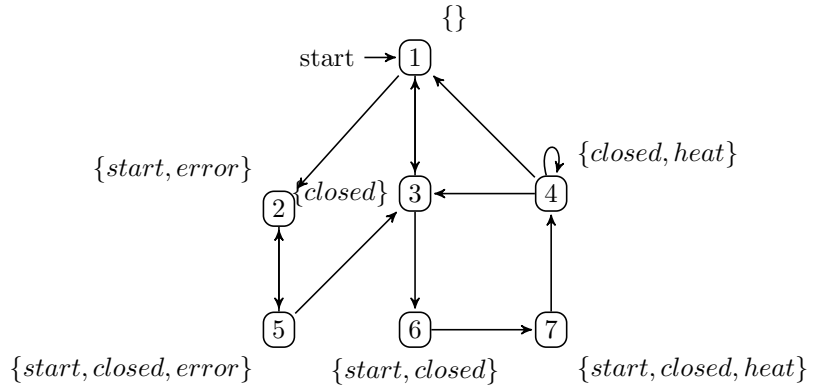
(6) Check($EG\varphi$):

Hausaufgabe.

Bemerkung. Der Algorithmus Check benutzt nur Mengen, die bereits in vorherigen Iterationen befüllt wurden. Sei zum Beispiel φ eine Formel mit Tiefe i. Dann hat $\neg\varphi$ Tiefe $i+1$, und beim Ausrechnen von $S(\neg\varphi)$ (in der $i+1$-ten Iteration) steht $S(\varphi)$ zur Verfügung (wurde in der i-ten Iteration berechnet).

6.2.3 Beispiel

Sei die Kripke-Struktur K_M, die eine Mikrowelle beschreibt und durch das folgende beschriftete Transitionssystem definiert ist, gegeben.



Wir möchten überprüfen ob die Mikrowelle nachdem Start gedrückt wurde (d.h. Zustand ist mit *start* gelabeled) immer irgendwann einen Zustand erreicht, in dem der Inhalt ausgewärmt wurde (d.h. Zustand ist mit *heat* gelabeled). Wir möchten also prüfen, ob

$$K_M \models AG(start \rightarrow AFheat)$$

gilt. Wenn wir die Abkürzungen ausfalten, müssen wir also überprüfen, ob $K_M \models \varphi_0$

gilt, mit

$$\varphi_0 = \neg\left(E\left(true\ \mathcal{U}\left(start \wedge EG\neg heat\right)\right)\right)$$

Wir berechnen iterativ:

$$\begin{aligned}
S(start) &= \{2,5,6,7\} \\
S(heat) &= \{4,7\} \\
S(\neg heat) &= \{1,2,3,5,6\} \\
S(EG\neg heat) &= \{1,2,3,5\} \\
S(start \wedge EG\neg heat) &= S(start) \cap S(EG\neg heat) = \{2,5\} \\
S(E\left(true\ \mathcal{U}\left(start \wedge EG\neg heat\right)\right)) &= S = \{1,2,3,4,5,6,7\} \\
S(\varphi_0) &= S \setminus S = \emptyset
\end{aligned}$$

Insbesondere gilt also $K_M \models \varphi_0$ *nicht*.

6.2.2 Satz von Hennessy & Milner

Ziel:

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

Wir zeigen beiden Richtungen jeweils als Lemma. Für die Richtung "links nach rechts"benötigen wir den Begriff der entsprechenden Pfade.

6.2.4 Definition
Seien $K = (AP, S, S_0, \rightarrow, l)$ und $K = (AP, S', S_0', \rightarrow', l')$ zwei Kripke-Strukturen und sei $R \subseteq S \times S'$ eine Bisimulationsäquivalenz zwischen ihnen. Zwei Pfade

$$\pi = s_0 s_1 \ldots \text{ mit } s_i \in S\ \forall i \in \mathbb{N}$$

$$\pi' = s_0' s_1' \ldots \text{ mit } s_i' \in S'\ \forall i \in \mathbb{N}$$

heißen *entsprechend*, falls $(s_i, s_i') \in R$ für alle $i \in \mathbb{N}$.

6.2.5 Lemma
Seien $K = (AP, S, S_0, \rightarrow, l)$ und $K = (AP, S', S_0', \rightarrow', l')$ zwei Kripke-Strukturen und sei $R \subseteq S \times S'$ eine Bisimulationsäquivalenz zwischen ihnen. Sei $(s, s') \in R$ ein Paar von bisimulationsäquivalenten Zuständen.

- Dann gibt es zu jedem Pfad

  $$\pi = s s_1 s_2 \ldots \text{ der in } s \text{ beginnt}$$

  einen entsprechenden Pfad

  $$\pi' = s' s_1' s_2' \ldots \text{ der in } s' \text{ beginnt.}$$

- Analog gibt es zu jedem Pfad, der in s' beginnt, einen entsprechenden Pfad, der in s beginnt.

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

Beweis:
Seien $K = (AP, S, S_0, \to, l)$ und $K = (AP, S', S'_0, \to', l')$ und sei $R \subseteq S \times S'$ eine Bisimulationsäquivalenz zwischen ihnen.

Wir zeigen für alle bisimulationsäquivalenten Zustandspaare $(s, s') \in R$ und alle CTL-Formeln φ:

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

Beweis durch Induktion nach der Struktur von φ.

IA: $\varphi = p \in AP$

Aus $(s, s') \in R$ folgt $l(s) = l'(s')$. Also insbesondere $p \in l(s)$ gdw. $p \in l'(s')$. Damit gilt $K, s \models p$ gdw. $K', s' \models p$.

IS: Wir zeigen hier nur den Fall $\varphi = EG\varphi_1$.

Gelte $K, s \models EG\varphi_1$. Dann gibt es einen Pfad

$$\pi = s_0 s_1 s_2 \ldots \text{ der in } s \text{ beginnt } (s = s_0)$$

so dass $K, s_i \models \varphi_1$ für alle $\imath \in \mathbb{N}$. Mit dem vorherigen Lemma gibt es dann einen entsprechenden Pfad

$$\pi' = s'_0 s'_1 s'_2 \ldots \text{ der in } s' \text{ beginnt } (s' = s'_0) \ ,$$

d.h. $(s_i, s'_i) \in R$ für alle $i \in \mathbb{N}$. Mit Induktion gilt für φ_1:

$$K, s_i \models \varphi_1 \text{ gdw. } K', s'_i \models \varphi_1 \ .$$

Hier angewandt erhalten wir $K', s'_i \models \varphi_1$ für alle $i \in \mathbb{N}$. Wir haben also einen Pfad π' gefunden, der zeigt, dass $K', s' \models EG\varphi_1$ gilt.

Die Rückrichtung kann analog bewiesen werden. □

6.2.7 Satz (Satz von Hennessy & Milner, '85)

$$K \approx K' \quad \text{gdw.} \quad \forall \varphi \in CTL\colon K \vDash \varphi \text{ gdw. } K' \vDash \varphi$$